9

Working with Evolutionary Algorithms

In this chapter we discuss the practical aspects of using EAs. Working with EAs often means comparing different versions experimentally, and we provide guidelines for doing this, including the issues of algorithm performance measures, statistics, and benchmark test suites. The example application (Sect. 9.4) is also adjusted to the special topics here; it illustrates the application of different experimental practices, rather than EA design.

9.1 What Do You Want an EA to Do?

Throughout this book so far, we have seemingly never considered this issue: "What do you want an EA to do?". The reason is that we tacitly assumed the trivial answer: "I want the EA solve my problem". Many of the subjects treated in this chapter concern specific interpretations and refinements of this answer, and it will become clear that different objectives imply different ways of designing and working with an EA.

A good first step is to examine the given problem context. We can roughly distinguish two main types of problems:

- design (one-off) problems
- repetitive problems, including on-line control problems as special cases

As an example of a design problem, let us consider the optimisation of extensions to an existing road network to meet new demands. This is most certainly a highly complex multiobjective optimisation problem, subject to many constraints. Computer support for this problem requires an algorithm that creates *one* excellent solution at least *once*. In this context the quality of the solution is of utmost importance, and other aspects of algorithm performance are secondary. For instance, since the time scale of the whole project spans years, the algorithm does not have to be fast. It can be given months of computing time, perhaps performing several runs and keeping the best result, if this helps in achieving superior solution quality. The algorithm does

© Springer-Verlag Berlin Heidelberg 2015

A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*,
Natural Computing Series, DOI 10.1007/978-3-662-44874-8_9

not need to be generally applicable either. The present problem most probably contains very specific aspects, hindering reapplication of the algorithm to other problems. Furthermore, a similar problem will occur as part of a similar project allowing enough time to develop a good EA for that problem.

Repetitive problems form a counterpoint to design problems. As an illustration, consider a domestic transportation firm, having dozens of trucks and drivers that need to be given a daily schedule every morning. The schedule should contain a pick-up and delivery plan, plus the corresponding route description for each truck and driver. For each of them, this is just a TSP problem (probably with time windows), but the optimisation criteria and the constraints must be taken across the whole firm, together making the actual problem very complex. Depending on the type of business, the data and requirements for a day's schedule might become available weeks, days, but maybe only hours before the schedules are to be handed out to the drivers. In any case, the dispatcher must provide a schedule every morning to every available driver. Suitable computer support for this problem, an EA in our case, must be able to find *good* solutions *quickly* and be able to do this *repeatedly* for *different instances* of the problem (i.e., with different data and requirements every day). The implications for the algorithm are radically different from those in the case of a design problem. The balance in the speed versus quality trade-off is clearly towards speed. Solutions must be good, e.g., better than hand-made ones, but not necessarily optimal. Speed, however, is crucial. For example, it could be required that the time between feeding the data into the system and receiving the schedules does not exceed 30 minutes. Closely related to this issue, it is important that the algorithm performance is stable. Since an EA is a stochastic algorithm, the quality of end solutions over a number of runs shows a certain variance. For a design problem we typically have the time to perform many runs and select the best solution. Therefore it is not a problem if some runs terminate with bad results, as long as others end with good solutions. For repetitive problems, however, we might only have time for one run. To reduce the probability of really bad runs, we need a consistent EA to keep the variance of end solution quality as low as possible. Finally, for repetitive problems the widescale applicability of the algorithm is also important as the system will be used under various circumstances. In other words, the algorithm will be run on different problem instances.

On-line control problems can also be seen as repetitive problems with extremely tight time constraints. To remain in the transportation context, we can think of traffic light optimisation. In particular, let us consider the task of optimising a controller to set the green times of a single crossing with four crossroads. We assume that each of the crossroads has sensors embedded in the road surface that continuously monitor traffic approaching the crossing.[1] This sensory information is sent to the traffic light controller, a piece of software running on a special device at the crossing. The task of this controller

[1] This is common in many countries, and standard in the Netherlands.

