
Journal of Mathematical Modelling and Algorithms **3:** 323–347, 2004.

Helper-Objectives: Using Multi-Objective Evolutionary Algorithms for Single-Objective Optimisation

MIKKEL T. JENSEN
The EVALife Group, Department of Computer Science, University of Aarhus, Denmark

Abstract. This paper investigates the use of multi-objective methods to guide the search when solving single-objective optimisation problems with genetic algorithms. Using the job shop scheduling and travelling salesman problems as examples, experiments demonstrate that the use of *helper-objectives* (additional objectives guiding the search) significantly improves the average performance of a standard GA. The helper-objectives guide the search towards solutions containing good building blocks and help the algorithm escape local optima. The experiments reveal that the approach works if the number of simultaneously used helper-objectives is low. However, a high number of helper-objectives can be used in the same run by changing the helper-objectives dynamically. The experiments reveal that for the majority of problem instances studied, the proposed approach significantly outperforms a traditional GA.

The experiments also demonstrate that controlling the proportion of non-dominated solutions in the population is very important when using helper-objectives, since the presence of too many non-dominated solutions removes the selection pressure in the algorithm.

Mathematics Subject Classification (2000): 68T20.

Key words: multi-objective optimisation, helper-objectives, multi-objectivisation, genetic algorithms, optimisation.

1. Introduction

Over the last decade, there has been intense research activity on evolutionary algorithms (*EAs*) for multi-objective optimisation. Multi-objective EAs are algorithms capable of optimising several different objectives at the same time, and can identify a set of solutions representing non-dominated trade-offs between conflicting objectives. They return an approximation of the *Pareto-optimal* set, the truly non-dominated solutions in the search-space. Well known multi-objective EAs include NSGA-II [8], PESA-II [6] and SPEA [25]. The focus of multi-objective optimisation so far has been on developing new algorithms capable of getting closer to the true Pareto-optimal set and achieving a better spread of the solutions returned.

Recent research [4, 17] indicates that methods from Pareto-based multi-objective optimisation may be helpful when solving single-objective optimisation problems. Important problems in single-objective optimisation include: (i) avoiding

local optima, (ii) keeping diversity at a reasonable level, and (iii) making the algorithm identify good building blocks that can later be assembled by crossover. The purpose of this paper is to further investigate if multi-objective methods can be used to overcome these difficulties in single-objective optimisation.

Traditional single-objective optimisation methods focus on one objective exclusively (denoted the *primary objective* in the following), but in this paper the notion of *helper-objectives* will be introduced. The idea is that the simultaneous optimisation of the primary objective and a number of helper-objectives may perform better than an algorithm focusing on the primary objective alone. This can happen if the helper-objective is chosen in such a way that it helps maintain diversity in the population, guides the search away from local optima, or helps the creation of good building blocks. Most multi-objective algorithms were designed to create diverse non-dominated fronts. Hence population diversity should be increased if the helper-objectives are in conflict with the primary objective.

The problems studied in this paper are the job shop scheduling problem using a total flow-time primary objective and the travelling salesman problem. For both problems, a number of helper-objectives and a multi-objective algorithm for minimising the primary objective will be developed, and the results of this algorithm will be compared to the results of a traditional genetic algorithm working only with the primary objective. The experiments reveal that on average the multi-objective algorithms perform significantly better than the traditional algorithms.

The outline of the paper is as follows. The next section discusses related work. Section 3 introduces helper-objectives, using the job shop scheduling problem as an example. The first subsection defines the helper-objectives for this problem and discusses what helper-objectives are and how they can be applied. An algorithm using helper-objectives as well as a traditional GA for the job shop scheduling problem are developed in Section 3.2 and tested in Section 3.3. The experiments demonstrate that the new approach outperforms the traditional algorithm, but that keeping the proportion of non-dominated solutions at a low level by niching may further enhance the performance. A few simple niching schemes are incorporated into the algorithm and tested in Section 3.4. Section 4 discusses the work done on the travelling salesman problem. The first subsection describes the local search used by the algorithms, while Section 4.2 discusses helper-objectives for the TSP. The algorithms are presented in Section 4.3, while experiments are presented in Section 4.4. The last section concludes the paper and discusses future work.

2. Related Work

The idea of decomposing a single-objective problem into several objectives was investigated as early as 1993 by Louis and Rawlins [19]. Their work demonstrated that a deceptive problem could be solved more easily using Pareto-based selection than standard selection in a GA.

Knowles et al. [17] proposed *multi-objectivisation*, a concept closely related to the helper-objectives proposed in this paper. The idea in multi-objectivisation is to decompose the optimisation problem into subcomponents by considering multiple objectives. The authors study mutation-based evolutionary algorithms, and argue that decomposition into several objectives will remove some of the local optima in the search-space, since a solution has to be trapped in all objectives in order to be truly "stuck". The problems investigated are the hierarchical-if-and-only-if function (*HIFF*) and the travelling salesman problem. For both problems an objective function of the form $A + B$ is decomposed into two separate objectives A and B. In experiments, the algorithms based on multi-objectivisation outperform comparable algorithms based on the single objective.

The main difference between the study of Knowles et al. [17] and the present one is in the details of how multi-objective methods are used to solve the single-objective problem. Knowles et al. disregard the primary objective in their experiments and use only what is referred to in this paper as helper-objectives.⋆ Furthermore, they use only static objectives, whereas the objectives used in this paper are changed dynamically.

The idea of using additional objectives to escape local optima has been used in a different setup by Wright [24]. Wright studied a timetabling problem using simulated annealing and threshold accepting algorithms. He introduced *sub-costs*, an idea closely related to the helpers used in this paper. The algorithms would accept moves degrading the objective provided at least one sub-cost was improved sufficiently. The approach was found to significantly improve performance when comparing to similar algorithms not using sub-costs.

In [5], Coello Coello and Aguirre develop a GA for the generation of logical circuits. They decompose the objective function into a high number of objectives, one objective corresponding to each of the outputs of the circuits. Their approach is based on a VEGA-like algorithm in which a subpopulation is created for each objective. The subpopulations cooperate to generate a solution that produces only correct outputs. Additionally, the algorithm rewards solutions using a small number of logical gates, provided that all outputs are correct. The algorithm is demonstrated to produce better solutions than human designers and a previous GA-based approach.

