Improving Simulated Annealing Performance by means of Automatic Parameter Tuning

Pablo CABRERA-GUERRERO[1,*], Guillermo GUERRERO[1], Jorge VEGA[2], Franklin JOHNSON[3]

[1] Pontificia Universidad Católica de Valparaíso, CHILE
pablo.cabrera.g@mail.pucv.cl
* *Corresponding author*

[2] Universidad de Antofagasta, Escuela de Ingeniería Electrica, CHILE
jvega@antofa.cl

[3] Universidad de Playa Ancha, CHILE
fjohnson@upla.cl

Abstract: A common problem when using (meta)-heuristic techniques to solve complex combinatorial optimization problems is related to parameters tuning. Finding "the right" parameter values can lead to significant improvements in terms of best solution objective value found by the heuristic, heuristic reliability and heuristic convergence, among others. Unfortunately, this is usually a tedious and complicated task if done manually. In this paper, we propose a framework that is based on Genetic Programming to fine-tune a key parameter of the well-known Simulated Annealing (SA) algorithm. Experiments on a set of small instances of the Facility Location Problem with capacity constraints are performed. Results show that automatically adjusting a key parameter in SA by means of Genetic Programming leads to an average value of the obtained solution that is closer to the optimal solution than the average value obtained by the simple SA algorithm with *a priori* selected values. More important, standard deviation of the algorithm is greatly improved by our approach which makes it much more reliable if time limitations are imposed.

Keywords: Genetic Programming, Simulated Annealing, Combinatorial Optimization, Automatic Parameter Tuning

1. Introduction

Optimization algorithms have been widely studied in literature. While exact methods are very efficient in solving small and medium size problems, they fail on finding (nearly) optimal solutions for large scale optimization problems [1]. When exact algorithms cannot be used directly on such large scale complex optimization problems, heuristic methods are very helpful. During the past five decades an increasing number of (meta-)heuristics have been developed to provide "good" solutions within an acceptable computational time. Although (meta-)heuristic techniques cannot ensure optimality (as exact method does), they have demonstrated to be very effective on solving complex large-scale combinatorial optimization problems.

One key element in the success of a (meta-)heuristic method is its parameters configuration. It has been shown that fine parameter tuning can lead to great improvements on the final solution quality as well as on the technique behaviour [2,3,4,5]. Unfortunately, fine tuning task is quite time consuming and problem-specific. Automatic parameter tuning has been widely studied in the literature (see [4,5,6]). In this paper we propose a genetic programming [7] algorithm to adjust a key parameter of a well-known local search heuristic namely Simulated Annealing (SA) [12]. We test our approach on a set of instances of the Capacitated Facility Location problem (CFLP), obtained from the OR Library [8].

The idea is to let Genetic Programming finds a good value of a key parameter in SA which leads to an improvement in both final objective function value found by the heuristic method and heuristic behaviour (reliability, convergence, etc).

2. Simulated Annealing

Simulated Annealing (SA) algorithm was firstly introduced in [10, 11, 12]. It can be defined as a local search (meta-)heuristic. Theoretically global optimal solutions can be found by SA, however unlimited time is needed, which makes this feature impractical. The method is inspired by thermodynamic systems where concepts such as *energy*, *state* and *temperature* are adapted to make it works in combinatorial optimization framework. The algorithm has been extensively studied and applied on a wide range of complex combinatorial optimization problems [13, 14, 15].

The inputs of the algorithm are as follows: t_0 is the initial temperature. The *maxIter*

parameter corresponds to the total number of iteration the SA algorithm would perform at each run. Additionally, the algorithm needs to define a neighbourhood move, so we can generate (or move) from current solution (state) to another. In this paper, we define the neighbourhood of a solution s, $N(s)$ as the set of all $s' \in S$, with S the set of feasible solutions, such that s' only differs from s in one client allocation, i.e. all clients but one in s' are allocated as in s. For instance, let $s^{\square}=\{1,1,3,1,2\}$ be a solution for a problem with 5 clients and 3 potential warehouses, where clients 1, 2, and 4 are allocated to warehouse 1, client 3 is allocated to warehouse 3 and client 5 is allocated to warehouse 2. Then, let $s^1=\{1,2,3,1,2\}$ and $s^2=\{3,1,3,1,2\}$ be neighbours of s. As we can see, s^1 only differs from s in client 2 allocation, which is now assigned to warehouse 2. Same situation occurs with s^2 where client 1 allocation changes.