is to calculate the green times for each of the roads in such a way that the throughput of vehicles is maximised. It is important to note that an EA can be used for this problem in two completely different ways. First, we can use an EA *to develop a controller*, based on simulations, which is then deployed at the crossing in question. This type of application was mentioned in Sect. 6.4 on genetic programming. The other way is to have an EA that *is the controller*, and this is what we have in mind here. The most important requirement here is, of course, speed. A controller is working in on-line mode, and it has to cope with streaming sensory information and needs to control the traffic lights in real time. The speed requirements are given in wall-clock time: the length of one full cycle[2] is typically a few minutes, and this time must be enough to calculate the green times for the following cycle. This can be very demanding for an EA that works with a whole population of candidate solutions and needs quite a few generations to evolve a good result. Fortunately, by the nature of traffic flows, the situation does not change very rapidly, which also holds for many other control problems. This means that the consecutive problem instances are rather similar to each other, and therefore it can be expected that the corresponding near-optimal solutions are similar as well. This motivates an EA that keeps the best solutions from previous runs and uses them in the new run. The second requirement, similar to repetitive problems, is a small variance in end solution quality. The third one is that the controller (the EA) must be very fault-tolerant and robust. This means that noise in the data (measurement errors of the sensors) or missing data (breakdown of a sensor) must not have a critical effect. The system, and the EA, must keep working and delivering the best possible results under the given circumstances.

Finally, let us mention a different but important context for working with EAs: academic research. The considerations above apply in an application-oriented situation, and it can be argued that making good applications is one of the major goals of the whole evolutionary computing field. However, an examination of the EC literature soon reveals that a huge majority of papers in scientific journals, conference proceedings, or monographs are ignorant of such concrete application-related issues. Scientific research apparently has its own dynamics, goals, methodologies, and conventions. Some of these arise from the fact that EAs can exhibit complex behaviours and emergent phenomena that are interesting per se, and developing a solid theoretical understanding may yield insight into real biological evolution. This chapter would not be complete without paying attention to working with EAs in an academic environment.

The objective in many experimental research papers, implicitly or explicitly, is to show that some EA is better than other EAs or their competitors – at least for some 'interesting' problems. This objective is typically not placed into an application context. The requirements of the algorithm are therefore

[2] One cycle is defined as the time between two consecutive turn-to-green moments of traffic light no. 1.

not inferred from what we want it to do; rather, they are based on conventions or ad hoc choices. Typical goals for academic experimentation are:

- Get a good solution for a given problem, e.g., challenging combinatorial optimisation.
- Show that EC is applicable in a (possibly new) problem domain.
- Show that an EA with some newly invented feature is better than some benchmark EA.
- Show that EAs outperform traditional methods on some relevant problems.
- Find best setup for parameters of a given EA, in particular, get data on the impact of varying some EA component, e.g., the population size.
- Obtain insights into algorithm behaviour, e.g., the interaction between selection and variation.
- See how an EA scales-up with problem size.
- See how the performance is influenced by parameters of the problem *and* the algorithm.

While these goals are different among themselves, and academic experimental research is apparently different from application-oriented work, there are general issues for all of these cases. The most prominent issue present in all experimental work is the objective of assessing algorithm performance.

9.2 Performance Measures

Assessing the quality of an evolutionary algorithm commonly implies experimental comparisons between the given EA and other evolutionary or traditional algorithms. Even if showing the superiority of some EA is not the main goal, parameter tuning for good performance still requires experimental work to compare different algorithm variants.

Such comparisons always assume the use of some algorithm performance measures, since claims about ranking algorithms are always meant in terms of their relative performances rather than, for instance, code length or readability. Because EAs are stochastic, performance measures are statistical in nature, meaning that a number of experiments need to be conducted to gain sufficient experimental data, as noted in Sect. 7.5. In the following we discuss three basic performance measures:

- success rate
- effectiveness (solution quality)
- efficiency (speed)

Additionally, we discuss the use of progress curves, i.e., plots of algorithm behaviour against time.

9.2.1 Different Performance Measures

In quite a few cases, experimental research concerns problems where either the optimal solution can be recognised, which is typical in academia, or a criterion for sufficient solution quality can be given, as in many practical applications. In these cases one can easily define a success criterion: finding a solution of the required quality, and the **success rate** (SR) measure can be defined as the percentage of runs where this happens. For problems where the optimal solutions cannot be recognised, the SR measure cannot be used in theory. This is the case if the optimum of the objective function is unknown, or if perhaps not even a lower/upper bound is available. Nevertheless, a success criterion in the *practical* sense can often be given even in these cases. For example, think of a university timetabling problem. The theoretical optimum for any given year is surely unknown here. However, one could use last year's timetable, or the one made by hand as benchmark and declare that a run ending with a timetable beating the benchmark by 10% is a success. Practical success criteria can also be used even in cases when the theoretical optimum is known, but the user does not require this optimum. For instance, it might be sufficient if we have a solution with an error less than a given $\epsilon > 0$.