Another application of multi-objective methods in single-objective optimisation is the reduction of bloat (uncontrolled growth of solution size) in genetic programming [4, 7]. Bleuler et al. [4] and de Jong et al. [7] independently studied the effect of using the size of the genetic program as an extra objective in genetic programming. Both studies find that the additional objective both reduces the average size of the solutions found and decreases the processing time needed to find an optimal solution. A related study is [9], in which bloat in a GP solving a regression problem is reduced with a multi-objective method.

⋆ They remark that keeping the primary objective is also an option.

Scharnow et al. [20] present theoretical arguments that the single source shortest path problem can be solved more efficiently by an EA when seen as multi-objective than as a single-objective problem. The authors show that the problem can be solved in expected time $O(n^3)$ when cast as multi-objective, and argue that in some cases seeing the problem as single-objective will be equivalent to a needle-in-a-haystack problem, giving a much higher processing time. Scharnow et al. assume that non-existing edges in the graph and infeasible solutions are represented as infinite weights. However, non-existing edges and infeasible solutions can also be represented using large (finite) weights. In this case the expected running time of the single-objective algorithm will be identical to that of the multi-objective algorithm.

3. Helper-Objectives and Job Shop Scheduling

This section introduces helper-objectives using the job shop scheduling problem (*JSSP*) [13] as an example. The JSSP is probably the most intensely studied scheduling problem in literature. It is NP-hard. A job shop problem of size $n \times m$ consists of n jobs $J = \{J_i\}$ and m machines $M = \{M_j\}$. For each job J_i a sequence of m operations $(o_{i1}, o_{i2}, \ldots, o_{im})$ describing the processing order of the operations of J_i is given. Each operation o_{ij} is to be processed on a certain machine and has a processing time τ_{ij}. The scheduler has to decide when to process each of the operations, satisfying the constraints that no machine can process more than one operation at a time and no job can have more than one operation processed at at time. Furthermore, there can be no preemption; once an operation has started processing, it must run until it is completed.

Several performance measures exist for job shop problems. The performance measure used in this paper is the *total flow-time*

$$F_\Sigma = \sum_{i=1}^{n} F_i, \tag{1}$$

where F_i is the flow-time of job i, the time elapsed from the beginning of processing until the last operation of J_i has completed processing.

3.1. HELPER-OBJECTIVES

When selecting helper-objectives for our problem, we should ensure that the helper-objectives are *in conflict with the primary objective*, at least for some parts of the search-space. Optimising the primary objective and a helper-objective simultaneously will be equivalent to simply optimising the primary objective if the objectives are not conflicting. Furthermore, helper-objectives should reflect some aspect of the problem that we expect could be helpful for the search. For the total flow-time job shop problem, the flow-times of the individual jobs F_i are a natural choice as helper-objectives. Minimising the flow-time of a particular job will often be in

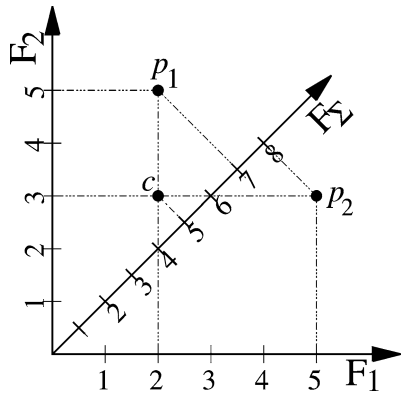

Figure 1. Illustration of helper-objective for the total flow-time JSSP.

conflict with minimising the sum of flow-times. However, since a decrease in the total flow-time can only happen if the flow-time is decreased for at least one of the jobs, the minimisation of individual flow-times may improve the minimisation of the total flow-time. Furthermore, the crossover operator may be able to combine a schedule with a low flow-time for J_i but a high total flow-time with a schedule with a low total flow-time to produce a schedule with an even lower total flow-time.

The effect of a helper-objective on the JSSP is illustrated on Figure 1. For simplicity, the figure deals with a problem instance with only two jobs, meaning the total flow-time can be calculated $F_\Sigma = F_1 + F_2$. The plot has F_1 and F_2 along the two main axes. Since F_Σ is a linear function of F_1 and F_2, the F_Σ value of a given solution (point in the plot) can be found as a projection of the point onto the third axis (the F_Σ-axis). Consider the two solutions p_1 and p_2 in the plot, and assume F_1, F_2 and F_Σ to be the objectives of a Pareto-based GA. Since p_1 has $F_\Sigma = 7, F_1 = 2, F_2 = 5$ and p_2 has $F_\Sigma = 8, F_1 = 5, F_2 = 3$, neither of the solutions Pareto-dominate the other.* If there are no more solutions in the population, p_1 and p_2 will both be assigned the same fitness in a Pareto-based EA. Consider the recombination of p_1 and p_2. Assuming the crossover-operator to be respectful and preserve good building blocks (i.e. the flow-times of individual jobs), the outcome of such a recombination could be the solution c. This solution has the low F_1 of p_1 and the low F_2 of p_2, meaning a F_Σ (the primary objective) lower than both of the parents. Of course, other outcomes of such a recombination are also possible.

In a job shop of n jobs, there are n different helper-objectives of the F_i kind. The benchmarks used in this paper have between 10 and 50 jobs and potential helper-objectives. Using that many helper-objectives simultaneously is probably more harmful than beneficial, since most of the individuals in the population could

* Since p_1 has the lowest F_1 and F_Σ-values, p_2 does not dominate p_1. p_2 has the lowest F_2-value, so p_1 does not dominate p_2. Recall that solution a has to be no worse than solution b in all objectives and better in at least one in order for a to Pareto-dominate b.

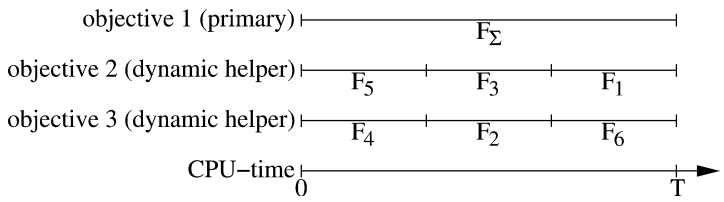

Figure 2. The use of dynamic helper-objectives.

be non-dominated very early in the program run, and the focus of the search will be shifted away from the primary objective to an unacceptable degree.

