Benchmarking Optimization Algorithms: An Open Source Framework for the Traveling Salesman Problem

©IMAGESTATE

Thomas Weise
UBRI, School of Computer Science and Technology, University of Science and Technology of China, Hefei, CHINA

Raymond Chiong
Faculty of Science and Information Technology, The University of Newcastle, Callaghan, AUSTRALIA

Jörg Lässig
Department of Computer Science, University of Applied Sciences Zittau/Görlitz, Görlitz, GERMANY

Ke Tang
UBRI, School of Computer Science and Technology, University of Science and Technology of China, Hefei, CHINA

Shigeyoshi Tsutsui
Department of Management and Information Science, Hannan University, Matsubara, JAPAN

Wenxiang Chen
Computer Science Department, Colorado State University, Fort Collins, USA

Zbigniew Michalewicz
School of Computer Science, The University of Adelaide, Adelaide, AUSTRALIA
Polish-Japanese Institute of Information Technology and the Institute of Computer Science, Polish Academy of Sciences, POLAND

Xin Yao
UBRI, School of Computer Science and Technology, University of Science and Technology of China, Hefei, CHINA
CERCIA, School of Computer Science, The University of Birmingham, Birmingham, UK

Abstract—We introduce an experimentation procedure for evaluating and comparing optimization algorithms based on the Traveling Salesman Problem (TSP). We argue that end-of-run results alone do not give sufficient information about an algorithm's performance, so our approach analyzes the algorithm's progress over time. Comparisons of performance curves in diagrams can be formalized by comparing the areas under them. Algorithms can be ranked according to a performance metric. Rankings based on different metrics can then be aggregated into a global ranking, which provides a quick overview of the quality of algorithms in comparison. An open source software framework, the *TSP Suite*, applies this experimental procedure to the TSP. The framework can support researchers in implementing TSP solvers, unit testing them, and running experiments in a parallel and distributed fashion. It also has an evaluator component, which implements the proposed evaluation process and produces detailed reports. We test the approach by using the *TSP Suite* to benchmark several local search and

Digital Object Identifier 10.1109/MCI.2014.2326101
Date of publication: 14 July 2014

evolutionary computation methods. This results in a large set of baseline data, which will be made available to the research community. Our experiments show that the tested pure global optimization algorithms are outperformed by local search, but the best results come from hybrid algorithms.

I. Introduction

In the field of metaheuristic optimization, experimentation is perhaps the most important tool to assess and compare the performance of different algorithms. However, most studies limit themselves to presenting means and standard deviations of final benchmark results. This article proposes an experimental procedure that can provide deeper insights into an algorithm's behavior and more holistic comparisons. We implement this procedure exemplarily for the Traveling Salesman Problem (TSP) in a software framework called the *TSP Suite*, which eases algorithm implementation, parallel and distributed experimentation, as well as automatic evaluation. We then use the *TSP Suite* to compare the performance of several local search methods and members of the main Evolutionary Computation (EC) algorithm families [1], e.g., Evolutionary Algorithms (EAs), Memetic Algorithms (MAs), Estimation of Distribution Algorithms (EDAs), and Ant Colony Optimization (ACO).

The TSP [2–4] is one of the most well-known combinatorial optimization tasks. A TSP is defined as a fully-connected graph with n nodes. Each edge has a weight, representing the distance. A candidate solution is a tour that visits each node in the graph exactly once and returns back to its starting node. The objective function f, subject to minimization, is the sum of the weights of all edges in the tour, i.e., the total tour length. This optimization version of the TSP is *NP-hard* [4]. It has been researched for decades, and algorithms that can exactly solve instances with tens of thousands of nodes and approximate the solution of million node problems with an error of less than one part per thousand within feasible time exist [5].

Still, the TSP remains an interesting subject for research for two reasons. First, the problem is easy to understand. Many results and standard benchmark instances with known optima are available. This makes the problem ideal for testing new approaches, be it general algorithms or improvements such as adaptation strategies. Second, while current experimentation approaches only focus on singular results, investigating the *behavior* and *progress* of TSP solvers is an equally important issue and may lead to the development of better solvers with better results.

Experiments for analyzing the behavior of an algorithm over runtime are cumbersome. They generate much data and their manual evaluation can take more time than the algorithm implementation itself. The COmparing Continuous Optimizers (COCO) [6] system for numerical optimization, used in the Black-Box Optimization Benchmarking (BBOB) workshops, is one of the first approaches aiming to reduce the workload of an experimenter by automatizing most of the steps involved. Its evaluation procedure generates statically structured papers that contain diagrams with runtime behavior information. The necessary data is automatically collected from automatically executed experiments.

UBCSAT [7], on the other hand, is an experimental framework for satisfiability (SAT) problems. It focuses on a specific algorithm family, the stochastic local search (SLS) [8]. SLS methods can be implemented by utilizing a trigger architecture defined on top of a default algorithm structure. In COCO, the objective function will automatically gather log data before returning its result to the algorithm. In UBCSAT, this is done in its trigger architecture. The trigger architecture can also compute complex statistics online and provide them to the running algorithm. COCO and UBCSAT have in common that they both explore algorithm behavior over runtime instead of focusing only on final results. They are thus different from contests such as the *DIMACS* challenge [9] or the Large-Scale Global Optimization [10] competitions.

In this paper, we introduce a new experimental procedure for evaluating and comparing optimization algorithms. Different from COCO, we prescribe a more general data collection scheme (see Section II-A). The proposed evaluation process makes use of diagrams similar to those in COCO and UBCSAT (see Section II-B), but it does not stop there: It combines the results from different evaluation criteria and constructs text-based discussions and conclusions, resulting in comprehensive reports instead of rigidly structured papers.

The focus of COCO, UBCSAT, and our proposed approach is on *analyzing and comparing* concrete algorithm setups. They complement frameworks such as the Sequential Parameter Optimization Toolbox (SPOT) [11], which *finds* good setups via efficient automatic parameter tuning. One could, for example, use the SPOT to configure an algorithm before analyzing it with COCO.

We first discuss our approach to experimentation with optimization algorithms in general and for TSP solvers in particular (Section II). It will become clear that, in order to gain a deeper insight into the behavior of an algorithm, a very large amount of work is necessary, both for measurement and evaluation. Next, we introduce the *TSP Suite*, an open source Java software framework for experimentation with TSP solvers, in Section III. The *TSP Suite* implements our experimental procedure and provides automatic data collection and benchmarking capabilities, as well as a component for automatically evaluating the gathered data and comparing the performance of different algorithms. This suite allows researchers working on the TSP to conduct more comprehensive experiments in a shorter amount of time and significantly reduces the work needed to gain valuable results and insights.

The *TSP Suite* also contains implementations of several different TSP solvers, including local search algorithms, EAs, MAs, EDAs, and ACO methods. We report the results of a large set of experiments with these methods in Section IV as a proof-of-concept for the *TSP Suite* and will provide all collected data on the web so that other researchers may use it for comparison. The system can be downloaded from http://www.logisticPlanning.org/tsp/.

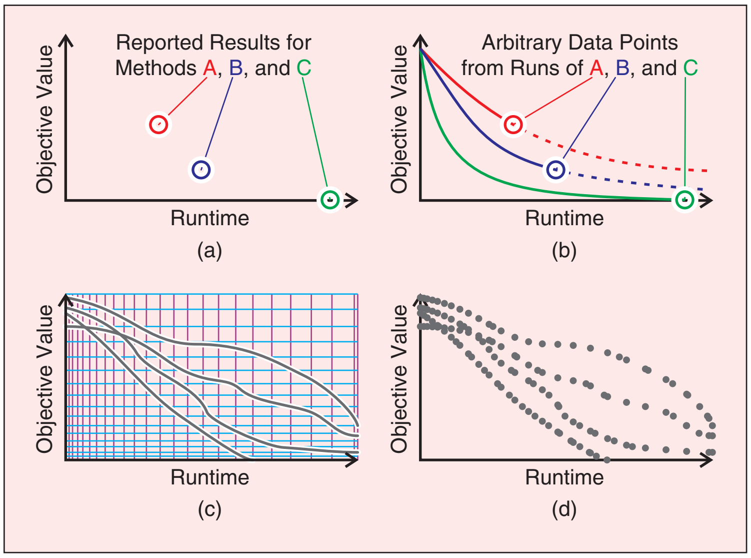

FIGURE 1 The problem of reporting singular results and how to collect multiple data samples per run. (a) Reported results in the literature: Methods A, B, and C appear to be viable alternatives for different available computational budgets. (b) Potential actual behavior of anytime algorithms: Method C is better. (c) Strategically placed horizontal and vertical cuts where log points from different runs of an anytime algorithm are to be taken. (d) The log points caught at these cuts provide sufficient data to reconstruct the original curves.

II. Experimentation with Optimization Algorithms

In this section, we will discuss several issues that arise when experimentally analyzing metaheuristics in general and TSP solvers in particular. Some of these issues, such as the time measures and considerations about performance, may also be relevant to theoretical algorithm analysis [12–14].

A. Data Collection

Before beginning an experiment, a benchmark dataset must be chosen. There must also be a definition of how to measure an algorithm's *performance*. Such a definition will always be based on runtime, so how to measure the time must be clarified.