The **mean best fitness** measure (MBF) can be defined for any problem that is tackled with an EA – at least for any EA using an explicit fitness measure (thus excluding, for instance, interactive evolution applications, Sect. 14.1). For each run of a given EA, we record the fitness of the best individual at termination. The MBF is the average of these values over all runs.

Note that although SR and MBF are related, they are different, and there is no general advice on which one to use for algorithm comparison. The difference between the two measures is rather obvious: SR cannot be defined for some problems, while the MBF is always a valid measure. Furthermore, all possible combinations of low or high SR and MBF values can occur. For example, low SR and high MBF is possible and indicates a good approximizer algorithm: it gets close consistently, but seldom really makes it. Such an outcome could motivate increasing the length of the runs, hoping that this allows the algorithm to finish the search. An opposite combination of a high SR and low MBF is also possible, indicating a 'Murphy algorithm': if it goes wrong, it goes very wrong. That is, those few runs that terminate without an (optimal) solution end in a disaster, with a very bad best fitness value deteriorating MBF. Clearly, whether the first or the second type of algorithm behaviour is preferable depends on the problem. As mentioned above, for a timetabling problem the SR measure might not be meaningful, so one should be interested in a high MBF. To demonstrate the other situation, think of solving the 3-SAT problem with the number of unsatisfied clauses as fitness measure. In this case a high SR is pursued, since the MBF measure – although formally correct – is useless because the number of unsatisfied clauses at termination says, in general, very little about how close the EA got to a solution. Notice,

however, that the particular application objectives (coming from the original problem-solving context) might necessitate a refinement of this picture. For instance, if the 3-SAT problem to be solved represents a practical problem, with some tolerance for a solution, then measuring MBF and striving for a good MBF value might be appropriate.

In addition to the mean best fitness calculated over a number of runs, in specific cases one might be interested in the best-ever or the worst-ever fitness. As discussed above, for design problems, the best-ever fitness is more appropriate than MBF, since *one* excellent solution is all that is required. For repetitive problems the worst-ever fitness can be interesting, as it can be used for studying worst-case scenarios and can help to establish statistical guarantees on solution quality.

It is important to note that for both SR and MBF, it is assumed that they are measured using an a priori specified limit of computational efforts. That is, SR and MBF always reflect performance within a fixed maximum amount of computing. If this maximum is changed, the ranking of algorithms might change as well. This is illustrated in Fig. 9.1, which shows a 'tortoise and hare' situation, where algorithm A (the hare) shows rapid progress, and in the case of limited time it beats algorithm B (the tortoise). In turn algorithm B outperforms algorithm A if given more time. Summarising, SR and MBF are performance measures for an algorithm's **effectiveness**, indicating how far can it get within a given computational limit.

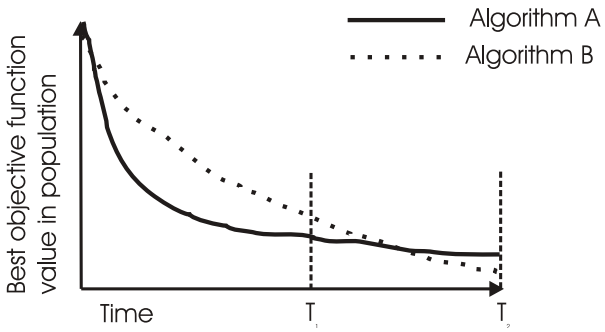

Fig. 9.1. Comparing algorithms A and B by after terminating at time T_1 and T_2 (for a minimisation problem). Algorithm A clearly wins in the first case, while B is better in the second one

The complementary approach is to specify when a candidate solution is satisfactory and measure the amount of computing needed to achieve this solution quality. Roughly speaking, this is the issue of algorithm **efficiency** or speed. Speed is often measured in elapsed computer time, CPU time, or user time. However, these measures depend on the specific hardware, operating system, compiler, network load, and so on, and therefore are ill-suited

for reproducible research. In other words, repeating the same experiments, possibly elsewhere, may lead to different results. For generate-and-test-style algorithms, such as EAs, a common way around this problem is to count the number of points visited in the search space. Since EAs immediately evaluate each newly generated candidate solution, this measure is usually expressed as the number of fitness evaluations. Of necessity, because of the stochastic nature of EAs, this is always measured over a number of independent runs, and the **average number of evaluations to a solution** (AES) is used. It is important to note that the average is only taken over the successful runs (that is, 'to a solution'). Sometimes the average number of evaluations to termination measure is used instead of the AES, but this has clear disadvantages. Namely, for runs finding no solutions, the specified maximum number of evaluations will be used when calculating this average. This means that the values obtained will depend on how long the unsuccessful runs are allowed to continue. That is, this measure mixes the AES and the SR measures, and the outcome figures are hard to interpret.