A solution to this problem is to use only a subset of the helper-objectives at any given time, and sometimes change the helper-objectives. The "slots" in the program used to hold the current helper-objectives will be termed *dynamic helper-objectives*. When the helper-objectives are changed during the program run, we need to decide how to schedule the changes. Using h dynamic helper-objectives and having a total of H helper-objectives to be used in a run of T CPU-seconds, each helper-objective can be used for

$$D = \frac{hT}{H}$$

CPU-seconds, assuming all helper-objectives to be used for the same amount of time. Potentially, the helper-objectives could be changed for every generation, but since there will not be time for good building blocks to be formed during such a short period, the opposite approach was taken in the experiments: each helper-objective was used for one period of the maximal length possible. Every D generations the helper-objectives were changed to the next objectives in the *helper-objective sequence*. The sequence in which the helper-objectives are used may have an influence on the search, but since we have no way to know which order is the best one, a random ordering of the helper-objectives was used. This is illustrated for a six-job problem in Figure 2. The illustration uses two dynamic helper-objectives, meaning that at any given time the primary objective and two of the six helper-objectives are in use. At fixed time intervals, the dynamic helper-objectives are changed to the next objectives from the helper-objective sequence.

The experiments with the multi-objective algorithm were conducted using one, two and three dynamic helper-objectives and n static helper-objectives, where n is the number of jobs in the problem instance in question. In the following, an algorithm using x helper-objectives simultaneously will be called an *x-helper algorithm*.

3.2. THE ALGORITHMS

The multi-objective algorithm used as a starting point for the algorithm used in the experiments is the *Non-dominated Sorting GA version II*, NSGA-II, published by

Deb et al. [8]. This algorithm has been demonstrated to be among the most efficient algorithms for multi-objective optimisation on a number of benchmarks.

When selecting a multi-objective algorithm for single-objective optimisation, the running time of the algorithm is an important issue. A computational overhead will be associated with using the helper-objectives, and it is important to minimise this overhead as much as possible. A traditional single-objective GA runs in time $O(GN)$, where G is the number of generations and N is the population size. The traditional implementation of the NSGA-II runs in time $O(GMN^2)$, where M is the number of objectives [8]. However, recent research [15] shows that the algorithm can be modified to run in time $O(GN \log^{M-1} N)$. The vast majority of multi-objective algorithms known today have running times no better than this.

A detailed description of the NSGA-II is outside the scope of this paper. The interested reader is referred to [8, 15]. A very brief description of the algorithm follows: (i) Fitness is assigned based on non-dominated sorting, (ii) All individuals not dominated by any other individuals are assigned front number 1. All individuals only dominated by individuals in front number 1 are assigned front number 2, etc. Selection is made in size two tournaments. If the two individuals are from different fronts, the individual with the lowest front number wins. If they are from the same front, the individual with the highest crowding distance wins; in this way higher fitness is assigned to individuals located on a sparsely populated part of the front. In every generation N new individuals are generated. These compete with their parents (also N individuals) for inclusion in the next generation.

Other multi-objective algorithms, such as SPEA [25], PDE [1] or PESA-II [6] could have been used as a starting point for the helper-objective algorithm, but note that in their basic forms all of these have longer processing times than the improved NSGA-II. This may result in longer processing times (or conversely, less fitness evaluations in the same processing time).

The traditional stopping criterion used when working with GAs is the number of fitness evaluations used. This stopping criterion was used in a previous study of helper-objectives [14]. However, the multi-objective part of the helper-objective algorithm requires more processing time than a standard single-objective algorithm, and it can be argued that comparing a helper-objective algorithm and a standard algorithm on the basis of a fixed number of fitness evaluations is not entirely fair. Using the same amount of processing time, the single-objective algorithm may be able to perform more fitness evaluations than the helper-algorithm. In this study, the processing time used by the program is used as a stopping criterion. For the experiments on job shop scheduling, the programs were allowed to spend 0.02 CPU-seconds per operation in the problem instance.* Thus for a 200 operation instance, the programs were allowed to run for 4 CPU-seconds. The computers used in the experiments were 1.7 GHz pentium 4 PCs running Linux. The programs where implemented in C++ and compiled using the gnu compiler.

* Since the stopping criterion is only tested at the end of each generation, the programs may actually run for one generation more than this.

3.2.1. *Multi-Objective GA for the JSSP*

The genetic representation used in the algorithm was the *permutation with repetition* representation widely used for the job shop problem, see, e.g., [13]. In this representation, a schedule is represented by a sequence of job numbers, e.g., $(1, 3, 1, 2, \ldots)$. If job J_i consists of k operations, then there will be k instances of i in the sequence. Decoding is done using the Giffler–Thompson schedule builder [11]. This decoder builds the schedule from left to right, adding one operation at a time. The sequence of the gene is interpreted as a list of priorities on the operations. The gene $(1, 3, 1, 2, \ldots)$ gives the highest possible priority to the first operation of job 1, followed by the first operation of job 3, the second operation of job 1, etc. The permutation with repetition representation has the advantage that it cannot represent infeasible solutions. All active schedules can be represented, which guarantees that the optimal schedule is in the search-space.

The algorithm uses the Generalised Order crossover operator (*GOX*) and the position based mutation operator (*PBM*), both widely used in scheduling research. The GOX operator selects a substring in one parent, deletes the operations corresponding to the substring in the other parent, and inserts the substring at the position of the first deleted operation. The PBM operator moves an operation picked from a random position in the string to another position. For more detailed descriptions on GOX and PBM, see [13].

The crossover rate was set to 1.0 and mutation was performed on offspring after crossover. These choices were made after experiments with two different ways of doing crossover: (i) GOX followed by PBM and (ii) GOX alone.* Experiments were conducted using one dynamic helper-objective on the `la26` instance for crossover rates 0.0, 0.1, 0.2, ..., 1.0. For each crossover rate 500 runs of the algorithms were performed, and the best performing parameter setting was selected. Using a resampling test (see Appendix A) the parameter setting of crossover rate 1.0 and mutation after crossover was found to perform significantly better than all the other parameter settings tested. After fixing the crossover rate, an experiment to select the population size was performed. The population sizes 10, 20, ..., 500 were tested, again running the algorithm 500 times on the `la26` instance for each parameter setting. A population size in the range 90–120 gave the best performance, so the setting 100 was used in the rest of the experiments.

3.2.2. *Traditional GA for the JSSP*

A traditional GA for solving the JSSP was created. It used the same representation, decoding and genetic operators as the multi-objective algorithm. The algorithm used an elite of one. As in the multi-objective algorithm, selection was done in two-tournaments. Since the algorithm is fundamentally different from the multi-

* In an earlier study, [14], the possibility of using the PPX crossover operator was investigated as well. It was found to be inferior to GOX and was not used in this study.

objective algorithm, it was deemed inadequate to simply use the parameter settings found for that algorithm.