1) Benchmark Datasets

The most well-known benchmark dataset for the TSP is the *TSPLib* [15]. This library contains 110 instances of symmetric TSPs with scales n ranging from 14 to 85900. There are 93 instances with less than 2000 nodes. For each instance, the globally optimal tour is known.

Another dataset, based on the *DIMACS* 2008 challenge [9], was used in extensive experiments with the results [16, 17] published in [4]. This dataset contains instances with n between 1000 and 1000000, not all of which have been provably solved to optimality. The website of Cook [18] holds further large-scale TSP instances.

The decision on which benchmark set to use depends on the goal of the research: The *DIMACS* instances and similar large-scale problems allow researchers to explore the limit of what TSP solvers can achieve and to push this boundary forward. If the goal is to perform many experiments for statistical analysis repeatedly, with different parameter settings, then the *TSPLib* is the better choice, since its (mainly) smaller instances allow for faster experiments.

2) What Is Performance?

Relevant literature like the webpage and chapters on the *DIMACS* challenge [9] as well as the Large-Scale Global Optimization competition [10] typically report tuples of (benchmark instance, result, runtime) as outcomes of experiments, as shown in Figure 1a. However, most of the common metaheuristics are *anytime algorithms* [19]. Anytime algorithms can provide an approximate solution for a problem at any point during their runtime and the approximation quality may improve if more time is given. If such an algorithm is applied to the TSP, the point at which the algorithm is terminated and its result is reported becomes an arbitrary choice of the experimenter.

This means that the reported results from Figure 1a may actually be singular snapshots of the performance curves depicted in Figure 1b. Based on Figure 1a, one may assume that the depicted algorithms A, B, and C are viable alternatives depending on the available computational budget. Figure 1b debunks this assumption by uncovering that method C always has a better approximation quality than the other two. *Proper experimentation should thus avoid reducing algorithm performance to singular points*.

The other extreme, to record all the solution improvements an algorithm makes, is not a feasible option. There could be millions of such events, leading to unmanageably large log files. Thus, an intermediate approach is necessary, which collects a limited amount of data but sufficient information to approximate an algorithm's runtime behavior.

Performance can be defined as the solution quality (tour length) that can be reached within a given time frame *or* as the time needed to reach a given solution quality. The former is very commonly seen in benchmarking [10], but the latter has several advantages [6]. We suggest using both methods and to strategically define a fixed set of points in time (vertical lines in Figure 1c) *and* goal objective values f_t (horizontal lines) at which "log points" are to be collected, as illustrated in Figure 1d.[1]

3) What Is Time?

Any collected measurement from a run holds one objective value and a value for the elapsed time. The question of how to measure time seems trivial, but it actually has a major impact on the results the evaluation procedure will provide. We can define *four* time measures for TSPs:

a) *Absolute Runtime AT:* Runtime, traditionally, is measured as the *absolute time AT* that has elapsed since the algorithm was started. This has several advantages. For example, many

[1]These log points will not necessarily be exactly on the specified thresholds, as finding a tour of length 107 would, e.g., satisfy a threshold tour length 128.

related papers report CPU times in milliseconds. Also, clock time is a quantity that makes physical sense. Furthermore, the measurements will include all actions performed by the algorithm, be it memory allocations or complex matrix operations. However, CPU times are inherently incomparable since they largely depend on the machine, operating system, and software environment. If a runtime of 30 minutes on an Intel Pentium II processor was reported about ten years ago, this result is basically meaningless today.

Experimentation is perhaps the most important tool to assess and compare the performance of different algorithms.

b) Normalized Runtime NT: One idea to reduce the incomparability of absolute runtime is to *normalize* it with a system-dependent "performance value" Z [9], i.e., to provide a *normalized runtime NT*. Before applying a TSP solver $\mathcal{A}$ to a given problem instance I, we also apply a standardized algorithm $\mathcal{B}$, the Double-Ended Nearest Neighbor Heuristic [20], to I and measure its runtime $Z(I)$. The operations that $\mathcal{B}$ performs are similar to those that any TSP solver will carry out. $Z(I)$ thus should contain most of the system-dependent aspects that would influence the runtime of $\mathcal{A}$, ranging from the processor speed to whether the cache is large enough to hold a whole candidate solution for I. All AT values measured for $\mathcal{A}$ are divided by $Z(I)$ to obtain the corresponding NT values. If the same algorithm $\mathcal{A}$ is executed on two different computers, for instance, the performance curves over NT should still look approximately the same.

c) Function Evaluations FE: In the field of optimization, the runtime of an algorithm is often measured in terms of *function evaluations* (*FE*s), i.e., the total number of constructed solutions passed to the objective function [6, 10]. This measure is entirely independent of the clock time and system effects. However, it does not reveal "hidden complexities" of the algorithms such as the runtime of model updating in an EDA. Moreover, 1 *FE* may have largely different costs in different algorithms. The complexity of creating a tour in ACO is in $\mathcal{O}(n^2)$, for crossover in an EA it may be in $\mathcal{O}(n)$, while a local search that swaps two cities in a tour of known length needs $\mathcal{O}(1)$ steps to obtain and evaluate the new solution. Thus, comparing algorithms based on consumed *FE*s may be grossly unfair.

d) Distance Evaluations DE: The three examples in the previous paragraph have in common that their different complexities of 1 *FE* are related to the number of times city distances are computed. When choosing the next city to move to, an ant in ACO computes a probability value for each not-yet-visited city. This value also depends on the distance to the city, i.e., creating a new tour in ACO takes a number of distance evaluations in $\mathcal{O}(n^2)$. The length of a new tour created by crossover in an EA is the sum of $n \in \mathcal{O}(n)$ distances, while only $8 \in \mathcal{O}(1)$ distances need to be computed in the case of the local search move mentioned above. Counting the number of *distance evaluations* (*DE*s) may thus often be a fairer machine-independent runtime measure for TSP solvers. In other domains, there are similar elementary operations that could be counted, such as variable flips [7] in SAT problems.

In summary, measuring runtime is actually a non-trivial issue. A data point collected from a run of an optimization algorithm on a TSP instance, in our proposed approach, is a five-dimensional tuple of the best achieved objective value f_b and the four time measures *AT*, *NT*, *FE*, and *DE*.

B. Data Evaluation

Carrying out an experiment means to apply the same algorithm to a set of benchmark problem instances, performing several independent runs for each of them. From each run, a list of "log points" is collected, which can then be analyzed to gain insights into the algorithm's performance.

1) Literature Comparison

Comparison with other studies in the literature becomes easy if data is collected as discussed above. If a paper reports results in terms of the arithmetic mean of f_b after a specific time measured in *FE*s, we can look up how long it takes for the benchmarked algorithm to reach the same or better solution quality in mean. Such a comparison is one of the basic requirements asked for by any reviewer. It should be noted that the literature often reports results in terms of runtime measures *AT* or *FE*, which have the drawbacks discussed above. Thus, such comparisons may not be fair, regardless of whether they are done manually or automatically with the *TSP Suite*.

2) Statistical Tests

For each defined runtime or objective value threshold, statistical comparisons between different benchmarked algorithms are possible, although it is normal to only compare the final results of the algorithms.

For this purpose, non-parametric tests like the Mann-Whitney U test should be used, since they make fewer assumptions about the underlying distribution of the measured data. If $N > 2$ algorithms are compared, performing $0.5\,N(N-1)$ tests directly is not advisable. Instead, additional provisions such as (at least) the conservative Bonferroni correction [21] or (better) more sophisticated tests together with post-hoc methods [22, 23] are needed. Statistical tests require the full set of measured data for all compared algorithms and therefore cannot be performed with results from the literature, which are condensed to means or medians.

3) Data Normalization

We often may want to aggregate data over multiple problem instances. The objective values f^* of the globally optimal tours are known for all *TSPLib* instances, but they differ significantly. We use the best objective value f_b that a process has discovered until a given point in time to compute a relative error $F_b = (f_b - f^*)/f^*$. $F_b = 0$ means the globally optimal solution has been found and

The TSP Suite allows researchers to conduct more comprehensive experiments in a shorter amount of time and significantly reduces the work needed to gain valuable results and insights.

$F_b = 1$ means the best discovered solution is twice as long as the optimum. We will refer to F_b as *error* and to corresponding goal thresholds F_t as *goal errors*.

In addition to having different optimal tour lengths, the *TSPLib* instances also differ in terms of their scales n. This makes it hard to draw diagrams aggregating benchmarking information from different problem instances. Such aggregation is necessary, however, since no paper can contain 110 separate figures, which would be impossible to interpret. The COCO/BBOB [6] system often presents the *FE* axes of diagrams scaled with the problem dimension. We found that scaling *FE* and *AT* values with n usually leads to curves similar enough for meaningful aggregation (although we are still looking for a better option here). Since creating an entirely new solution requires n distance evaluations in order to compute the tour length, *DE* can be scaled by n^2. *NT* does not need to be scaled, since the complexity of algorithm $\mathcal{B}$ used for time normalization already contains n.

4) Progress Curves

Based on the collected data points, it is possible to approximate the progress of an algorithm in terms of the median or other quantiles (based on all runs) of the error F_b over a given time measure. An example for such diagrams is given in Figure 5a later in this paper.