The SA algorithm starts with a randomly generated solution $s_{current}$ (or state). After that, a neighbour solution $s_k = N(s_{current})$ is generated, where $k < maxIter$ denotes the current iteration. Once the neighbour solution is generated, the change in objective function values when moving from $s_{current}$ to s_k, $\Delta E = cost(s_k) - cost(s_{current})$ is calculated. If $\Delta E < 0$ then the neighbour s_k is accepted and the current solution is updated. If $\Delta E \geq 0$ then the neighbour solution s_k is accepted with a probability

$$P(\Delta E) = e^{\frac{-\Delta E}{temp}} \quad (1)$$

It is clear that acceptance probability $P(\Delta E)$ depends on the temperature parameter *temp*. Usually, *temp* varies over the algorithm execution, although some constant values have proven to be effective in finding good solutions for some combinatorial optimization problems. As the *temp* variable *cools down*, worst solutions are not longer accepted, which provokes that the algorithm converges to a locally optimal solution. The *temp* variable cools down according to an *annealing schedule*. Commonly used annealing schedules are

$$temp_k = \alpha \, temp_{k-1} \quad (2)$$

with α in the range $[0,1]$ and

$$temp_k = \frac{temp_{k-1}}{1 + \beta temp_{k-1}} \quad (3)$$

with $\beta \ll temp_0$ [11,16,17]. The algorithm ends when either no further improvements can be made or the maximum number of iterations $maxIter$ is reached.

In this paper, genetic programming is used to optimize annealing schedules used within our Simulated Annealing algorithm. Next two sections shows how we combine genetic programming and simulated annealing to solve the CFLP problem.

3. Genetic Programming

Since mid-sixties when [18], [19] and [20] separately developed ideas based on Darwin's findings, evolutionary computing has been a very fruitful research area in artificial intelligence and, particularly, in optimization. Nature inspired algorithms have been developed mainly in the last two decades to tackled difficult optimization problems. One evolutionary strategy proposed by [7] in early nineties is Genetic Programming. A distinctive feature of this strategy is that it attempts to apply evolutionary strategies on computational programs rather than optimization or classification problems, as previously proposed evolutionary strategies do.

In [7] the author claims that the process of solving optimization problems can be reformulated as a search for a highly fit individual computer program in the space of possible computer programs. In this way the author changes the focus from the solutions of the optimization problem to computer programs which lead to high quality solutions of the optimization problem. This is substantially different from other evolutionary strategies such as genetic algorithm or swarm intelligence. Furthermore, [7] states that when viewed in this way, the process of solving these problems becomes equivalent to searching a space of possible computer programs for the fittest individual computer program. In particular, the search space is the space of all possible computer programs composed of functions and terminals appropriate to the

problem domain. Since individual representation and, consequently, search space are different, specific strategies to seek for the fittest individual must be considered.

In genetic programming, populations of hundreds or thousands of computer programs are genetically bred. This breeding is done using the Darwinian principle of survival and reproduction of the fittest along with a genetic recombination (crossover) operation appropriate for mating computer programs. At the end, a computer program that (approximately) solves a given problem would emerge from this combination of selection and genetic operations [7]. To do that, a common representation used in genetic programming is by means of parse trees structure similar to the ones used by compilers, being a link between user-level programming languages and low-level machine specific code. Using parse trees has advantages since it prevents syntax errors, which could lead to invalid individuals, and the hierarchy in a parse tree resolves any issues regarding function precedence [21].

Individuals in genetic programming are adaptive computational programs which are hierarchically structured. Size and content of such programs will dynamically change during the algorithm execution. The key elements of an individual are its genes which are organized as chromosomes (code). Following with the tree structure of the individuals, there are two types of genes: functions and terminals [22]. Terminals, in tree terminology, are leaves (nodes or *points* without branches) while functions are points with children. Figure 1 shows an example of a very simple individual which returns $3 \times (8+6)$.