Experiments were conducted to identify good settings of the crossover rate and population size. The method used to identify these parameters was identical to the one used for identifying parameters for the helper-algorithm. For the traditional GA, a crossover rate of 0.8, no mutation after crossover and a population size of 120 were observed to be the best choices.

3.3. EXPERIMENTS

The multi-objective algorithm was run with 1, 2 and 3 dynamic helper-objectives, and with static helper-objectives using all n helper-objectives at the same time. Experiments were performed on the 24 problem instances listed in Table I. The instances prefixed by `ft` are from [10], the instances prefixed by `la` are from [18], and the instances prefixed by `swv` are from [23].

Each experiment consisted of 500 runs, from which the average best performance was calculated. Similar experiments were conducted for the traditional GA, the results are shown in Table II. The table holds one row for every problem instance. The five rightmost columns correspond to the five algorithms tested, while the leftmost column reports the lowest known total flow-time for the problem instance.* The five rightmost columns report the average best total flow-time F_Σ in percent above the best known value. Resampling tests have been performed to compare the averages statistically. Performances marked '$+$' are significantly better (i.e. have a lower average) than the performance produced by the traditional algorithm, while numbers marked '$-$' are significantly worse (i.e. have a higher average) than the performance produced by the traditional algorithm. For every problem instance, the best performance has been printed in bold, and this number is significantly lower than numbers not marked '$*$'. The last row holds the total flow-time for each algorithm averaged over all the instances.

Table I. The benchmarks used in the experiments

Size	Instances	Size	Instances
10×5	`la01,la02`	20×10	`la26,la27,swv01,swv02`
15×5	`la06,la07`	30×10	`la31,la32`
20×5	`la11,la12,ft20`	15×15	`la36,la37`
10×10	`la16,la17,ft10`	20×15	`swv06,swv07`
15×10	`la21,la22`	50×10	`swv11,swv12`

* Since the author is unaware of any other results on the lowest total flow-times for the problems used, the best known values are the best results achieved in the experiments.

Table II. Average best total flow-times in percent above the best known value. In each row, the lowest number has been printed in bold. For brevity, standard deviations are not included in the table. For all of the experiments the standard deviation lies between 1.0% and 3.0% of the average best performance, the average value being 1.8%

	Best known	Traditional GA	1-helper	2-helper	3-helper	n-helper
la01	4832	4.0232	2.9180+	**2.4731**∗+	4.1805	8.9963−
la02	4468	4.2346	**3.1513**∗+	3.5698+	5.3850−	12.0636−
la06	8694	7.4350	**5.7672**∗+	7.7835−	10.8259−	16.4125−
la07	8219	7.6530	**5.7829**∗+	7.6396	10.8103−	17.1177−
la11	14801	6.8259	**5.1969**∗+	7.9332−	11.2087−	18.3096−
la12	12490	7.4516	**5.8799**∗+	8.7790−	12.4980−	22.1041−
la16	7393	5.8569	4.9317+	**4.3879**∗+	5.3280+	9.0248−
la17	6555	4.0153	3.3822∗+	**3.2006**∗+	4.0641	7.1960−
la21	12990	5.8052∗	**5.6120**∗	6.5050−	9.3002−	14.1901−
la22	12106	5.6030	**5.3825**∗+	5.8351−	8.3727−	14.3780−
la26	20538	5.7401	**5.3827**∗+	6.7928−	9.2662−	13.8285−
la27	20992	6.3577∗	**6.2671**∗	7.8792−	10.2720−	14.7066−
la31	39929	6.0485	**5.6130**∗+	6.9476−	8.9737−	14.2711−
la32	42951	6.4017∗	**6.2394**∗	7.8096−	9.5779−	14.0160−
la36	17073	5.4765	5.5655	**5.2334**∗+	6.8477−	10.9705−
la37	17886	**5.3869**∗	5.8716−	6.2965−	8.5782−	12.9459−
ft10	7557	8.2572	7.1602+	**6.1493**∗+	7.1232+	10.5624−
ft20	14350	9.2397	**7.8321**∗+	11.0704−	15.5986−	27.8753−
swv01	21444	9.3094∗	**9.0841**∗	10.5960−	13.7516−	21.7306−
swv02	22082	9.3302	**8.7202**∗+	10.1458−	13.3339−	21.4854−
swv06	29051	**8.6537**∗	9.3332−	9.9081−	12.2237−	18.2754−
swv07	27839	**8.5391**∗	9.3527−	9.7633−	12.1086−	18.0326−
swv11	111890	9.3836∗	**9.3082**∗	9.8221−	13.0585−	29.7921−
swv12	110183	11.7348	**11.1930**∗+	12.2730−	15.6204−	32.4346−
Average	–	7.0318	**6.4553**	7.4497	9.9295	16.6967

Considering the average over all the instances, on average the traditional GA is 7.03% above the best known value, while for the 1-helper algorithm this number is 6.45%. This difference corresponds to an average difference of total flow-times of 81.7. When considering the total averages, the two-, three- and n-helper algorithms all produce averages higher than the traditional algorithm.

Focusing on the individual problem instances, the experiments reveal that in most cases the multi-objective algorithm using one helper-objective performs better than the traditional algorithm. For this parameter setting, the multi-objective

Table III. Average number of fitness evaluations for each type of algorithm. The last row reports the number of fitness evaluations relative to the traditional algorithm

Algorithm	Traditional	1-helper	2-helper	3-helper	n-helper
Fitness evals	21373	20408	18937	17839	15932
Relative fitness evals	100%	95.5%	88.6%	83.5%	74.5%

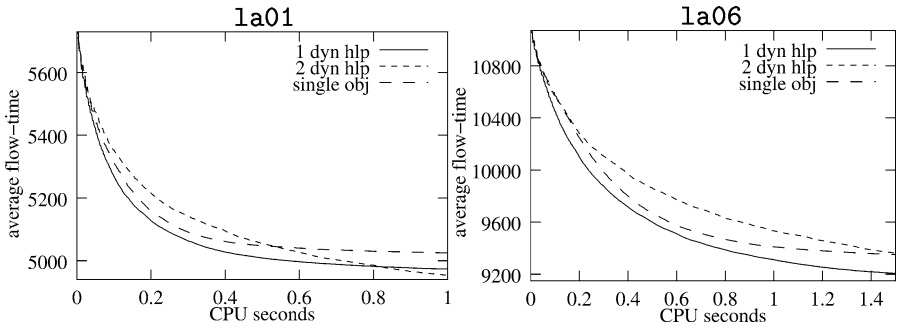

Figure 3. Average best total flow-time as a function of the processing time for the `la01` (left) and `la06` (right) instances.

algorithm produces a lower average best F_Σ than the traditional algorithm for 20 out of the 24 problem instances. In 16 of these cases, a permutation test revealed the difference to be statistically significant. For four instances (`la36`, `la37`, `swv06`, and `swv07`), the traditional algorithm performed better than the one-helper algorithm, the difference being statistically significant in three of these cases.