5) Estimated Running Time (ERT)

For each of the goal objective values f_t that are specified for the data collection of a given benchmark instance (and the corresponding error F_t), it is possible to compute the estimated running time $ERT_T(F_t)$ needed to attain it (for a time measure T) [6].

The *ERT* can be plotted in two different ways. One can put F_t on the x-axis and *ERT* on the y-axis for fixed benchmark instances (see Figure 5b later in this paper). This shows how the runtime of an algorithm increases as the goal error reduces. Alternatively, a fixed threshold F_t can be chosen, the problem scale n is put on the x-axis, and the mean or median *ERT* for F_t and the benchmark instances of that scale are on the y-axis. This provides information about how the runtime needed to get a given approximation quality increases with n.

6) Empirical (Cumulative) Distribution Function (ECDF)

For a time measure T, the empirical cumulative distribution function $(ECDF_T)$ [6, 7, 24] returns the fraction of runs that have reached a given goal error F_t (normally, $F_t = 0$). It is plotted over the runtime and should, ideally, reach 1 as quickly as possible. Figure 6 later in this paper is an example for *ECDF* diagrams.

7) Curve Comparison

Diagrams that display the *ERT* or *ECDF* are more than just visual aids. However, it is not easy to formalize statements like *"this curve tends to be lower than that one."* One idea to do so is to compare the area under the curve(s) (*AUC*) [25, 26]. Algorithms that can find better solutions faster tend to have smaller areas under their progress and *ERT* curves as well as larger areas under their *ECDF*.

For some problem instances and goal errors F_t, the *ERT* may go to infinity and so would the *AUC*. Here, one can first compare the length on the x-axis for which the *ERT* is infinite. If one algorithm has a shorter section here, it is better. In case of a tie (or if both discover the global optimum and thus have all-finite *ERT*s), the areas are compared and the one with the smaller area is considered as better.

8) Information Aggregation via Ranking

We now can compare algorithms from many different perspectives and, often, findings will be consistent over different statistics. Yet, there should be a formal concept to join them into conclusions. A simple approach here is to rank each algorithm according to each aspect. Let us assume that we compare five algorithms according to their *ECDF* over the *DE*s. The *TSPLib* provides 110 benchmark cases and for each of those, we can draw a diagram. The *AUC*-based comparison will lead to a ranking of the algorithms in each of these diagrams. We then can re-rank the algorithms for $ECDF_{DE}$ according to their median rank over all the individual diagrams. The resulting ranking can now contribute to a "global" ranking, which is a ranking that orders the algorithms according to their median ranks from many different aspects, including, e.g., $ECDF_{DE}$, ERT_{NT}, and progress in terms of *FE* and *NT*. Of course, depending on research goals, the global ranking can also be based on a narrower set of performance metrics.

The ranking approach has the advantage that it can reduce many information sources into a simple conclusion. Such a conclusion would provide a general idea about the performance relationship of different algorithms that can then be further explored by a researcher.

III. The *TSP Suite*

Most studies on the TSP limit their analyses to comparing their results with those from the literature or, at best, using statistical tests on the end results. Thorough experimentation requires a significant amount of work. If it is done by hand, the time needed to evaluate the benchmark results may equal or even exceed the time spent in implementing the TSP solver and running the experiments.

However, as mentioned in the introduction, thorough experimentation is necessary for solid research in metaheuristics. In this section, we present our open source software system: the *TSP Suite*. It is a Java 1.7 framework that assists algorithm developers in implementing and testing their methods, running experiments and collecting data, as well as evaluating and comparing results. We describe the experimental procedure with the *TSP Suite* step-by-step in the following section.

The TSP Suite is a Java 1.7 framework that assists algorithm developers in implementing and testing their methods, running experiments and collecting data, as well as evaluating and comparing results.

A. Implementing the Algorithm

Since the *TSP Suite* is a Java framework, the optimization algorithm to be investigated must be implemented as a Java class (program). This class must be an extended class of a class called `TSPAlgorithm` and implement a method `solve` taking as input an instance of the class `ObjectiveFunction`. This instance provides, among others,

1) a method to compute the tour length of a candidate solution (either in path or adjacency representation [27]);
2) a method to compute the distance between two nodes;
3) a function that returns `true` when the algorithm should terminate, either because the granted computational budget has elapsed or the global optimum has been discovered[2];
4) the random number generator to be used during the run of the algorithm; as well as
5) information about the elapsed runtime and best solution discovered so far.

Additionally, similar to the objective functions used in the COCO framework [6], it automatically gathers all logging information (in memory).

To be executable, the algorithm class must have a specific `main` method, which is a single line of code that can basically be copied from the documentation of the *TSP Suite*. By *optionally* implementing some methods, an algorithm may be extended with typed parameters (such as an integer value for the population size of an EA) that can be passed in via the command line or in a configuration file.

JUnit tests [28], which can automatically apply a TSP solver to some of the benchmark instances and check if it produces invalid results, are provided. In order to unit-test a new algorithm, one additional class with a single one-line method needs to be provided. Although testing cannot guard against errors entirely, it may help to reduce them. The *TSP Suite* comes with extensive documentation on how to implement and test TSP solvers.

There are very few requirements on the algorithm type, structure, and implementation imposed by the *TSP Suite*. This makes it easy to both benchmark existing algorithms and to re-use *TSP Suite*-based algorithms for other purposes.

B. Executing the Experiment

To execute the experiment, the algorithm will be instantiated and applied to all benchmark instances 30 times each by default. A folder structure with one folder per benchmark case and one log file per run will be produced. These log files follow an easy-to-parse text format and contain, among others,

1) the benchmarking data captured according to Sections II-A2 and II-A3;
2) the algorithm name and class;
3) all parameter settings (if typed parameters have been specified);
4) information about the software (Java version, OS) and hardware (processor, available memory) environment;
5) the seed of the random number generator; and
6) if specified, the name, e-mail address, and website of the experimenter.

A log file thus serves as a complete, publishable documentation of a single run, maximizing replicability of experiments.

As mentioned before, there are 110 symmetric *TSPLib* instances and the system conducts by default 30 runs per instance. If each run would use up the maximum time of 1h even on the smallest instances, the experiment could take about 138 days to complete. In order to decrease the runtime, all processors on a machine can be utilized for executing independent runs. From our experience, a complete experiment with a sub-par algorithm takes about 1 week on an 8-core machine. Additionally, several instances of the same program may be executed in a cluster. If their output paths point to the same shared folder, the runs to be performed are automatically divided among them. This way, the workload of the experiment is parallelized and distributed, but not the algorithms themselves. Thus, no additional provisions from the algorithm implementer (apart from not using globally shared variables) are required.

C. Evaluating and Documenting the Results

Once an experiment has been conducted using the *TSP Suite*, the evaluator program can be applied to the output folder(s) with the log files. This evaluator generates a comprehensive report, which includes all the steps detailed in Section II-B. The report is structured into three parts.

The first part contains a description of the TSP, the experiment, and the evaluation process, i.e., a more extensive version of the text in this paper up to the start of Section III. This makes the report a standalone document that can be understood with no further context needed.

The second part of the report is focused on the evaluation of individual algorithms separately. If the evaluator is fed with results from multiple experiments, this part contains one section for each of them. In such a section, the parameter settings of the experiment are listed, several curves are plotted, and an automatic comparison with results from the literature is performed and presented in tabular form.

If multiple experiments have been conducted, a third section is added to the report, which provides statistical algorithm comparisons. Here, curves of the algorithms can be

[2]The default termination criterion is to exhaust $100n^3$ *FEs*, $100n^4$ *DEs*, or 1h of CPU time, or when the optimum is found. This can be changed via the command line of the system.

x = (A,B|**C**,D,E,F,**G**|H,I) $\Longrightarrow$ x' = rev(x, 3, 7) = (A,B|$\underline{\textbf{G},\text{F},\text{E},\text{D},\textbf{C}}$|H,I)

f(x') = f(x) − dist($\overline{BC}$) − dist($\overline{GH}$) + dist($\overline{BG}$) + dist($\overline{CH}$)

(a)

x = (A,B|**C**|D,E,F,**G**|H,I) $\Longrightarrow$ x' = rol(x, 3, 7) = (A,B|$\underline{\text{D},\text{E},\text{F},\textbf{G}}$|$\underline{\textbf{C}}$|H,I)

f(x') = f(x) − dist($\overline{BC}$) − dist($\overline{CD}$) − dist($\overline{GH}$)
+ dist($\overline{BD}$) + dist($\overline{GC}$) + dist($\overline{CH}$)

(b)

x = (A,B|**C**,D,E,F|**G**|H,I) $\Longrightarrow$ x' = ror(x, 3, 7) = (A,B|$\underline{\textbf{G}}$|$\underline{\textbf{C},\text{D},\text{E},\text{F}}$|H,I)

f(x') = f(x) − dist($\overline{BC}$) − dist($\overline{FG}$) − dist($\overline{GH}$)
+ dist($\overline{BG}$) + dist($\overline{GC}$) + dist($\overline{FH}$)

(c)

x = (A,B|**C**|D,E,F|**G**|H,I) $\Longrightarrow$ x' = swap(x, 3, 7) = (A,B|$\underline{\textbf{G}}$|D,E,F|$\underline{\textbf{C}}$|H,I)

f(x') = f(x) − dist($\overline{BC}$) − dist($\overline{CD}$) − dist($\overline{FG}$) − dist($\overline{GH}$)
+ dist($\overline{BG}$) + dist($\overline{GD}$) + dist($\overline{FC}$) + dist($\overline{CH}$)