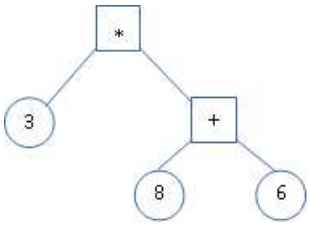

Figure 1. Step 1. Example of an individual in GP.

More complex functions such as *sen()*, *cos()*, *exp()*, AND, OR, NOT as well as structures such as *IF-THEN-ELSE*, *FOR*, *REPEAT*, among others, can be used within genetic programming framework.

In genetic programming, we need to ensure that the a priori defined set of terminals and functions should be selected so as to satisfy the requirements of *closure* and *sufficiency*. The closure property requires that each of the functions in the function set be able to accept, as its arguments, any value and data type that may possibly be returned by any function in the function set and any value and data type that may possibly be assumed by any terminal in the terminal set. The sufficiency property requires that the set of terminals and the set of primitive functions be capable of expressing a solution to the problem. The user of genetic programming should know or believe that some composition of the functions and terminals he supplies can yield a solution to the problem [7].

Both terminals and primitive functions sets (T and F respectively) must be defined by users. Users need to do this independently of the specific problem to be solved and then other problem specific algorithms, such as Tabu Search, can be considered within this framework. Once sets T and F have been defined, the first population needs to be generated. To do that, an initialization function must be implemented. This function should provide valid and feasible trees. Figures 2, 3 and 4 show an example of how this initialization function works.

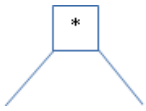

Figure 2. A random function is selected from *F*

Figure 2 shows the first step to create an individual of the first generation. In this step, a function from set F is selected. Since this is the first step, elements from set T of terminals cannot be used.

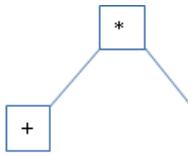

Figure 3. Step 2. A random function is again selected from *F*. In this case a random terminal from *T* could have also been selected.

Next step (see Figure 3) is to set child nodes. Here either elements from T or F can be chosen. If an element from F is selected, no further action is needed in the corresponding branch. If an element from T is selected, we proceed with the same procedure for the new node. The initialization procedure ends when

all tree leaves are set to terminal elements, as in Figure 4. Following this procedure we will obtain a set of different individuals for our first generation. These individuals are different in both size and shape. Different methods have been proposed to the implementation of the initialization procedure (e.g. grow method, ramped half-and-half method and complete method). See [7] for further details on these methods. After the initial population is generated, normalised fitness is calculated [7, 9] in order to identify those individuals which have a larger probability of being considered into the next generation.

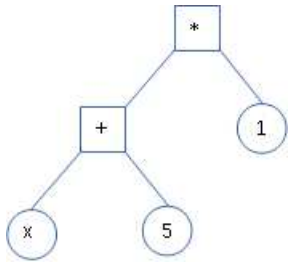

Figure 4. Step 3. Random terminals are selected to complete remaining tree leaves.

As in other evolutionary techniques, the better the fitness, the more probability of being passed on to the next generation. Based on that probability, crossover as well as mutation operations are performed on those promising individuals to generate the next generation of the population. Also, a small portion of the best individuals of the population is selected to be passed on to the next generation without any change. This is call *reproduction*. Apart from reproduction, the simplest operation is the mutation. In this paper mutation is performed as follows: Given an individual (tree) i, a randomly selected function node is replaced by a new function chosen from the set F. As we can see in Figure 5, shape of the tree remains with no change after the mutation process, although the resulting tree is a new individual that belongs to the next generation.

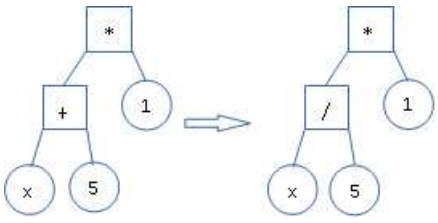

Figure 5. Example of the mutation operation.

In this paper crossover is also considered to generate new individuals for the next generation. It is performed as follows: Given two parents selected from the current generation based on their normalised fitness, a node at each parent is randomly selected. The selected nodes are the crossover points. Sub-trees below the crossover points are then exchanged as in Figure 6.