Using the AES measure generally gives a fair comparison of algorithm speed, but its usage can be disputed, or even misleading in some cases:

1. First, if an EA uses 'hidden labour', for instance, some local search heuristics incorporated in the mutation operator. The extra computational efforts may increase performance, but are invisible to the AES measure.
2. Second, if some evaluations take longer than others. For instance, if a repair mechanism is applied, then evaluations invoking this repair take much longer. One EA with good variation operators might proceed by chromosomes that do not have to be repaired, while another EA may need a lot of repair. The AES values of the two may be close, but the second EA would be much slower, and this is not an artifact of the implementation.
3. Third, if evaluations can be done very quickly compared with executing other steps in the EA cycle.[3] Then the AES does not truly reflect algorithm speed as other components of the EA have a relatively large impact.

An additional problem with AES is that it can be difficult to apply for comparing an EA with search algorithms that do not work in the same search space, in the same fashion. An EA iteratively improves complete candidate solutions, so each elementary search step consists of the creation and testing of one new candidate solution. However, a constructive search algorithm works in the space of partial solutions (including the complete ones through which an EA is searching), so one elementary search step consists of extending the current solution. In general, counting the number of elementary search steps is misleading unless the nature of those steps is the same. A possible treatment for this, and also for the hidden labour problem, is to compare the scale-up behaviour of the algorithms. This requires a problem that is scalable, i.e., its

[3] Typically this is not the case, and around 70–90% of the time is spent on fitness evaluations.

size can be changed. The number of variables is a natural scaling parameter for many problems. Two different types of methods can then be compared by plotting their own speed measure figures against the problem size. Even though the measures used in each curve are different, the steepness information is a fair basis for comparison: the curve that grows at a higher rate indicates an inferior algorithm (Fig. 9.2). A great advantage of this comparison is that it can also be applied to plain running times (e.g., CPU times), not only to the number of abstract search steps. As discussed above, there are important arguments against using running times themselves for comparisons. However, the scale-up curves of running times do give a fair comparison without those drawbacks.

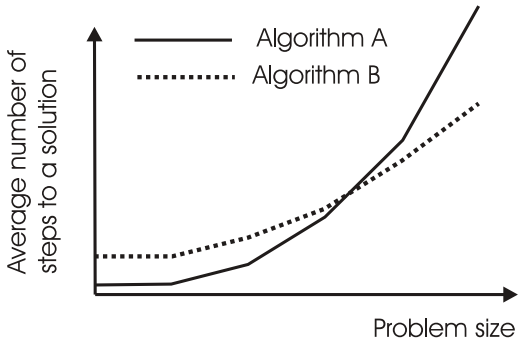

Fig. 9.2. Comparing algorithms A and B by their scale-up behaviour. Algorithm B can be considered preferable because its scale-up curve is less steep

Success percentages and run lengths can be meaningfully combined into a measure expressing the amount of processing required to solve a problem with a given probability [252, Chap. 8]. This measure (defined for generational EAs and used frequently in GP) depends on the population size and the number of generations as tuneable quantities. The probability $Y(\mu, i)$ that a given run with population size μ hits a solution for the first time in generation i is estimated by observed statistics, which require a substantial number of runs. Cumulating these estimations, we can calculate the probability $P(\mu, i)$ that a given generation i will contain a solution (found in a generation $j \leq i$), and hence the probability that generation i finds a solution at least once in R runs as $1 - (1 - P(\mu, i))^R$. Then the number of independent runs needed to find a solution by generation i with a probability of z is

$$R(\mu, i, z) = \left\lceil \frac{log(1 - z)}{log(1 - P(\mu, i))} \right\rceil , \tag{9.1}$$

where $\lceil \ \rceil$ is the ceiling function. Being a function of the population size, this measure can give information on how to set μ. For instance, after collecting

enough data with different settings, the total amount of processing, that is, the number of fitness evaluations, needed to find a solution with a probability of z by generation i using a population size μ is $I(\mu, i, z) = \mu \cdot i \cdot R(\mu, i, z)$. Notice that the dependence on μ is not the crucial issue here; in fact, any algorithm parameter p can be used in an analogous way to estimate for $R(p, i, z)$.