(d)

FIGURE 2 Examples of the four search operations that modify a candidate solution x in path representation (as permutation) according to two input indices $i,j \in 1..n$, along with efficient methods for computing the objective value $f(x')$ of the resulting new permutation x'. (a) Reversing operator: $\texttt{rev}(x,i,j)$. (b) Left-rotation operator: $\texttt{rol}(x,i,j)$. (c) Right-rotation operator: $\texttt{ror}(x,i,j)$. (d) Swap operator: $\texttt{swap}(x,i,j)$.

plotted together in the same diagrams and are compared by using the *AUC*-approach detailed in Section II-B7. Statistical comparisons of end results and runtimes are performed based on the Mann-Whitney U test with Bonferroni correction. For each of the above aspects, the algorithms are ranked and these rankings are reflected in descriptive texts and conclusions. A final section aggregates all the single rankings and makes a suggestion about which algorithms tend to perform the best in overall. This aggregate combines rankings resulting from the following sources:

1) the aggregated rank for mean *ECDF* over all benchmark instances;
2) separate ranks for *ECDF* over benchmark instances grouped by n in powers of two;
3) the aggregated rank for *ERT* over F_t and all benchmark instances;
4) separate ranks for *ERT* over F_t and over benchmark instances grouped by n in powers of two;
5) the aggregated rank for *ERT* over n;
6) separate ranks for F_b over runtime, over benchmark instances grouped by n in powers of two;
7) separate ranks for F_b over runtime, for each individual benchmark instance; and
8) statistical test results, involving comparison of the final result and the runtime to optimality and to $F_b \leq 0.01$.

All time-dependent statistics are computed and ranked separately for the three time measures *DE*, *FE*, and *NT*. In summary, the aggregated ranking rewards algorithm speed, good results, and the ability to discover global optima.

Each section, result, or diagram is always accompanied with descriptive texts. Additionally, each evaluation step in the last two parts of the report can independently be turned on or off and configured via command line parameters of the evaluator. This way, researchers can adapt the ranking towards their specific research goals. If the main goal is to find algorithms that can find the global optimum, then all modules not focused on that goal can be turned off. The algorithm comparison will then rank the results solely based on this aspect.

As for the output format, the user can choose between XHTML with PNG figures and LaTeX with EPS figures. In the latter case, one can choose among the following document classes: standard article, ACM conference, IEEE article, IEEE conference, and Springer conference. The LaTeX output can be automatically compiled into a PDF format. Its figures can easily be re-used for writing papers and articles. XHTML, on the other hand, has a smoother layout due to not needing page breaks and can easily be published on the web.

IV. Proof-of-Concept: An Experimental Study

We used the *TSP Suite* to conduct an extensive set of experiments with different metaheuristic TSP solvers. We designed these experiments with two goals in mind: 1) to create a large amount of comparison data and algorithm implementations that covers the major EC algorithm families as well as local search methods, and 2) to explore several questions about the performance of these algorithms.

A. Prerequisites

In order to make the comparison fair, we have endeavored to apply the algorithms in similar configurations. All algorithms worked on the path representation [27], where solutions are encoded as permutations of integer numbers. Here, each node has an id in $1\ldots n$ and if id β is listed directly after id α in a permutation, β will be visited directly after α in the tour.

All algorithms except *PACO* and *TEHBSA* (see the next section) used the same four unary search operations (neighborhoods) sketched in Figure 2. If one of these operators creates a new tour x' from an existing tour x with known length $f(x)$, then the length $f(x')$ of x' can be computed in $O(1)$ *DE*s.

The reversing operator reverses a sub-sequence of a tour [27, 29]. As a two-opt move, this procedure deletes two edges and adds two new ones. The left-rotation operator rotates a subsequence of a tour one step to the left and the right-rotation operator rotates one step to the right [27, 30]. Both are possible three-opt moves, i.e., delete and insert (at most) three edges. Finally, the swap move simply exchanges two nodes [27, 31] and is a possible four-opt move, as it leads to the deletion and insertion of at most four edges.

B. Research Questions

We also conducted comprehensive experiments to test whether the experimental procedure discussed in Section II can provide meaningful answers to research problems. Some general questions are posed, such as whether global, local, or hybrid optimization algorithms can perform well on the TSP. There are questions with a narrower scope too, such as whether Frequency Fitness Assignment (*ffa*) [32] is a good diversity enhancement strategy in an MA for the TSP.

The best results have been achieved when global search methods are seeded and hybridized with local search.

1) Which EC Method Would Be Most Suited for the TSP?

In order to find some directions regarding which EC method is better for solving TSPs, we conducted experiments with several setups of three main branches of EC, namely an EA [33–35], an ACO [36], and an EDA [37]:

The *EA* uses the four aforementioned neighborhoods (see Figure 2) as mutation operators, and for each offspring it randomly chooses one to apply. It has a crossover rate of 1/3, and was tested with either the well-known Edge Crossover [38] or a new *Savings Crossover* operator. Savings Crossover constructs a new tour with the Savings heuristic [20], but only uses edges present in either of the parents. If cycles would occur, it reverts back to the original heuristic.

Population-based ACO (*PACO*) [39] is a version of the ACO algorithm that maintains a set (population) of k solutions. The edges present in those solutions define the pheromones. In each iteration, m solutions are generated as in standard ACO and the best of them replaces the oldest one in the population.

The Edge-Histogram based Sampling Algorithm [40, 41] is an EDA, which is here applied in the template-based version (*TEHBSA*) [41]. It uses a candidate set containing the 20 nearest nodes to any other node. This allows the reduction of the edge histogram matrix size to $20n$, but in this case the sampling process may arrive at a point where following the model would lead to a cycle in the tour. The tour is then either augmented with the closest unvisited node or a random node. We have also tested a *TEHBSA* version that maintains a complete edge histogram.

2) Does Seeding Improve the Results of Metaheuristic TSP Solvers?

Usually, the *EA*, *PACO*, and *TEHBSA* begin their search with random solutions, empty populations, and uniformly initialized models, respectively. However, the algorithms can instead be *seeded*. They can receive their initial population of solutions from heuristics for the TSP (discussed, e.g., in [20]).

We tested seeded versions (*hEA*, *hPACO*, *hTEHBSA*) of all three algorithms in order to confirm whether this approach is beneficial. The first two individuals in a seeded population are generated with the Edge-Greedy and Double Minimum Spanning Tree heuristic. The remaining slots are filled alternatingly with individuals resulting from the Savings, Double-Ended Nearest Neighbor, and Nearest Neighbor Heuristic, which are started at different, randomly chosen initial nodes.

3) Can Pure EC Approaches Outperform Local Search Methods on the TSP?

Currently, some of the most efficient approaches for solving the TSP are local search methods [42]. We therefore also benchmarked four local search algorithms in order to verify

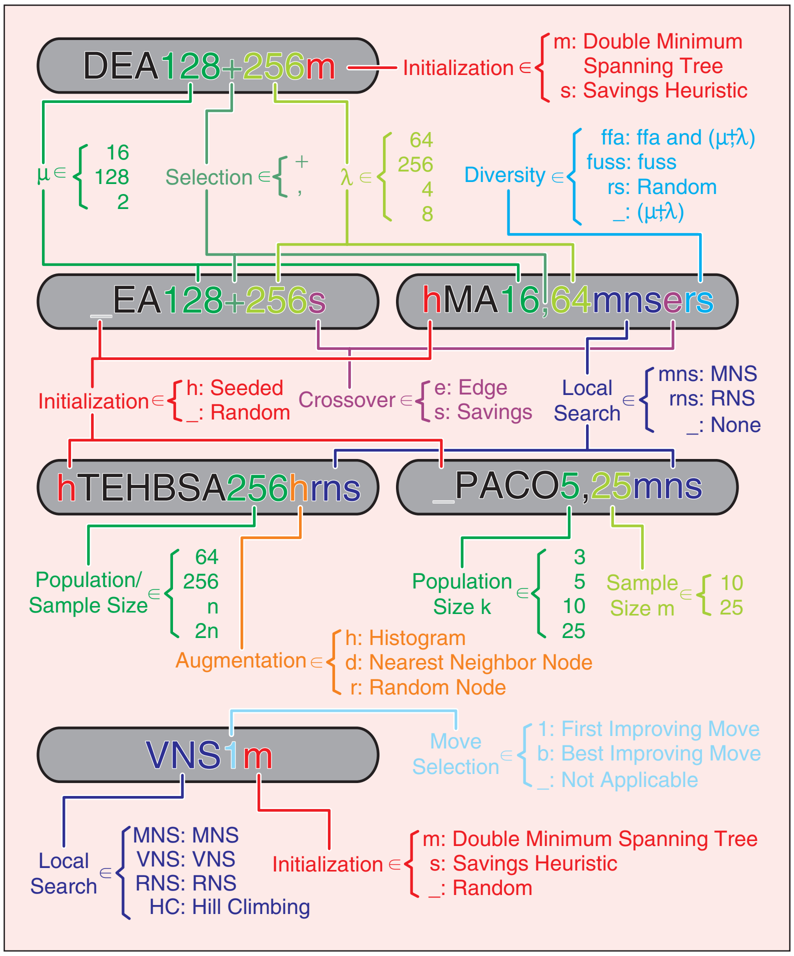
DEA128+256m
Initialization ∈ m: Double Minimum Spanning Tree; s: Savings Heuristic
μ ∈ 16, 128, 2
Selection ∈ +, ,
λ ∈ 64, 256, 4, 8
Diversity ∈ ffa: ffa and (μ†λ); fuss: fuss; rs: Random; _: (μ†λ)
_EA128+256s
hMA16,64mnsers
Initialization ∈ h: Seeded; _: Random
Crossover ∈ e: Edge; s: Savings
Local Search ∈ mns: MNS; rns: RNS; _: None
hTEHBSA256hrns
_PACO5,25mns
Population/ Sample Size ∈ 64, 256, n, 2n
Population Size k ∈ 3, 5, 10, 25
Sample Size m ∈ 10, 25
Augmentation ∈ h: Histogram; d: Nearest Neighbor Node; r: Random Node
VNS1m
Move Selection ∈ 1: First Improving Move; b: Best Improving Move; _: Not Applicable
Local Search ∈ MNS: MNS; VNS: VNS; RNS: RNS; HC: Hill Climbing
Initialization ∈ m: Double Minimum Spanning Tree; s: Savings Heuristic; _: Random