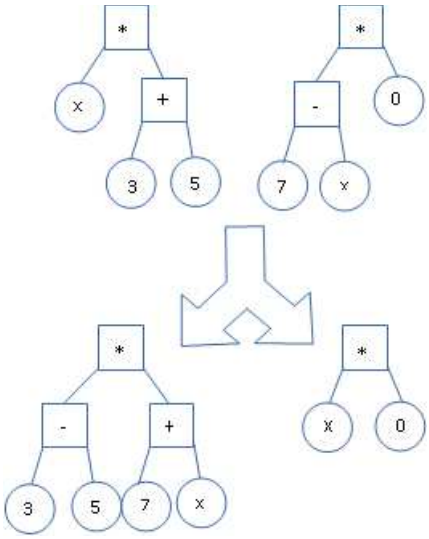

Figure 6. Example of the crossover operation. Parent combination leads to two off-springs.

As in any other evolutionary algorithm, genetic programming ends when a predefined criterion is met. In our case, the maximum number of generations is the termination criterion.

4. Proposed Framework

As we mentioned before, in this paper genetic programming is used to define the annealing schedule function. Although decreasing functions are commonly used to implement annealing schedules, in this paper we do not restrict the algorithm to such functions, but include constant values as well as increasing functions. Thus, hereafter we denote $temp_{k+1}$ variable in SA also as, the *annealing function*.

In order to generate the annealing function expressions, we define set $F=\{+,-,\div,mod,\times,\sqrt{}\}$ as the set of functions to be used by genetic programming and set $T=\{R,temp_k,temp_{max},k\}$ as the set of terminals. Using elements of these two sets, genetic programming is able to generate mathematical expressions for the annealing function. Figure 7 shows the proposed framework that combines genetic programming and SA.

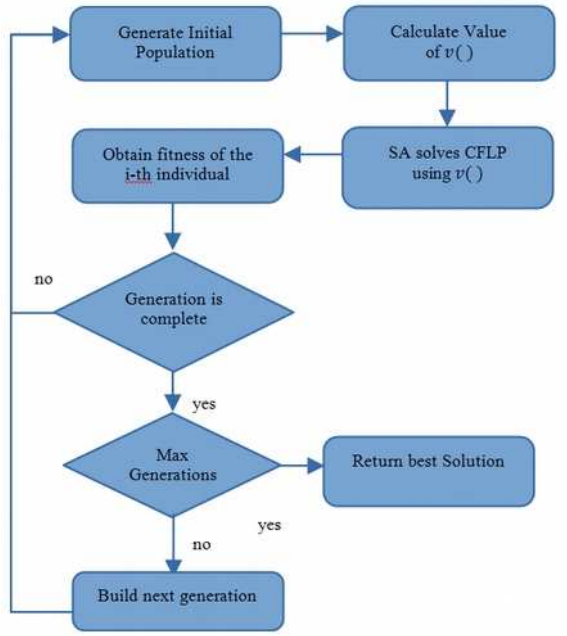

Figure 7. Proposed GP + SA framework to solve the CFLP problem.

We need to point out at this stage that the proposed framework does not depend on the problem that is going to be solved nor does on the (meta-)heuristic genetic programming is combined with. This means that a similar framework can be used considering (meta-)heuristics such as particle swarm optimization or simple local search algorithms. We only need to identify the key parameter of such heuristics so we can optimize it using genetic programming.

As Figure 7 shows, the SA algorithm is performed for each individual of the population in the genetic programming algorithm. Parameter v_i is calculated for each individual i based on its corresponding tree structure. Next generation is built using evolutionary operators such as the ones described before in this paper (reproduction, crossover and mutation). After the maximum number of generations is reached, the best solution found so far is returned.