Another alternative to AES, especially in cases where one cannot specify satisfactory solution quality in advance, is the pace of progress to indicate algorithm speed. Here the best (or alternatively the worst or average) fitness value of the consecutive populations is plotted against a time axis – typically the number of generations or fitness evaluations (Fig. 9.1). Clearly, such a plot provides much more information than the AES, and therefore it can also be used when a clear success criterion is available. In particular, a progress plot can help rank two algorithms that score the same on AES. For example, progress curves might disclose that algorithm A has achieved the desired quality halfway through the run. Then the maximum number of evaluations might be decreased and the competition redone. The chance is high that algorithm A keeps its performance, e.g., its MBF, at lower costs and algorithm B does not, thus a well-motivated preference can be formulated. Another possible difference between progress curves of algorithms can be the steepness towards the end of the run. If, for instance, curve A has already flattened out, but curve B did not, one might extend the runs. The chance is high that B will make further progress in the extra time, but A will not; thus again, the two algorithms can be distinguished.

A problem with using such progress plots is that it is hard to use them in a statistical way. Averaging the data of, say, 100 runs and only drawing the average plot can hide interesting effects by smoothening them out. Overlaying all curves forms an alternative, but has obvious disadvantages: it might result in a chaotic figure with too much black ink and no visible structure. A practical solution is depicting a typical curve, that is, one single plot that is representative for all others. This option might not have a solid statistical basis, but it can deliver the most information when used with care.

9.2.2 Peak Versus Average Performance

For some, but not all, performance measures, there is an additional question of whether one is interested in peak performance, or average performance, considered over all these experiments. In evolutionary computing it is typical to suggest that algorithm A is better than algorithm B if its average performance is better. In many applications, however, one is often interested in the best solution found in X runs or within Y hours/days/weeks (peak performance), and the average performance is not that relevant. This is typical, for instance, in design problems as discussed in Section 9.1. In general, if there is time for more runs on the given problem and the final solution can be selected from the best solutions of these runs, then peak performance is more relevant than average performance.

We have a different situation if a problem-solving session only allows time for one run that must deliver the solution. This might be the case if a computationally expensive simulation is needed to calculate fitness values or for a real-time application, like repetitive and on-line control problems. Here, an algorithm with high average performance and small standard deviation is the best option, since it carries the lowest risk of missing the only chance we have.

It is interesting to note that academic experimental EC research falls in the first category – there is always time to perform more runs on any given set of test problems. In this light, it is strange that the huge majority of experimental EC research is comparing average performances of algorithms. This might be because researchers do not consider the differences between design and repetitive problems, and do not realise the different implications for the requirements of the problem-solving algorithm. Instead, it seems, they simply assume that the EA will be used in the repetitive mode.

Next we consider an example to show how the interpretation of figures concerning averages and standard deviations can depend on application objectives. In EC it is common to express preferences for algorithms with better averages for a given performance measure, e.g., higher MBF or lower AES, especially if a better average is coupled to a lower standard deviation. This attitude is never discussed, but it is less self-evident than it might seem. Using the timetabling example, let us assume that two algorithms are compared based on 50 independent runs and the resulting MBF values that are given in Fig. 9.3. Given these results, it could be tempting to conclude that algorithm A is better, because of the slightly higher MBF, and the more consistent behaviour (that is, lower variation in best fitness values at termination). This is indeed a sound argument in the case of a repetitive application, for instance, if a team of hospital employees must be scheduled every morning, based on fresh data and constraints. Notice, however, that six runs of algorithm B terminated with a solution quality that algorithm A never achieved. Therefore in a design application algorithm B is preferable, because of the higher chance of delivering a better timetable. Making a university timetable would fall into this category, since it has to be made only once a year, and the data is available weeks, perhaps months, before the timetable must be effective. This discussion of performance measures is not exhaustive, but it illustrates the point that for a sound comparison it is necessary to specify the objectives of the algorithms in light of some problem-solving context and to derive the performance measures used for comparison from these objectives.