FIGURE 3 The notation for setups of the algorithms explored in our experiments (*DEA, EA, hMA, TEHBSA, PACO*, and local search) as introduced in Section IV and used in Figure 4. The gray boxes hold one example setup for each algorithm. The notation elements in the setup names are connected to the corresponding parameters and parameter values. "_" represents default values not explicitly signified in the setup name.

hPACO3,10mns (rank 1), *hPACO3,25mns* (2.5), *hPACO5,10mns* (2.5), *hPACO5,25mns* (4), *hPACO10,10mns* (5), *hPACO10,25mns* (6), *hMA16+64mnse* (7), *hMA2+8mnse* (8), *hMA2+4mnse* (9), *hMA16,64mnse* (10), *hMA2+8mnss* (11), *hMA2,8mnse* (12), *hMA2+4mnss* (13), *hMA128+256mnse* (14), *hMA16+64mnss* (15), *hMA16+64mnseffa* (16), *hTEHBSA64hmns* (17), *hMA16,64mnseffa* (18), *hTEHBSA64dmns* (19), *hMA16+64mnsers* (20), *hMA128,256mnse* (21), *hMA128+256mnseffa* (22), ***hPACO3,10rns*** (23), *hMA128+256mnss* (24.5), *MNSm* (24.5), ***hPACO3,25rns*** (26), *hMA2,4mnse* (27), *hMA128+256mnsefuss* (28), ***hPACO5,25rns*** (29), *hMA16,64mnsers* (30), *hMA128,256mnsefuss* (31), ***hPACO10,25rns*** (32), *hMA128,256mnseffa* (33), ***hPACO10,10rns*** (34), *hMA128+256mnsers* (35), *hTEHBSA256hmns* (36), *hMA16+64mnsefuss* (37), *hTEHBSA64rmns* (38), *hMA16,64mnss* (39), *hTEHBSAndmns* (40.5), *hTEHBSAnhmns* (40.5), *hMA16,64mnsefuss* (42), *hMA128+256mnssfuss* (43), *hMA128,256mnsers* (44), *MNSs* (45), ***hPACO5,10rns*** (46), *hTEHBSA2nhmns* (47), *hMA16+64mnssffa* (48.5), *hTEHBSA256dmns* (48.5), *hTEHBSA2ndmns* (50), *hTEHBSA256rmns* (51), *hTEHBSA2nrmns* (52.5), *hTEHBSAnrmns* (52.5), *hMA128+256mnssffa* (54), *hMA128,256mnss* (55), *MNS* (56), *hMA2,8mnss* (57), *hMA128,256mnssfuss* (58), *hMA16,64mnssffa* (59), *hMA128,256mnssffa* (60), *hMA128,256mnssrs* (61), *hMA128+256mnssrs* (62.5), *hMA2,4mnss* (62.5), *hMA16,64mnssfuss* (64), *hMA16+64mnssfuss* (65), ***hMA2+8rnss*** (66), ***hMA2+4rnse*** (67), ***hMA16+64rnss*** (68.5), ***hMA2+4rnss*** (68.5), ***hMA2+8rnse*** (70), *hMA16,64mnssrs* (71), *hMA16+64mnssrs* (72), ***hTEHBSA64hrns*** (73), ***hTEHBSAndrns*** (74), ***hTEHBSA64rrns*** (75), ***hTEHBSA64drns*** (76), ***hMA16+64rnse*** (77.5), ***hTEHBSA256hrns*** (77.5), ***hTEHBSAnhrns*** (79), ***hMA128+256rnss*** (80), ***hTEHBSA2nhrns*** (81), ***hTEHBSA256drns*** (82), ***hMA16,64rnss*** (83), *RNSbm* (84), ***hTEHBSA2ndrns*** (85), ***hTEHBSA2nrrns*** (86), ***hMA128+256rnssfuss*** (87.5), ***hTEHBSA256rrns*** (87.5), ***hTEHBSAnrrns*** (89), *RNS1s* (90), *RNSbs* (91), ***hMA2,8rnss*** (92), *hMA128,256rnssrs* (93), *RNS1* (94), ***hMA16,64rnssffa*** (95), *VNSbm* (96), ***hMA128,256rnss*** (97), ***hMA128+256rnssffa*** (98), *hMA128+256rnssrs* (99), ***hMA16+64rnssffa*** (100), *hMA16+64rnssrs* (101), ***hMA128,256rnssfuss*** (102), *RNS1m* (103), ***hMA16,64rnssfuss*** (104), ***hMA128,256rnssffa*** (105), *VNSbs* (106), *RNSb* (107), *hMA16,64rnssrs* (108), ***hMA2,8rnse*** (109), ***hMA128+256rnse*** (110.5), ***hMA16+64rnssfuss*** (110.5), ***hMA128+256rnseffa*** (112), ***hMA16+64rnseffa*** (113), ***hMA128,256rnsers*** (114), ***hMA128,256rnseffa*** (115), *hMA128,256rnsefuss* (116), *hMA128+256rnsefuss* (117.5), ***hMA128+256rnsers*** (117.5), *hMA16,64rnsefuss* (119.5), *VNS1s* (119.5), ***hMA128,256rnse*** (121), ***hMA16+64rnsers*** (122.5), ***hMA16,64rnseffa*** (122.5), ***hMA16,64rnse*** (124), ***hMA2,4rnss*** (125), ***hMA2,4rnse*** (126), *hMA16+64rnsefuss* (127.5), ***hMA16,64rnsers*** (127.5), *VNS1m* (129), *VNS1* (130), *VNSb* (131), *hEA128+256e* (132), *hEA16+64e* (133), *hEA128+256s* (134.5), *hEA16,64e* (134.5), *hEA16,64s* (136), *hEA16+64s* (137), *hPACO3,25* (138), *hPACO5,25* (139), *hPACO10,25* (140), *hPACO3,10* (141), *hPACO5,10* (142.5), *hTEHBSA256h* (142.5), *hPACO10,10* (144.5), *PACO3,25* (144.5), *hTEHBSA256d* (146), *hTEHBSA2nd* (147), *hTEHBSA2nh* (148.5), *hTEHBSA2nr* (148.5), *HCs* (150), *hTEHBSA256r* (151), *PACO5,25* (152), *hTEHBSAnh* (153), *PACO3,10* (154), *hTEHBSA64r* (155), *hEA128,256e* (156), *hTEHBSA64h* (157.5), *PACO10,25* (157.5), *hTEHBSA64d* (159.5), *hTEHBSAnr* (159.5), *hTEHBSAnd* (161), *DEA16,64s* (162), *PACO5,10* (163), *DEA16+64s* (164.5), *hEA128,256s* (164.5), *HC* (166), *PACO10,10* (167), *DEA128+256m* (168.5), *DEA128,256s* (168.5), *DEA128+256s* (170), *DEA128,256m* (171), *EA128+256e* (172), *DEA16,64m* (173), *EA16+64e* (174), *DEA16+64m* (175), *EA16,64e* (176), *EA16+64s* (177), *EA16,64s* (178), *EA128+256s* (179), *TEHBSA64d* (180), *TEHBSAnd* (181), *TEHBSA256d* (182), *TEHBSA2nd* (183), *TEHBSA2nr* (184), *EA128,256s* (185), *TEHBSA2nh* (187), *TEHBSAnh* (187), *TEHBSAnr* (187), *TEHBSA64h* (189.5), *TEHBSA64r* (189.5), *EA128,256e* (191), *TEHBSA256h* (192), and *TEHBSA256r* (193)

FIGURE 4 Rankings of the 193 algorithm configurations according to the metrics defined in Section III-C, specified in the notation given in Figure 3. The algorithm families are represented by different colors as follows: *local search*, *EA*, *hMA*, *PACO*, *DEA*, and *TEHBSA*. Names of algorithms that are hybridized with *MNS* are underlined and those that employ *RNS* are written in **bold face**. The best-ranked version of each algorithm family is displayed in a frame. The best pure global search is displayed in a red frame and the best non-hybrid, seeded global search has a blue frame.