Regarding the two-helper algorithm, in most cases it is inferior to the one-helper and the traditional algorithm. However, there are a number of cases (`la01`, `la16`, `la17`, `la36` and `ft10`) for which this algorithm outperforms all the other algorithms. The algorithm using the maximal number of helper-objectives performs much worse than the other algorithms for every single problem instance, while the three-helper algorithm is always inferior to the one- and two-helper algorithms.

The average number of fitness evaluations used in each algorithm has been printed in Table III. The averages have been taken over all the problems. The table reveals that the overhead associated with using the helper-objectives increases as the number of helper-objectives increases. For one helper-objective, the algorithm has time to do 95.5% of the fitness evaluations done in the traditional GA, but as the number of helper-objectives increases, this percentage drops to 74.5% when using the maximal number of helpers.

The performances of the traditional algorithm and multi-objective algorithm using one and two dynamic helper-objectives are shown on Figure 3. The plots show the average best total flow-time as a function of the processing time for the

la01 and la06 problem instances. Comparing the algorithm using one dynamic helper-objective to the traditional algorithm, it is evident that both algorithms start off at the same level of performance, but after a few generations the multi-objective algorithm has a lower average best total flow-time. For both instances, the difference between the two algorithms slowly gets larger as time progresses. This behaviour was observed for all of the instances except la27, la36, la37, swv06 and swv07.

Considering the two-helper algorithm, it performs worse than both of the other algorithms in the early stages of the run of both diagrams. For la01 in the late stages of the run, the two-helper algorithm decreases the flow-time faster than the other algorithms, and ends up with a slightly better performance than the one-helper algorithm. For la06 the one-helper algorithm clearly beats the two-helper algorithm.

The plot for la01 is typical for the instances for which the two-helper algorithm has the best performance. The progression of this algorithm is slow during the early stages of the run, but in the later stages it overtakes the other algorithms.

3.3.1. *Investigating the Size of the Non-Dominated Set*

The mediocre performance of the multi-objective algorithm for many dynamic helper-objectives may be caused by too large non-dominated sets when the algorithm is running. If most of the population belongs to the first non-dominated front, most of the population will be assigned the same fitness (disregarding crowding), and the selection pressure will be very low. Theoretical investigations reveal that for random solutions the proportion of non-dominated solutions can be expected to grow with the number of objectives [3]. The average size of the first non-dominated front was investigated for all of the instances. Since the size of the front cannot be expected to be constant during the run, the dependency on the CPU-time spent was also investigated. Experiments were made for each combination of problem instance and multi-objective algorithm of Table II.

The average size of the first non-dominated front is shown in Figure 4 for the instances la01 and la06. In each diagram, the average front-size is plotted for the algorithms using 1, 2 and 3 dynamic helper-objectives. The front-size for the algorithm using all n helper-objectives is not plotted, since the average front-size was always equal to the population size (100). This was the case for all the instances, and accounts for the poor performance of this algorithm. The sudden drops in average front-sizes on the plots is caused by the changes of helper-objectives; every time the helper-objectives are changed, there is a decrease in the average front-size. The front-size then gradually increases until the helper-objectives are changed again. Comparing the algorithms to each other, it is evident that the average front-size is always large for the algorithm using three helper-objectives. This was found to be the case for all of the instances, and the low selection pressure resulting from the large front-size probably explains the poor performance for the three-helper algorithm.

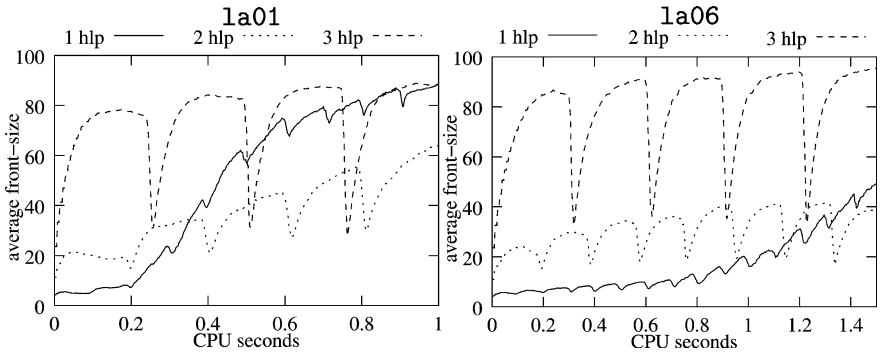

Figure 4. Average size of the non-dominated front as a function of processing time for the `la01` (left) and `la06` (right) instances.

Considering the one-helper algorithm, the front-size at the beginning of the run is low in both cases, which is also the case for all other problem instances. For the `la06` instance, the front-size stays at a relatively low level during the entire run, while for `la01` it increases and almost becomes equal to the population size at the end of the run. Compared to the two-helper algorithm, the front-size starts at a lower level, but it increases much more. For the `la01` instance, the front-size of the one-helper algorithm becomes larger than that of the two-helper algorithm after approximately 0.35 CPU-seconds.

Behaviour similar to this was found in all the instances for which the two-helper algorithm performed better than the one-helper algorithm. For all of these instances, the average front-size of the one-helper algorithm was larger than in the two-helper algorithm after around one third of the processing time or earlier. For the other instances, the front-size of the one-helper algorithm stayed below that of the two-helper algorithm during most of the run. This strongly indicates that keeping the size of the first non-dominated front small is an important element in tuning this kind of algorithm.

Another reason for the mediocre performance of the algorithms using many helper-objectives could be that as the number of helper-objectives increases, the proportion of non-dominated solutions with a low primary objective decreases. This can be seen by investigating moves in the search-space as perceived by a search-algorithm. When considering only the primary objective, moves that do not degrade the primary objective will be seen as acceptable. When a helper-objective is added, moves that are incomparable (i.e. none of the solutions dominate each other) will be seen as acceptable. This allows the algorithm to escape local optima in the primary objective by making seemingly ‘bad’ moves. Judging from the experiments with one dynamic helper-objective, the ability to escape local optima more than compensates for these bad moves. However, when more dynamic helper-objectives are added, more bad moves are allowed. The experiments indicate that when many helper-objectives are used simultaneously, the disadvantage of the bad moves outweighs the advantage of escaping local optima.