5. Capacitated Facility Location Problem

In this paper the well-known capacitated facility location problem (CFLP) is used as a test problem. The CFLP consists on locating plants, warehouses, and distribution centres among a set of available locations and allocating customers to such facilities. Here we consider a situation where a single distribution centre serves a set of warehouses. Customers are served by such warehouses. The goal is to find a sub-set of warehouses that allows us to serve all customers minimising the total system cost. Each customer (or cluster) is served only by one warehouse. Customers are uniformly distributed within a limited area. The problem considers the location cost (i.e., the cost associated with opening a specific warehouse) and the allocation cost (i.e., the cost related to transportation of a specific amount of products from a warehouse to a customer) [1]. The mathematical model for the CFLP is presented as follows:

$$min \sum_{i=1}^{N} (F_i \times X_i) + \sum_{i=1}^{N} \sum_{j=1}^{M} (C_{ij} d_j Y_{ij}) \quad (4)$$

$$\sum_{j=1}^{M} d_j Y_{ij} \leq I_i^{cap} X_i, \ \forall i = 1,2,\ldots, N \quad (5)$$

$$\sum_{i=1}^{N} Y_{ij} = 1, \ \forall j = 1,2,\ldots, M \quad (6)$$

$$Y_{ij} \leq X_i \ \forall i = 1,2,\ldots, N, \ \forall j = 1,2,\ldots, M \quad (7)$$

$$X_i, Y_{ij} \in \{0,1\} \ \forall i = 1,\ldots, N; \forall j = 1,\ldots, M \quad (8)$$

Equation (4) is the total system cost. The first term is the fixed setup and operating cost when opening warehouses. The second term is the daily transport cost between warehouse and customers which depends on the customer demand d and distance C_{ij} between warehouse i and customer j. Inequality (5) ensures that total demand of warehouse i will never be greater than its capacity I_i^{cap}. Equations (6) and (7) ensures that customers are served by only one warehouse. Finally, Equation (8) states integrality $(0-1)$ for the binary variables X_i and Y_{ij}.

6. Computational Results

In order to evaluate the performance of our approach, we consider a set of 8 well-known CFLP instances obtained from the OR Library [23], namely cap61, cap62, cap63, cap64, cap71, cap72, cap73 and cap74. We select these instances as they have a known optimal solution. We perform experiments on an Intel i5 processor running on Linux (Ubuntu 14.02) with 8GB of memory. All tested algorithms were implemented in Java 7. We need to state at this point that there exists very efficient exact algorithms that are able

to solve these small instances to optimality in only few seconds. It is clear that we do not want to compete with such methods, as heuristic methods must be used when such exact methods fail in finding optimal solutions. Instead, we want to prove whether our approach improves results obtained by the SA itself.

Thus, we first apply the SA over the entire set of instances. SA is performed 350 times over each instance. Figure 8 shows the behaviour of SA for one specific run on the instance cap61. As we can see, SA reaches the optimum only twice within the allowed time per run.

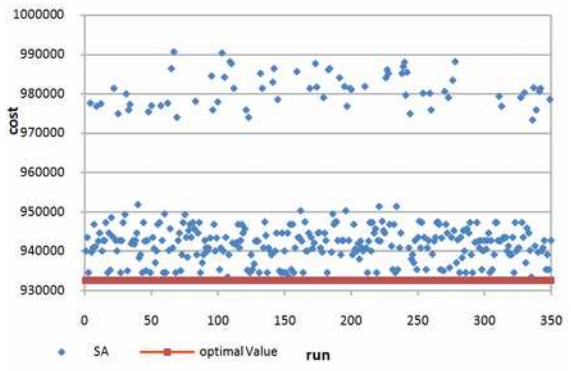

Figure 8. Results obtained for the SA algorithm after 350 runs, for instance cap61. Red line is the optimal solution.

Although SA performs generally well on all these instances (it finds the optimum for all of them if no time limit is imposed), average of the current solution value and its standard deviation as well as its convergence could still be improved as variation in the current solution value is too large provoking that the algorithm becomes both less reliable and slower.

We then perform several experiments on the same set of instances using now our combined approach of SA and GP.

As we can see in Figure 9, when SA is combined with GP, the obtained results are better in terms of average solution value, standard deviation and convergence. While average of the solution values is clearly closer to the optimal solution value, its variation is much less than the one obtained when only SA is applied. Table 1 shows the obtained results for all the instances considered in this study

Column *Instance* in Table 1 is the instance name that is considered. Then columns *AVG* show the average value obtained by both the SA and GP + SA algorithms for each instance. Column *SD* shows the standard deviation as a percentage of the optimal solution of each instance. Columns GAP is the difference, as a % of the optimal solution value, between the average value obtained by the algorithms and the optimal solution of each instance.