whether EC methods can outperform them (on the *TSPLib* instances):

a) A simple Hill Climber (*HC*) that applies the aforementioned four search operators in a loop.

b) The Variable Neighborhood Search (*VNS*) [43] implemented in the *TSP Suite* is based on the same four neighborhoods. In its main loop, it first shuffles them randomly and then performs a neighborhood descend. Once it reaches an optimum it cannot escape from anymore, a random part of the solution is shuffled and the procedure begins again with shuffling the neighborhoods.

c) Instead of descending the neighborhoods in a specific order and always returning to the first neighborhood when an improvement is made, we also tested an algorithm that tries to pick a random, different neighborhood for each move it makes. We call this method "Random Neighborhood Search" (*RNS*). Both *VNS* and *RNS* were tested in versions that either take the best (*VNSb*, *RNSb*) or first discovered (*VNS1*, *RNS1*) improving move of a neighborhood.

d) Inspired by [44], we developed a new local search method called Multi-Neighborhood Search (*MNS*). The *MNS* algorithm performs a $O(n^2)$ scan of the solution and collects all improving moves under the four defined neighborhoods in a queue. The best move is extracted from the queue and immediately executed. This may invalidate some moves in the queue, e.g., if it is a `rev` that directly intersects with `rol` (see Figure 2). These moves are pruned from the queue. Non-intersecting moves are not affected and some moves may be modified but are still applicable: A `rol` fully included inside a reversed sub-sequence, for instance, simply becomes a corresponding `ror` move. To walk through a queue of length l and find the best remaining move while pruning invalidated ones can be done in $O(l)$ steps. If the queue is empty, it is filled again. Otherwise, the best remaining move is applied. If no further moves can be discovered, a random fraction of the tour is randomly shuffled, exactly in the same way as in *VNS* or *RNS*.

We started these four algorithms either at a random solution, with one generated by the Savings, or the Double Minimum Spanning Tree Heuristic.

4) Can Local Search Methods Benefit from Being Hybridized with EC Methods?

MAs [45], which combine EAs with local search, are amongst the best optimization algorithms for combinatorial problems. We created hybrid versions of the three EC approaches from Section IV-B2 to verify whether these can outperform the local search methods.

Proper experimentation should avoid reducing algorithm performance to singular points.

We refer to our hybrid (seeded) *hEA* as the *hMA*. Here, the crossover rate has been set to 1 and each solution that is generated will be refined with a local search method. For this purpose, we have slightly modified versions of *RNS* (which randomly chooses between the best- and first-improvement selection policy per call) and *MNS*. These versions will stop when converging to an optimum that is *different* than the input solution. We also tested versions of *hPACO* and *hTEHBSA* that have undergone the same hybridization. The tested setups of these hybrid global search algorithms have a name suffix indicating the local search used for hybridization (see Figure 3) and are always seeded (prefix *h*).

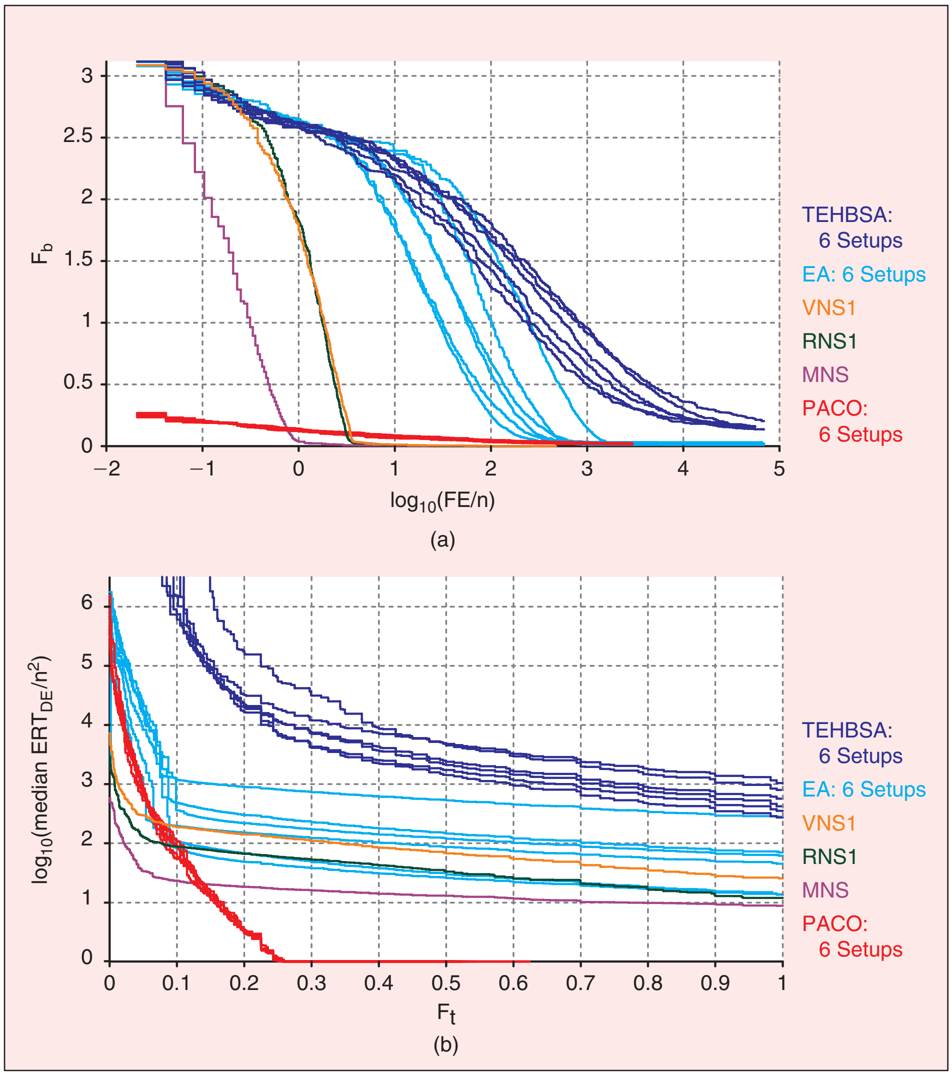

FIGURE 5 Examples of (manually post-processed) progress and *ERT* diagrams for the six best non-seeded and non-hybridized setups of each algorithm family. (a) A progress diagram showing the median smallest error (F_b) achieved after a given amount of *FE*s has been consumed (log-scaled) on the benchmark instance *gr48*. (b) The median estimated running time ERT_{DE} in terms of *DE*s to reach a relative error threshold F_t over the 8 *TSPLib* instances with $32 \leq n < 64$.

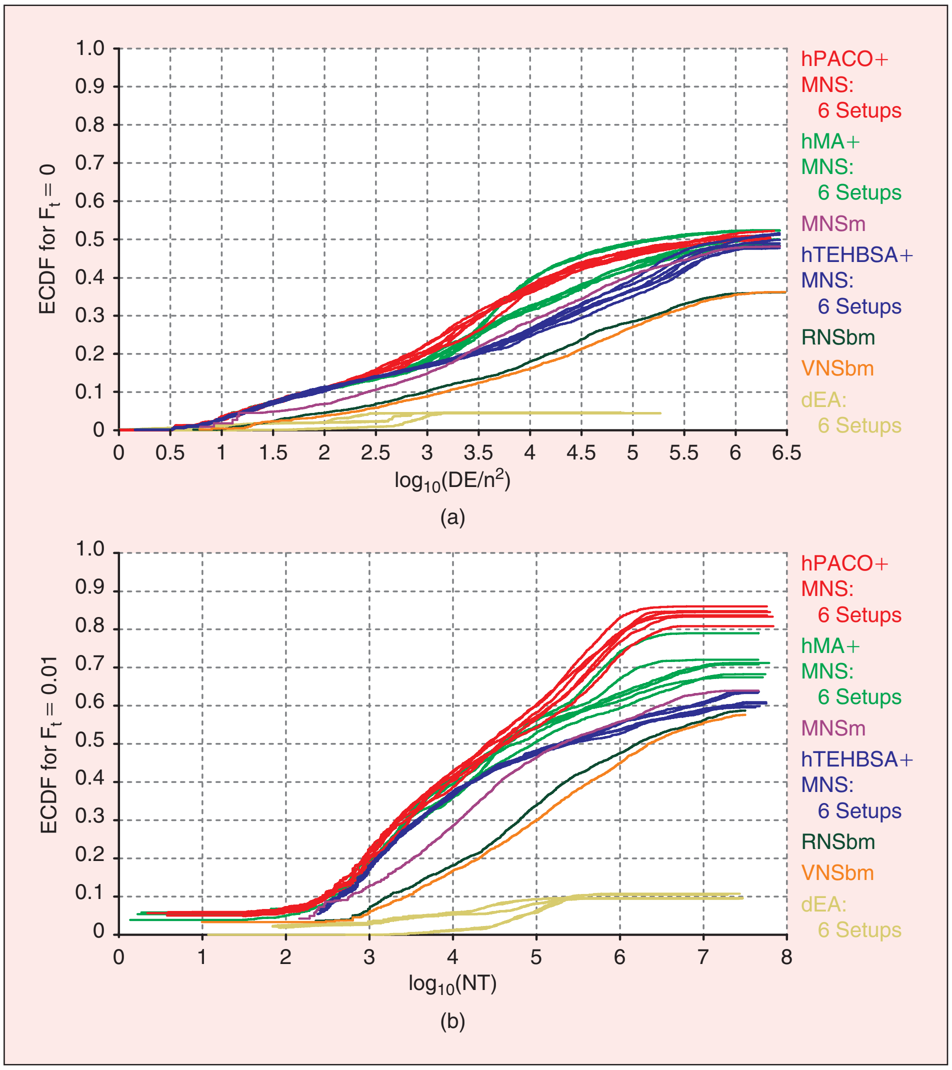

FIGURE 6 Examples of (manually post-processed) *ECDF* diagrams containing the six best seeded variants of all algorithm families. These happened to always be hybridized with *MNS* (except for *DEA*, for which no hybrid setups were tested, and the local search algorithms themselves). (a) The *ECDF* over *DEs* (scaled by n^2 on a log-scaled axis) for goal $F_t = 0$, i.e., the fraction of runs that have discovered the globally optimal solution after a given number of *DEs*. (b) The *ECDF* over *NT* (log-scaled) for goal $F_t = 0.01$, i.e., the fraction of runs that have discovered a solution not more than 1 percent longer than the globally optimal solution after a given amount of normalized runtime.