Finally, let us pay some attention to using statistics. It is clear that by the stochastic nature of EAs only statistical statements about behaviour are possible. Typically, averages and standard deviations supply the necessary basis for claims about (relative) performance of algorithms. In a few cases these can be considered as sufficient, but in principle it is possible that two (or more) series of runs deliver data that are statistically indistinguishable, i.e., may come from the same distribution, and the differences are due to random effects. This means that the two series, and the behaviour of the two

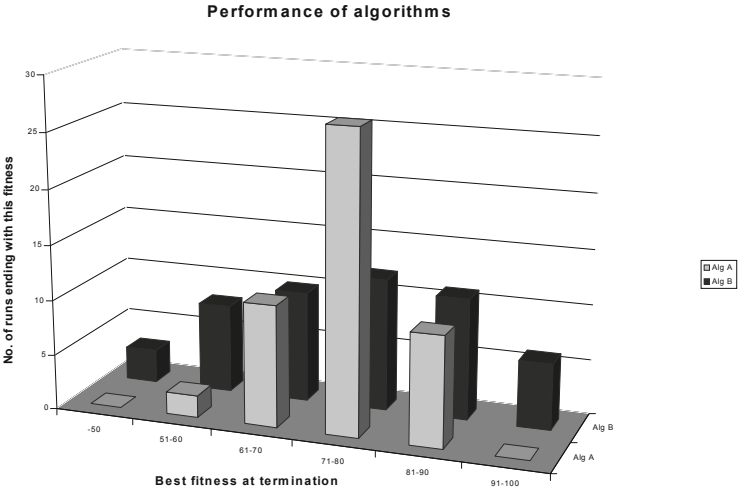

Fig. 9.3. Comparing algorithms by histograms of the best found fitness values

EAs behind these series, should be considered as statistically identical, and claims about one being better are ill-founded. It is important to recall that the mean and standard deviation of any given observable are only *two* values by which we try to describe the whole set of data. Consequently, considering only the standard deviations often cannot eliminate the possibility that any observed difference is only a result of randomness.

Good experimental practice therefore requires the use of specific tests to establish whether observed differences in performance are truly statistically significant. A popular method for this purpose is the two-tailed t-test, which gives an indication about the chance that the values came from the same underlying distribution. The applicability of this test is subject to certain conditions, for instance, that the data are normally distributed, but in practice it proves rather robust and is often used without verifying these conditions. When more than two algorithms are being compared, it is suggested that an analysis of variance (ANOVA) test be performed. This uses all of the data to calculate a probability that any observed differences are due to random effects, and should be performed *before* comparing algorithms pairwise. More sophisticated variants also exist: for instance, if we want to investigate the effects of two parameters (say, population size and mutation rate), then we can perform a two-way ANOVA, which simultaneously analyses the effects of each parameter and their interactions.

If we have limited amounts of data, or our results are not normally distributed, then using t-tests and ANOVA is not appropriate. For example, if we are comparing MBF values for a number of algorithms, with $SR < 1$ for some (but not all) of them, then our data will almost certainly not be normally distributed, since there is a fixed upper limit to the MBF value defined by the problem's optimum. In this type of cases, it is better to use the equivalent rank-based non-parametric test, noting that it is less likely to show a difference, since it makes fewer assumptions about the nature of the data.

Unfortunately, the present experimental EC practice seems rather unaware of the importance of statistics. This is a great shame, since there are many readily available software packages for performing these tests, so there is no excuse for not performing a proper statistical analysis of results. However this problem is easily remediable. There are any number of excellent books on statistics that deal with these issues, aimed at experimental sciences, or business and management courses, see for instance [290, 472]. The areas of concern are broadly known as hypothesis testing, and experimental design. Additionally, a wealth of information and on-line course material can be found by entering these terms into any Internet search engine.

9.3 Test Problems for Experimental Comparisons

In addition to the issue of performance measures, experimental comparisons between algorithms require a choice of benchmark problems and problem instances. We distinguish three different approaches:

1. Using problem instances from an academic benchmark repository.
2. Using problem instances created by a problem instance generator.
3. Using real-life problem instances.

9.3.1 Using Predefined Problem Instances

The first option amounts to obtaining prepared problem instances that are freely available from Web-based repositories, monographs, or other printed literature. In the history of EC some objective functions had a large impact on experimental studies. For instance, the so-called De Jong test suite, consisting of five functions has long been very popular [102]. This test suite was carefully designed to span an interesting variety of fitness landscapes. However, both computing power and our understanding of EAs have advanced considerably since the 1970s. Consequently, a modern study that only showed results on these functions and then proceeded to make general claims would not be considered methodologically sound. Over the last decade other functions have been added to the 'obligatory' list and are used frequently, such as the Ackley, Griewank, and Rastrigin functions, just to name the most popular ones. Obviously, new functions pose new challenges to evolutionary algorithms, but

the improvement is still rather quantitive. To put it plainly: How much better is a claim based on ten test landscapes than one only using the five De Jong functions? There is, of course, a straightforward solution to this problem by limiting the scope of statements about EA performance and restricting it to *the functions used* in the comparative experiments. Formally, this is a sound option, but in practice these careful refinements can easily skip the reader's attention. Additionally, the whole EC community using the same test suite can lead to overfitting new algorithms to these test functions. In other words, the community will not develop better and better EAs over the years, but only better and better EAs for these problems!