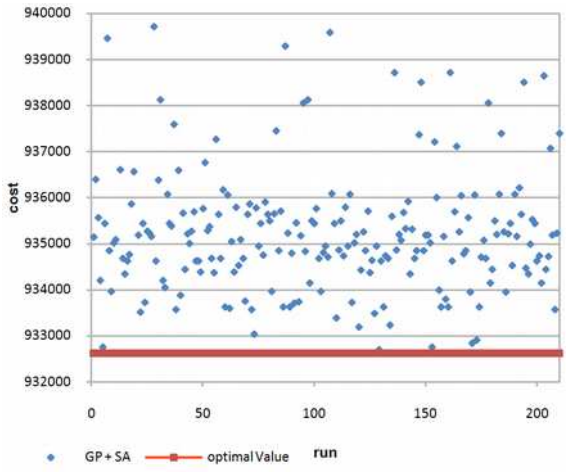

Figure 9. Results obtained for the proposed GP + SA algorithm after 210 runs, for instance cap61. Red line is the optimal solution.

As we can see, in average, the genetic programming when combined with the SA algorithm is able to improve the obtained results in more than 2%. Moreover, the standard deviation is greatly improved by using GP along with SA.

This is especially important for large instances where the *any-time* behaviour is very important.

Table 1. Obtained results for both the simple SA and our GP+SA algorithms for all the considered instances.

Simulated Annealing			
Instance	**AVG**	**SD**	**GAP**
cap61	947,429.770	2%	98.44%
cap62	997,807.802	1.32%	97.99%
cap63	1,045,489.044	1.51%	96.99%
cap64	1,106,129.655	1.83%	94.53%
cap71	940,887.741	0.44%	99.12%
cap72	993,318.233	0.64%	98.44%
cap73	1,035,588.771	0.88%	97.59%
cap74	1,085,484.459	1.24%	95.35%
	Average	**1.20%**	**97.31%**

GP + SA			
Instance	**AVG**	**SD**	**GAP**
cap61	935,198.470	0.14%	99.72%
cap62	986,218.591	0.23%	99.15%
cap63	1,028,318.877	0.42%	98.61%
cap64	1,081,126.701	0.66%	96.72%
cap71	936,001.933	0.18%	99.64%
cap72	986,959.779	0.26%	99.07%
cap73	1,024,116.414	0.43%	98.68%
cap74	1,067,882.256	0.81%	96.92%
	Average	**0.39%**	**98.56%**

7. Conclusions and Future Work

In this paper a genetic programming algorithm is used to automatically fine tune a key parameter of a Simulated Annealing algorithm, namely annealing function, v . The proposed approach is tested on small and medium size instances of the facility location problem with capacity constraints. Results shown that adjusting function v of the SA algorithm by using Genetic Programming leads to a rapid convergence to optimal solutions and reduces variance of the current solution objective function value. This means that SA method becomes more reliable as its *any-time* behaviour is better. Convergence of the SA algorithm is greatly improved as less iterations are required to find optimal solution for all tested instances. Furthermore, the fact that average solution value of our approach is closer to optimal solution in all instances makes it very attractive if only limited time is provided to solve the problem. This is because we known that the solution provided by our approach will be likely better than the one provided by the simple SA at any time during the execution of the algorithm.

As future work, heuristic methods other than SA could be tested within the framework presented here. In particular other local search algorithms such as Tabu Search or Variable Neighbourhood Search could be considered. Moreover, the proposed approach could be tested on large-scale combinatorial optimization problems so we can evaluate its performance when a large number of decision variables is considered.