5) Further Questions

Methods for preventing convergence have recently received much attention in the EC community. In order to verify whether such approaches can be beneficial in solving TSPs, we applied three of them in the *hMA*: Fitness Uniform Selection (*fuss*) [46], *ffa*, and random selection (*rs*). Furthermore, we also investigated the performance of the Genetic Programming based developmental approach (Developmental EA, *DEA* in short) for the TSP introduced in [47].

C. Evaluation Results

Each of the algorithms described in Section IV-B was implemented in the *TSP Suite*. We investigated 193 different setups of these algorithms based on the parameter values given in Figure 3. The experiments resulted in about 20GB of log files, which will be made available online. Due to space constraints, we will limit ourselves to the main conclusions and representative examples of the generated diagrams obtained with the evaluator component. The global ranking of the setups given in Figure 4 directly provides answers to the research questions from Section IV-B.

In their non-hybridized and non-seeded variants, none of the tested EC approaches (*EA*, *PACO*, and *TEHBSA*) performed well. In Figure 5, where examples of the *ERT* and progress diagrams for the six best such setups of each algorithm family are plotted, it can be seen that the benchmarked local search algorithms are faster in obtaining solution qualities below $F_t = 0.05$.

The ranking in Figure 4 shows that the seeded global optimization algorithm variants (*hEA*, *hTEHBSA*, *hPACO*) have performed better than the non-seeded ones. They started at good solution qualities, but are still outperformed by the investigated local search algorithms.

The best results have been achieved when global search methods are seeded and hybridized with local search. Figure 6 contains some examples of *ECDF* plots for the six best-ranked hybridized and seeded variants of the global search algorithms.

Figure 6a shows that several *hMA* setups can find globally optimal solutions in more than 50% of all runs over all the benchmark problems. In this respect, the *hMA* is slightly better than the hybrid *hPACO*, but *hPACO* with local search can find solutions with $F_b \leq 1\%$ more often and is faster (Figure 6b).

PACO has performed the best among the tested EC methods and also provided the overall best results when seeded and hybridized with local search. This trend is very clear, especially if we consider that only 24 *PACO*-based configurations had been tested versus 80 *hMA*-based and 48 *TEHBSA*-based setups.

MNS has performed better than the other local search algorithms, both in pure and hybrid forms, likely because one

$O(n^2)$-scan of the solution can result in multiple improvements. With ranks 24.5 (seeded) and 56 (not seeded), it outperformed all pure and seeded EC methods as well as most of their hybrids.

The TSP Suite, including its source code, extensive documentation, example data, and example reports, can be downloaded from http://www.logisticPlanning.org/tsp/.

Although *ffa* appears to be a slightly better convergence prevention strategy for the *hMA* than *fuss* or *rs*, the *hMA* using neither of them can perform even better. While the always seeded *DEA* was able to outperform the non-seeded *EA* and *TEHBSA* variants, it was inferior to a simple seeded *HC*. Edge Crossover is better than the new Savings Crossover.

To sum up, the best local search (*MNS*) together with the best pure global search (*PACO*) produces the best hybrid search. Pure *PACO* performs better than the pure *EA*. Their hybridized versions with *MNS* show the same behavior. One may thus assume that hybridization would have an almost additive effect. However, this is not always true, since all tested hybrid algorithms are seeded and the (seeded) *hEA* is better than the (seeded) *hPACO*.

Based on the results presented, we hope that our experimental approach can lead to the development of better algorithms through identifying behaviors and trends of different algorithms. Prior to these experiments using the *TSP Suite*, some of us would expect an *MA* to be the best hybrid EC method for the TSP. The results, however, tell us that *PACO* is the method of choice.

V. Conclusions and Future Work

This paper has made four contributions to the research on combinatorial optimization with metaheuristics in general and the TSP in particular.

First, we proposed an experimentation procedure that allows for analyzing and comparing optimization algorithms from several different points of view. This procedure marks a step forward between the traditional experimental analysis and data mining applied to performance data from optimization processes. It is not limited to the TSP and may be useful in other domains as well.

Second, this experimentation procedure is realized in a general open source framework for the implementation, unit testing, experimentation, and analysis of TSP algorithms. The *TSP Suite*, including its source code, extensive documentation, example data, and example reports, can be downloaded from http://www.logisticPlanning.org/tsp/. With the *TSP Suite*, experiments can be run in a parallel or distributed fashion and evaluation reports containing high-level, human-readable conclusions can be produced.

Third, the *TSP Suite* has been used to obtain a baseline set of data generated from several local search methods as well as members of the main EC algorithm families (EAs, MAs, ACO, EDAs) in pure, seeded, and hybridized versions. This allows users of the *TSP Suite* to acquire data from algorithms that are related and suitable for comparison purposes. All of these implementations and collected data will be made available.

Fourth, with the experiments we have shown that the *TSP Suite* is an efficient tool for answering both general and specific research questions. This also validates the experimental procedure (which is not limited to TSPs). We confirmed that local search can outperform pure global search methods on the TSP, but that EC methods hybridized with local search can be even better. *PACO* and the new *MNS* have been found to be the best global and local optimization algorithms in the tests, respectively.

There are five major strands of future work, which will be followed by the authors.

First, we are working to set up a centralized website that provides the *TSP Suite* and its documentation as well as all the generated benchmarking results for download. This site will allow other researchers to upload their results and maintain an up-to-date list of the best TSP solvers.

Second, the collection of algorithms in our *TSP Suite* is far from complete. We are currently implementing Branch and Bound methods and Lin-Kernighan local search [48]. Comprehensive experiments with these two methods will be performed.

Third, the experimentation approach described in Section II-B will be extended to other well-known optimization tasks such as Knapsack or Set Covering problems. We plan to repeat our initial analysis for viable time measures (Section II-A3) and then re-use existing code from the *TSP Suite*.

Fourth, at present the *TSP Suite* can compare different *experiments*, but it cannot automatically analyze what influences their *parameters* may have. For example, in Section IV-C, we had to *manually* conclude that Edge Crossover is better than our new Savings Crossover. It would be more convenient if the *TSP Suite* could *automatically* derive such conclusions (instead of treating each algorithm setup as a different algorithm).

Fifth, there are several limitations with the current version of the experimental procedure that need to be addressed, some of which are 1) The global ranking of algorithms depends strongly on the mixture of statistics it is based on, so this mixture needs to be further discussed and analyzed; 2) The comparison of the areas under performance curves is our first formal idea to compare dynamic behaviors, but probably not statistically robust; 3) The best discovered solution must be preserved by the *TSP Suite* in the log files, which entails performing an $O(n)$ copy operation whenever the running algorithm registers an improvement. This may potentially skew the measured CPU times. Since a running algorithm can query the system for the best solution it has found so far, it is relieved from making this copy itself, which *may* offset this expense; 4) Additionally, we are looking for better methods to mine the data gathered during the experimental runs.

Acknowledgment

The authors at UBRI would like to acknowledge support from the National Natural Science Foundation of China under Grants 61175065, 61329302, and 61150110488, the Technological Fund of Anhui Province for Outstanding Youth under Grant 1108085J16, the Special Financial Grant 201104329 from the China Postdoctoral Science Foundation, the Chinese Academy of Sciences (CAS) Fellowship for Young International Scientists 2011Y1GB01, and the European Union 7th Framework Program under Grant 247619. The second author would like to acknowledge support from the University of Newcastle Faculty of Science and Information Technology's Strategic Initiatives Research Fund (Grant Code 10.31415). The experiments reported in this paper were executed on the supercomputing system in the Supercomputing Center of University of Science and Technology of China.

References

[1] C. Blum, R. Chiong, M. Clerc, K. A. De Jong, Z. Michalewicz, F. Neri, and T. Weise, "Evolutionary optimization," in *Variants of Evolutionary Algorithms for Real-World Applications*, R. Chiong, T. Weise, and Z. Michalewicz, Eds. Berlin, Germany: Springer-Verlag, 2011, ch. 1, pp. 1–29.

[2] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton, NJ: Princeton Univ. Press, 2007.