3.4. CONTROLLING THE SIZE OF THE NON-DOMINATED SET

When the size of the non-dominated set increases, it is often because there is a high number of identical individuals in the non-dominated set. The presence of many identical individuals is self-perpetuating, since once many identical individuals are present, there is a high probability of two identical individuals mating, creating even more indviduals of this kind. The presence of many identical individuals is a sign that the crowding scheme used is not powerful enough, so a method that will be termed *niche enforcement* was tested. In niche enforcement, after the non-dominated sorting of the population, all individuals that share the same objective values are identified, and if more individuals than a prespecified value (called the *maximum niche-count*) are present, excess individuals are removed at random until only the allowed number of individuals share the same objective values. Preliminary experiments revealed that the performance of the algorithm using niche enforcement was quite sensitive to the value of the maximal niche count, and that a maximal niche-count of 2 was a good value.

Instead of improving the crowding mechanism a simpler scheme was also tested. Prior to the crossover operation, the two parents were compared. If they turned out to have the same objective values (primary and helpers), they were deemed too similar, and one parent was replaced by the best of two randomly chosen individuals from the population. The test was not carried out a second time; crossover would be performed if the new parent was similar to the old parent. In the following, this scheme will be referred to as *in-breeding control*.

Experiments with the one-helper algorithm using niche enforcement only, in-breeding control only, and niche enforcement and in-breeding control simultaneously were carried out, the results are in Table IV. The table has been constructed in the same way as Table II, except that the '+' now means *"statistically better than the standard one-helper algorithm"*, while '−' means *"statistically worse than the standard one-helper algorithm"*. By inspecting the averages of the last row we realize that on average all of the methods for controlling the size of the non-dominated set improve performance when comparing to the standard one-helper algorithm. The best overall algorithm uses niche enforcement as well as in-breeding control, on average it is 6.07% above the best known total flow-time. Averaged over all the problem instances, this algorithm has a total flow-time which is 82.0 better than the standard one-helper algorithm and 163.7 better than the traditional algorithm. However, inspecting the table reveals that for individual problem instances the pure in-breeding control and niche enforcement algorithms achieve the best performance for a number of instances. Comparing the standard one-helper algorithm to the algorithms controlling the size of the non-dominated set reveals that there is only a single case (`swv07`) in which the standard algorithm significantly outperforms the improved algorithms, but that the opposite is the case for 21 of the instances.

Comparing the performance of the improved algorithms to the performance of the two-helper algorithm from Table II reveals that in the cases for which the two-

Table IV. Average best total flow-times in percent above the best known value for the experiments on controlling the size of the non-dominated set. For all of the experiments the standard deviation lies between 1.1% and 3.0% of the average best performance, the average value being 1.9%

	Traditional GA	Standard 1-helper	In-breed control	Niche enforcement	In-breed cntr. +niche enf.
la01	4.0232−	2.9180	2.6635+	2.1275∗+	**2.1006**∗+
la02	4.2346−	3.1513	3.0394	**2.6365**∗+	2.6432∗+
la06	7.4350−	5.7672∗	5.7787∗	**5.6407**∗	5.8270∗
la07	7.6530−	5.7829∗	**5.5518**∗	5.5931∗	5.7087∗
la11	6.8259−	5.1969∗	**5.1875**∗	5.2888∗	5.3625∗
la12	7.4516−	5.8799∗	**5.7766**∗	6.1017	6.0865
la16	5.8569−	4.9317	4.9317	3.8996∗+	**3.8374**∗+
la17	4.0153−	3.3822	3.3425	2.4119∗+	**2.3570**∗+
la21	5.8052	5.6120	5.4527	**5.2386**∗+	5.4257∗
la22	5.6030−	5.3825	5.2007∗	**5.0355**∗+	5.0438∗+
la26	5.7401−	5.3827∗	**5.1889**∗	5.3106∗	5.2206∗
la27	6.3577	6.2671	**5.8194**∗+	6.2143	6.0604
la31	6.0485−	5.6130	**5.3976**∗+	5.6082	5.5160∗
la32	6.4017	6.2394	**5.9139**∗+	6.2210	6.0827∗
la36	5.4765	15.5655	5.2873+	4.9113∗+	**4.8468**∗+
la37	**5.3869**∗+	5.8716	5.6374+	5.3964∗+	5.4093∗+
ft10	8.2572−	7.1602	6.7739+	**5.5366**∗+	5.6200∗+
ft20	9.2397−	7.8321∗	7.8523∗	**7.7937**∗	7.8105∗
swv01	9.3094	9.0841∗	**8.7428**∗	9.2455	8.9582∗
swv02	9.3302	8.7202∗	**8.4032**∗	8.6500∗	8.4127∗
swv06	**8.6537**∗	9.3332	8.9746∗+	9.2038	8.7205∗+
swv07	**8.5391**∗	9.3527	9.0628	8.9730+	9.0125
swv11	9.3836	9.3082	**8.6956**∗+	9.0870+	8.7101∗+
swv12	11.7348	11.1930∗	11.2163∗	11.3044∗	**11.0134**∗
Average	7.0318	6.4553	6.2455	6.1429	**6.0744**

helper algorithm outperformed the one-helper algorithm (la01, la16, la17, la36 and ft10), the two-helper algorithm is outperformed by the algorithms using niche-enforcement. The in-breeding control and niche enforcement methods were also tested on the two-helper algorithm, but it was inferior to the one-helper algorithms using the same methods.

The average size of the non-dominated sets were investigated as a function of the processing time. This revealed that the size of the first non-dominated front was substantially reduced by niche-enforcement. Typically, the size of the non-dominated front stayed below 10 during the entire program run. The in-breeding

control scheme delayed the growth of the non-dominated set, but for instances such as la01, most of the population would still be non-dominated at the end of the run.