Another problem with the present practice of using particular objective functions or fitness landscapes is that these functions do not form a systematically searchable collection. That is, using 15 such functions will deliver 15 data points without structure. Unfortunately, although we have some ideas about the sorts of features that make problems hard for EAs, we do not currently possess the tools to divide these functions into meaningful categories, so it is not possible to draw conclusions on the relationship between characteristics of the problem (the objective function) and the EA behaviour. A deliberate attempt by Eiben and Bäck [130] in this direction failed in the sense that the EA behaviour turned out to be inconsistent with the borders of the test function categories. In other words, the EAs showed different behaviours within one category and similar behaviours on functions belonging to different categories. This example shows that developing a meaningful classification of objective functions or test landscapes is nontrivial because the present vocabulary to describe and to distinguish test functions seems inappropriate to define good categories (see [239] for a good survey of these issues). For the time being this remains a research challenge [135].

Building on cumulative experience in the EC community, for instance that of Whitley et al. [454], Bäck and Michalewicz gave some general guidelines for composing test suites for EC research in [29]. Below we reproduce the main points from their recommendations. The test suite should contain:

1. A few unimodal functions for comparisons of convergence velocity (efficiency), e.g., AES.
2. Several multimodal functions with a *large* number of local optima (e.g., a number growing exponentially with n, the search space dimension). These functions are intended to be representatives of the characteristics that are typical for real-world problems, where the best out of a number of local optima is sought.
3. A test function with randomly perturbed objective function values models a typical characteristic of numerous real-world applications and helps to investigate the robustness of the algorithm with respect to noise.
4. Constrained problems, since real-world problems are typically constrained, and constraint handling is a topic of active research.

5. High-dimensional objective functions, because these are representative of real-world applications. Furthermore, low-dimensional functions (e.g., with $n = 2$) are not suitable representatives of application problems where an evolutionary algorithm would be applied, because they can be solved optimally with traditional methods. Most useful are test functions that are *scalable* with respect to n, i.e., which can be used for arbitrary dimensions.

9.3.2 Using Problem Instance Generators

An alternative to such test landscapes is formed by problem instances of a certain (larger) class, for instance, operations research problems, constrained problems or machine-learning problems. The related research communities have developed their collections, like the OR library http://www.ms.ic.ac.uk/info.html, the constraints archive at http://www.cs.unh.edu/ccc/archive, or the UCI Machine Learning Repository at http://www.ics.uci.edu/~mlearn/MLRepository.html. The advantage of such collections is that the problem instances are interesting in the sense that many other researchers have investigated and evaluated them already. Besides, an archive often contains performance reports of other techniques, facilitating direct feedback on one's own achievements.

Over the last few years there has been a growing research interest in using problem instance generators. Using such a generator, which could of course come from an archive, means that problem instances are produced on-the-spot. Generators usually have some problem-specific parameters, for example, the number of clauses and the number of variables for 3-SAT, or the number of variables and the extent of their interaction for NK landscapes [244], and can generate random instances for each parameter value. The advantage of this approach is that the characteristics of the problem instances can be tuned by the generator's parameters. In particular, for many combinatorial problems there is a lot of information available about the location of really hard problem instances, the so-called phase transition, related to the given parameters of the problem [80, 183, 344]. A generator makes it possible to perform a systematic investigation in and around the hardest parameter range. Thus one can create results relating problem characteristics to algorithm performance. An illustration is given in Fig. 9.4. The question "which of the two algorithms is better" can now be refined to "which algorithm is better on which problem instances". On mid-range parameter values (apparently the hardest instances) algorithm B outperforms algorithm A. On the easier instances belonging to low and high parameter values, this behaviour is reversed.

9.3.3 Using Real-World Problems

Testing on real data has the advantages that results can be considered as very relevant viewed from the application domain (data supplier). However, it also has some disadvantages. Namely, practical problems can be overcomplicated.