[3] E. L. G. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. New York: Wiley Interscience, 1985.

[4] G. Z. Gutin and A. P. Punnen, Eds., *The Traveling Salesman Problem and its Variations* (Combinatorial Optimization, vol. 12). Norwell, MA: Kluwer Academic Publishers, 2002.

[5] W. J. Cook. (2013). World TSP. [Online]. Available: http://www.math.uwaterloo.ca/tsp/world/

[6] N. Hansen, A. Auger, S. Finck, and R. Ros, "Real-parameter blackbox optimization benchmarking: Experimental setup," INRIA Futurs, Équipe TAO, Univ. Paris Sud, Orsay, France, Tech. Rep. RR-6828, Mar. 24, 2012.

[7] D. A. D. Tompkins and H. H. Hoos, "UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAXSAT," in *Proc. Revised Selected Papers 7th Int. Conf. Theory Applications Satisfiability Testing*, Vancouver, Canada, May 5–13, 2004, vol. 3542, pp. 306–320.

[8] H. H. Hoos and T. Stützle, *Stochastic Local Search: Foundations and Applications*. San Francisco, CA: Morgan Kaufmann, 2005.

[9] D. S. Johnson and L. A. McGeoch. (2008). 8th DIMACS implementation challenge: The traveling salesman problem. [Online]. Available: http://dimacs.rutgers.edu/Challenges/TSP/

[10] K. Tang, Z. Yang, and T. Weise, "Special session on evolutionary computation for large scale global optimization at 2012 IEEE world congress on computational intelligence (CEC@WCCI-2012)," Nature Inspired Comput. Applicat. Lab., Univ. Sci. Technol. China, Hefei, China, Tech. Rep., June 14, 2012.

[11] T. Bartz-Beielstein, "SPOT: An R package for automatic and interactive tuning of optimization algorithms by sequential parameter optimization," Cologne Univ. Appl. Sci., Gummersbach, Germany, Tech. Rep. CIOP TR 05-10, 2010.

[12] T. Chen, K. Tang, G. Chen, and X. Yao, "Analysis of computational time of simple estimation of distribution algorithms," *IEEE Trans. Evol. Comput.*, vol. 14, no. 1, pp. 1–22, 2010.

[13] J. Lässig and D. Sudholt, "Experimental supplements to the theoretical analysis of migration in the island model," in *Parallel Problem Solving from Nature XI* (Lecture Notes in Computer Science, vol. 6238). Kraków, Poland: Springer, Sept. 11–15, 2010, pp. 224–233.

[14] J. Lässig and K. H. Hoffmann, "Threshold-selecting strategy for best possible ground state detection with genetic algorithms," *Phys. Rev. E*, vol. 79, no. 4, p. 046702, 2009.

[15] G. Reinelt, "TSPLIB—A traveling salesman problem library," *ORSA J. Comput.*, vol. 3, no. 4, pp. 376–384, 1991.

[16] D. S. Johnson and L. A. McGeoch, "Experimental analysis of heuristics for the STSP," in *The Traveling Salesman Problem and its Variations [4]*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2002, ch. 9, pp. 369–443.

[17] D. S. Johnson, G. Z. Gutin, L. A. McGeoch, A. Yeo, W. Zhang, and A. Zverovitch, "Experimental analysis of heuristics for the ATSP," in *The Traveling Salesman Problem and its Variations [4]*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2002, ch. 10, pp. 445–487.

[18] W. J. Cook. (2011). Traveling salesman problem. [Online]. Available: http://www.tsp.gatech.edu/

[19] M. S. Boddy and T. L. Dean, "Solving time-dependent planning problems," Dept. Comput. Sci., Brown Univ., Rhode Island, Providence, Tech. Rep. CS-89-03, Feb. 1989.

[20] D. S. Johnson and L. A. McGeoch, "The traveling salesman problem: A case study in local optimization," in *Local Search in Combinatorial Optimization*. Princeton, NJ: Princeton Univ. Press, 1995, pp. 215–310.

[21] O. J. Dunn, "Multiple comparisons among means," *J. Amer. Stat. Assoc.*, vol. 56, no. 293, pp. 52–64, 1961.

[22] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, Jan. 2006.

[23] S. García and F. Herrera, "An extension on 'Statistical comparisons of classifiers over multiple data sets' for all pairwise comparisons," *J. Mach. Learn. Res.*, vol. 9, pp. 2677–2694, Dec. 2008.

[24] H. H. Hoos and T. Stützle, "Evaluating Las Vegas algorithms—Pitfalls and remedies," in *Proc. 14th Conf. Uncertainty Artificial Intelligence*, July 24–26, 1998, pp. 238–245.

[25] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognit. Lett.*, vol. 27, no. 8, pp. 861–874, 2006.

[26] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognit.*, vol. 30, no. 7, pp. 1145–1159, 1997.

[27] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic, "Genetic algorithms for the travelling salesman problem: A review of representations and operators," *J. Artif. Intell. Res.*, vol. 13, no. 2, pp. 129–170, 1999.

[28] K. Beck, *JUnit Pocket Guide*. Sebastopol, CA: O'Reilly, 2009.

[29] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: Univ. Michigan Press, 1975.

[30] D. B. Fogel, "An evolutionary approach to the traveling salesman problem," *Biol. Cybern.*, vol. 60, no. 2, pp. 139–144, 1988.

[31] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. London, U. K.: Springer-Verlag, 1996.

[32] T. Weise, M. Wan, K. Tang, P. Wang, A. Devert, and X. Yao, "Frequency fitness assignment," *IEEE Trans. Evol. Comput.*, vol. 18, no. 2, pp. 226–243, Apr. 2014.

[33] K. A. de Jong, *Evolutionary Computation: A Unified Approach*. Cambridge, MA: MIT Press, 2006, vol. 4.

[34] T. Bäck, D. B. Fogel, and Z. Michalewicz, Eds., *Handbook of Evolutionary Computation*. London, U.K.: Oxford Univ. Press, 1997.

[35] T. Weise. (2009). Global optimization algorithms—Theory and application. [Online]. Available: http://www.it-weise.de/projects/book.pdf

[36] M. Dorigo, M. Birattari, and T. Stützle, "Ant colony optimization—Artificial ants as a computational intelligence technique," *IEEE Comput. Intell. Mag.*, vol. 1, no. 4, pp. 28–39, 2006.

[37] P. Larrañaga and J. A. Lozano, Eds., *Estimation of Distribution Algorithms—A New Tool for Evolutionary Computation* (Genetic Algorithms and Evolutionary Computation). Berlin Heidelberg, Germany: Springer-Verlag, 2001, vol. 2.

[38] L. D. Whitley, T. Starkweather, and D. Fuquay, "Scheduling problems and traveling salesman: The genetic edge recombination operator," in *Proc. 3rd Int. Conf. Genetic Algorithms*, Fairfax, VA, June 4–7, 1989, pp. 133–140.

[39] M. Guntsch and M. Middendorf, "Applying population based ACO to dynamic optimization problems," in *Ant Colonies to Artificial Ants—Proc. 3rd Int. Workshop Ant Colony Optimization*, Brussels, Belgium, Sept. 12–14, 2002, vol. 2463, pp. 111–122.

[40] S. Tsutsui, "Probabilistic model-building genetic algorithms in permutation representation domain using edge histogram," in *Proc. 7th Int. Conf. Parallel Problem Solving Nature (PPSN VII)*, Granada, Spain, Sept. 7–11, 2002, vol. 2439, pp. 224–233.

[41] S. Tsutsui, "Parallelization of an evolutionary algorithm on a platform with multi-core processors," in *Proc. Artificial Evolution: Revised Selected Papers 9th Int. Conf. Evolution Artificielle*, Strasbourg, France, Oct. 26–28, 2009, vol. 5975, pp. 61–73.

[42] K. Helsgaun, "General k-opt submoves for the Lin–Kernighan TSP heuristic," *Math. Program. Comput.*, vol. 1, nos. 2–3, pp. 119–163, 2009.

[43] P. Hansen, N. Mladenović, and J. A. M. Pérez, "Variable neighbourhood search: Methods and applications," *Ann. Oper. Res.*, vol. 175, no. 1, pp. 367–407, 2010.

[44] L. D. Whitley and W. Chen, "Constant time steepest descent local search with lookahead for NK-landscapes and MAX-kSAT," in *Proc. Genetic Evolutionary Computation Conf. ACM*, Philadelphia, PA, July 7–11, 2012, pp. 1357–1364.

[45] Y. Ong, M. H. Lim, and X. Chen, "Memetic computation—Past, present & future [Research Frontier]," *IEEE Comput. Intell. Mag.*, vol. 5, no. 2, pp. 24–31, 2010.

[46] M. Hutter and S. Legg, "Fitness uniform optimization," *IEEE Trans. Evol. Comput.*, vol. 10, no. 5, pp. 568–589, 2006.

[47] J. Ouyang, T. Weise, A. Devert, and R. Chiong, "SDGP: A developmental approach for traveling salesman problems," in *Proc. IEEE Symp. Computational Intelligence Production Logistics Systems*, IEEE Computer Society Press, Singapore, Apr. 15–19, 2013, pp. 78–85.

[48] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Oper. Res.*, vol. 21, no. 2, pp. 498–516, 1973.