4. The Travelling Salesman Problem

The travelling salesman problem (TSP) is a classical combinatorial optimisation problem. It consists of a set of N cities $c_1, \ldots, c_N$ and an associated $N \times N$ distance matrix M. The entries in M represent the distances between the cities, so $M(c_1, c_2)$ is the distance from c_1 to c_2. The objective is to find a Hamiltonian path (a circular path visiting each city exactly once) with the smallest possible total distance. If $\boldsymbol{\pi} = (\pi_1, \pi_2, \ldots, \pi_N)$ is a permutation of $(1, 2, \ldots, N)$ representing the tour of the cities, then the distance associated with the tour can be calculated as

$$D(\boldsymbol{\pi}) = \sum_{i=1}^{N} M(c_{\pi[i]}, c_{\pi[i \oplus 1]}), \tag{2}$$

$$\text{where } i \oplus 1 = \begin{cases} i+1 & \text{if } i < N, \\ 1 & \text{if } i = N. \end{cases}$$

For a good introduction to the TSP see [16]. The problem instances used in this paper are all Euclidean instances in the plane. They are all available for download at the TSPLIB website* or the travelling salesman website at Princeton.** Each instance is identified by one to three letters, followed by the number of cities. The problem instances have between 99 and 2103 cities.

4.1. LOCAL SEARCH

There has been substantial work on solving the TSP using a number of heuristics, including GAs. Current research indicates that if a GA solving the TSP is to be competitive, it should be hybridised with specialised TSP methods, such as the Lin–Kernighan algorithm [16].

The GAs used in this paper employ a simple local search heuristic called 2-opt [16, 2]. The 2-opt heuristic works by repeatedly applying moves of the kind illustrated on Figure 5. In one move, the endpoints of the edges (c_1, c_2) and (c_3, c_4) are interchanged so the edges (c_1, c_3) and (c_2, c_4) appear instead. The move is only carried out if it improves the tour. This is repeated until no more improving moves can be made.

In order to obtain an efficient implementation of the 2-opt local search, assume a fixed orientation of the tour. Each possible 2-opt move can be represented as a four-tuple $[c_1, c_2, c_3, c_4]$, where (c_1, c_2), (c_3, c_4) are the edges deleted from the original tour, and (c_1, c_3), (c_2, c_4) are the edges added to the new tour. Observe that for symmetric problem instances the same move can also be represented by the tuple $[c_3, c_4, c_1, c_2]$. Clearly, a move cannot improve the tour unless at least one of the

* www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/
** http://www.math.princeton.edu/tsp/

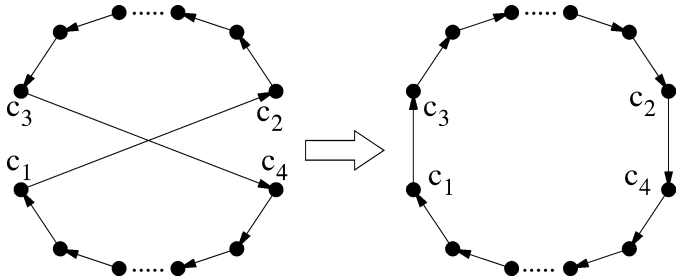

Figure 5. The *2-opt* move used by the local search procedure.

deleted edges is replaced by a shorter edge. Thus, when considering tuples of the form $[c_1, c_2, c_3, c_4]$, we need only consider c_3 and c_4 that satisfy the condition

$$M(c_1, c_2) > M(c_1, c_3) \vee M(c_1, c_2) > M(c_2, c_4).$$

This will allow moves that replace the (c_1, c_2) edge by a shorter edge. Moves that replace the (c_3, c_4) edge by a shorter edge can be allowed if we also consider moves represented by the tuple $[c_3, c_4, c_1, c_2]$.

When running the 2-opt local search, the algorithm starts by selecting an edge (c_1, c_2). It then tests all possible moves $[c_1, c_2, c_3, c_4]$ that satisfy the condition above, implementing the first improving move it finds. It then goes on to the next edge (c_1', c_2') and implements the first improving move it finds. This process continues until there are no more improving moves and the solution is locally optimal. Note that given the cities c_1 and c_2, we need not consider all possible choices of c_3 and c_4. For each city c, we keep a list $L^c = (c_a, c_b, \ldots)$ of other cities, sorted on the distance to c. When considering moves of the form $[c_1, c_2, c_3, c_4]$ with c_1 and c_2 fixed, we find the choices of c_3 and c_4 that satisfy the condition by traversing L^{c_1} until we find a c_3 for which $M(c_1, c_2) > M(c_1, c_3)$ does not hold, and by traversing L^{c_2} until we find a c_4 for which $M(c_1, c_2) > M(c_2, c_4)$ does not hold.

For solutions close to optimality, $M(c_1, c_2)$ can be expected to be low, meaning that few choices of (c_3, c_4) need to be tested.

4.2. HELPER-OBJECTIVES

We need to identify helper-objectives for the TSP. Knowles et al. [17] previously studied the TSP in their work on multi-objectivisation. They decomposed the TSP objective function (2) into two terms

$$D(\pi) = f_1(\boldsymbol{\pi}, a, b) + f_2(\boldsymbol{\pi}, a, b),$$

$$f_1(\boldsymbol{\pi}, a, b) = \sum_{i=\pi^{-1}[a]}^{\pi^{-1}[b]-1} M(c_{\pi[i]}, c_{\pi[i\oplus 1]}),$$

$$f_2(\boldsymbol{\pi}, a, b) = \sum_{i=\pi^{-1}[b]}^{N} M(c_{\pi[i]}, c_{\pi[i\oplus 1]}) + \sum_{i=1}^{\pi^{-1}[a]-1} M(c_{\pi[i]}, c_{\pi[i\oplus 1]}),$$

where $\pi^{-1}[x]$ denotes the position of x in π and a and b are parameters (cities) defining f_1 and f_2. Knowles et al. found f_1 and f_2 to be helpful for optimisation, but these objectives have a weakness for symmetric problems. Given two cities a and b defining f_1 and f_2, two different representations of the same tour can be Pareto-incomparable, with two different values for f_1 and f_2.

As an example, consider the choices $a = c_3$ and $b = c_4$. Now consider the two solutions $\pi_1 = (c_1, c_2, c_3, c_4)$ and $\pi_2 = (c_3, c_2, c_1, c_4)$. For a symmetric problem, π_1 and π_2 represent the same tour. Calculating the objectives f_1 and f_2 for π_1 and π_2 reveals that

$$
\begin{aligned}
f_1(\pi_1, 3, 4) &= M(c_3, c_4),\\
f_2(\pi_1, 3, 4) &= M(c_4, c_1) + M(c_1, c_2) + M(c_2, c_3),\\
f_1(\pi_2, 3, 4) &= M(c_3, c_2) + M(c_2, c_1) + M(c_1, c_4) = f_2(\pi_1, 3, 4),\\
f_2(\pi_2, 3, 4) &= M(c_4, c_3) = f_1(\pi_1, 3, 4).
\end{aligned}
$$

Thus, despite representing the same tour, generally π_1 and π_2 will be given different objectives, and will be non-identical, and incomparable in the Pareto-sense. For algorithms employing niching this is not a good property, since niching relies on being able to recognise identical solutions.