REFERENCES

1. CABRERA, G., E. CABRERA, R. SOTO, J. M. RUBIO, B. CRAWFORD, F. PAREDES, **A Hybrid Approach using an Artificial Bee Algorithm with Mixed Integer Programming Applied to a Large-scale Capacitated Facility Location Problem**, Mathematical Problems in Engineering, vol. 2012, 2012.
2. GLOVER, F., M. LAGUNA, **Tabu Search** . Norwell, MA, USA: Kluwer Academic Publishers, 1997.
3. HOOS, H. **Automated Algorithm Configuration and Parameter Tuning**, in Autonomous Search, Y. Hamadi, E. Monfroy, and F. Saubion, Eds. Springer Berlin Heidelberg, 2012, pp. 37-71.
4. VERA-PÉREZ, O. L., A. MESEJO-CHIONG, A. JAUME-I-CAPÓ, M. GONZÁLEZ-HIDALGO, **Automatic Parameter Configuration: A Case Study on a Rehabilitation Oriented Human Limb Tracking Algorithm**, Studies in Informatics and Control , vol. 23, no. 3, 2014, pp. 313-323.
5. MORI, M., R. KOBAYASHI, M. SAMEJIMA, N. KOMODA, **Cost-benefit Analysis of Decentralized Ordering on Multitier Supply Chain by Risk Simulator**, Studies in Informatics and Control , vol. 21, no. 1, 2012, pp. 75-83.
6. CRAWFORD, B., C. VALENZUELA, R. SOTO, E. MONFROY, F., PAREDES, **Parameter Tuning of Metaheuristics using Metaheuristics**, Adv. Sc. Letters, vol. 19, no. 12, 2013, pp. 3556-3559.
7. KOZA, J. **Genetic Programming as a Means for Programming Computers by Natural Selection**. Kluwer Academic Publishers, vol. 4, no. 2, 1994.
8. BEASLEY, J. **An Algorithm for Solving Large Capacitated Warehouse Location Problems**, European Journal of Operational Research, vol. 33, no. 3, 1998, pp. 314-325.
9. BÖLTE, A., U. W. THONEMANN, **Optimizing Simulated Annealing Schedules with Genetic Programming**, European Journal of Operational Research , vol. 92, no. 2, 1996, pp. 402-416.
10. KIRKPATRICK, S., **Optimization by Simulated Annealing: Quantitative Studies**. Journal of Statistical Physics, vol. 34, no. 5-6, 1984, pp. 975-986.
11. KIRKPATRICK S., C. D. GELATT, M. P. VECCHI, **Optimization by Simulated Annealing**. Science, vol. 220, no. 4598, 1983, pp. 671-680.
12. METROPOLIS, N., A. W. ROSENBLUTH, M. N. ROSENBLUTH, A. H. TELLER, E. TELLER, **Equation of State Calculations by Fast Computing Machines**. Journal of Chemical Physics vol. 21, no. 6, 1953, pp. 1087-1092.

13. CABRERA, G., S. RONCAGLIOLO, J. P. RIQUELME, C. CUBILLOS, R. SOTO, **A Hybrid Particle Swarm Optimization-Simulated Annealing Algorithm for the Probabilistic Travelling Salesman Problem**. Studies in Informatics and Control vol. 21, 2012, pp. 49-58.

14. LIU, S., W. H. WU, C. C. KANG, W. C. LIN, Z. CHENG, **A Single-machine Two-agent Scheduling Problem by a Branch-and-Bound and Three Simulated Annealing Algorithms**. Discrete Dynamics in Nature and Society, vol. 2015, 2015, Article ID 681854.

15. QIN, J., H. XIANG, Y. YE, L. NI, **A Simulated Annealing Methodology to Multiproduct Capacitated Facility Location with Stochastic Demand**. The Scientific World Journal vol. 2015, 2015, Article ID 826363.

16. WILHELM, M., T. WARD, **Solving Quadratic Assignment Problems by Simulated Annealing**. IIE Transactions, vol. 19, no. 1, 1987, pp. 107-119.

17. FOGEL, L. J., A. J. OWENS, M. J. WALSH, **Artificial Intelligence through Simulated Evolution**. New York, USA: John Wiley, 1966.

18. HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**. Ann Arbor, MI, USA: University of Michigan Press, 1975.

19. RECHENBERG, I., **Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution**, Ph.D. dissertation, TU Berlin, 1971.

20. EGGERMONT, J., **Data Mining using Genetic Programming: Classification and Symbolic Regression**, PhD dissertation, Universiteit Leiden, September 2005.

21. WALKER, M., **Introduction to Genetic Programming**, Computer Science Department, Montana State University, Tech. Rep., December 2001.

22. BEASLEY, J., **OR-Library: Distributing Test Problems by Electronic Mail**, Journal of the Operational Research Society , vol. 41(11), 1990, pp. 1069-1072.