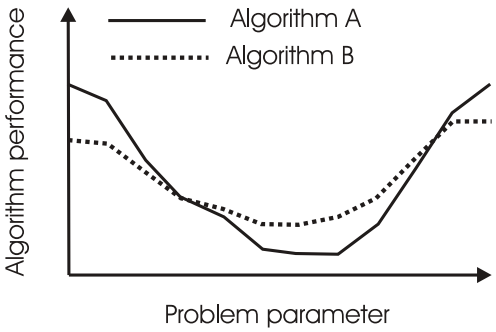

Fig. 9.4. Comparing algorithms on problem instances with a scalable parameter

Furthermore, there can be few available sets of real data, and these data may be commercially sensitive and therefore difficult to publish and to allow others to compare. Last, but not least, there might be so many application-specific aspects involved that the results are hard to generalise. Despite these drawbacks it remains highly relevant to tackle real-world problems as the proof of the pudding is in the eating!

9.4 Example Applications

As mentioned in the introduction to this chapter, instead of presenting two case studies with implementation details, we next describe examples of good and bad practice, in order to illustrate some of our points.

9.4.1 Bad Practice

This section shows a hypothetical example of an experimental study following the template that can be found in many EC publications.[4] In this imaginary case a researcher has invented a new EA feature, e.g., "tricky mutation", and assessed the value of this new feature by running a standard GA and "tricky GA" 20 times independently on each of 10 objective functions chosen from the literature. The outcomes of these experiments proved tricky GA better on seven, equal on one, and worse on two objective functions in terms of SR. On this basis it was concluded that the new feature is indeed valuable.

The main question here is what did we, the EC community, learn from this experience? We did learn a new feature (tricky mutation) and obtained some indication that it might be a promising idea to try in a GA. This can of course justify publishing a paper reporting this; however, there are also many things that we did not learn here, including:

[4] The authors admit that some of their own papers also follow this template.

- How relevant are these results, e.g., are the test functions typical of real-world problems, or important only from an academic perspective?
- What would have happened if a different performance metric had been used, or if the runs had been ended sooner, or later?
- What is the scope of claims about the superiority of the tricky GA?
- Is there a property distinguishing the seven good and two bad functions?
- Are these results generalisable? Alternatively, do some features of the tricky GA make it applicable for other specific problems, and if so which?
- How sensitive are these results to changes in the algorithm's parameters?
- Are the performance differences as measured here statistically significant, or can they be just artifacts caused by random effects?

The next example explicitly addresses some of these issues and therefore forms a showcase for a better, albeit still not perfect, practice.

9.4.2 Better Practice

A better example of how to evaluate the behaviour of a new algorithm takes into account questions such as:

- What type of problem am I trying to solve?
- What would be a desirable property of an algorithm for this type of problem, for example: speed of finding good solutions, reliably locating good solutions, or occasional brilliance?
- What methods currently exist for this problem, and why am I trying to make a new one, i.e., when do they not perform well?

After considering these issues, a particular problem type can be chosen, a careful set of experiments can be designed, and the necessary data to collect can be identified. A typical process might proceed along the following lines:

- inventing a new EA (xEA) for solving problem X
- identifying three other EAs and a traditional benchmark heuristic for problem X in the literature
- asking when and why xEA could be better than the other four methods
- obtaining a problem instance generator for problem X with two parameters: n (problem size) and k (some problem-specific indicator)
- selecting five values for k and five values for n
- generating 100 random problem instances for all 25 combinations
- executing all algorithms on each instance 100 times (the benchmark heuristic is also stochastic)
- recording AES, SR, and MBF values and standard deviations (not for SR)
- identifying appropriate tests based on the data and assessing the statistical significance of results
- putting the program code and the instances on the Web

The advantages of this template with respect to the one in the previous example are numerous:

- The results can be arranged in 3D: that is, as a performance landscape over the (n, k) plane with special attention to the effect of n on scale-up.
- The niche for xEA can be identified, e.g., weak with respect to other algorithms for (n, k) combinations of type 1, strong for (n, k) combinations of type 2, comparable otherwise. Thus the 'when' question can be answered.
- Analysing the specific features and the niches of each algorithm can shed light on the 'why' question.
- A lot of knowledge has been collected about problem X and its solvers.
- Generalisable results are achieved, or at least claims with well-identified scope based on solid data.
- Reproduction of the results, and further research elsewhere, is facilitated.

For exercises and recommended reading for this chapter, please visit
www.evolutionarycomputation.org.