A helper objective without this problem is the following:

$$
h(\boldsymbol{\pi}, p) = \sum_{i=1}^{|p|} M(c_{\pi[\pi^{-1}[p[i]]\ominus 1]}, c_{p[i]}) + M(c_{p[i]}, c_{\pi[\pi^{-1}[p[i]]\oplus 1]}), \tag{3}
$$

where p is a subset of $\{1, 2, \ldots, N\}$ and $\ominus 1$ is the reverse of $\oplus 1$. The helper-objective $h(\boldsymbol{\pi}, p)$ is the sum of distances in the path incident on the cities in p. This helper-objective has the property that all solutions representing the same tour will share the same objective value.

Any number of helpers $h_1(\pi, p_1), h_2(\pi, p_2), \ldots, h_H(\pi, p_H)$ can be used. The helper-objectives were generated by creating a number of random sets $p_1, p_2, \ldots, p_H$, where each city had a 50% probability of being in a given set.

4.3. THE ALGORITHMS

The same algorithmic templates used for the JSSP were used for the TSP. The only differences were in the representation, decoding, local search and operators. The representation used was the permutation encoding often used for the TSP [22].

The recombination used in the algorithms was the improved edge recombination operator (*ER*) published by Starkweather et al. [22]. This operator was constructed to convey adjacency information present in both parents to the offspring. If the edge (c_a, c_b) is present in both parents, it will also be present in the child. Starkweather et al. found the ER operator to be superior to a number of other crossover operators for the TSP. The interested reader is referred to [22] for detailed information on how the operator works.

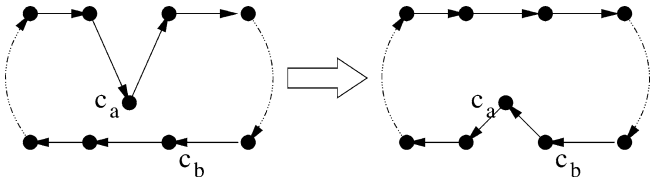

Figure 6. The mutation used by the TSP algorithms.

The mutation operator picks a random city in the tour, removes it and inserts it at a random position in the tour, see Figure 6 for an illustration. The local search of Section 4.1 was applied to all solutions produced by crossover or mutation.

In preliminary experiments, the time spent on local search to produce a locally optimal solution was found to be roughly proportional to $N^{3/2}$, where N is the number of cities. This can be justified by the fact that implementing a successful 2-opt move takes time proportional to N. Moreover, more moves are made on solutions with many cities before local optimality is achieved. Following this observation, the time allowed in a single GA run was made proportional to $N^{3/2}$. The CPU-time allowed for a run was set to $0.002N^{3/2}$ CPU-seconds.

4.3.1. *Multi-Objective GA for the TSP*

Since the experiments on the JSSP revealed that the algorithms using more than one dynamic helper-objective were inferior to the one-helper algorithm and its variants using control of the non-dominated set, only variants of the one-helper algorithm were tested for the TSP.

A set of experiments were conducted to identify the best number of helper-objectives. The experiments were made on the `pr439` problem instance, and the numbers 2, 4, 6, 8, 10, 12, 16, 20 and 40 were tested. For every parameter setting, 500 runs were made of the algorithm. The experiments revealed that the best setting was in the range 8–12, and 10 helper-objectives were used in the rest of the experiments.

The algorithm used a crossover rate of 0.7, with no mutation applied after crossover. Individuals not produced by crossover were created using the mutation operator. This choice was made after testing the crossover rates 0.0, 0.1, . . . , 1.0 on the `pr439` instance. The experiments revealed that the algorithm using a crossover rate of 0.7 showed the best performance, being significantly better than algorithms with crossover rates in the ranges 0.0 to 0.4 and 0.9 to 1.0.

The population sizes 10, 20, . . . , 300 were tested using 500 repetitions of the algorithm. The best performances was achieved with a population size of 100. This setting was found to be significantly better than population sizes in the ranges 10 to 70 and 170–300.

4.3.2. *Traditional GA for the TSP*

Experiments identical to the ones performed on the multi-objective algorithm were made for the traditional algorithm to set the crossover rate and population size. For this algorithm a crossover rate of 0.4 showed the best performance, being significantly better than algorithms with crossover rates in the ranges 0.0–0.1 and 0.5–1.0. A population size of 140 showed the best performance, being significantly better than population sizes in the ranges 10–70 and 150–300.

4.4. EXPERIMENTS

Experiments were conducted with the traditional GA, the standard one-helper algorithm, and three variants of the one-helper algorithm using control of the non-dominated set: only in-breeding control, only niche enforcement (using a maximum niche count of 2), and both. Each algorithm was run 500 times on every problem instance.

For each combination of algorithm and problem instance, the average best tour length as a percentage over the optimal tour length is given in Table V. Experiments were also conducted on the instances `pr144`, `pr226`, `pr264` and `rat99`, but since all of the algorithms found the optimal solution in all of the runs, these problems are not shown.

For brevity, standard deviations are not included in the table. However, for all of the experiments the standard deviation lies between 0.0% and 0.6% of the average best tour length, the average being 0.2%. Resampling tests have been performed to compare the numbers. In each row, numbers marked '$+$' are significantly lower than the traditional algorithm, while numbers marked '$-$' are significantly higher than the traditional algorithm. For each problem instance, the smallest number has been printed bold, and this number is significantly lower than numbers not marked '$*$'.

The numbers in the table clearly indicate that the traditional algorithm is inferior to the multi-objective algorithms. The traditional algorithm does not produce the lowest average best for any of the problem instances, and is significantly worse than at least one of the other algorithms for all of the problem instances except `pr299`. Considering the averages over all the problems, the best performer is the standard one-helper algorithm, achieving an average of 0.86% above the optimal solution. Considering the average standard deviation of 0.2%, this is much better than the average of 1.44% achieved by the traditional GA. When comparing the four variants of the one-helper algorithm to each other, it is evident that the choice of algorithm is dependent on the problem instance. The standard one-helper algorithm produces the best performance for the majority of the problem instances, but is inferior to the two algorithms using niche enforcement when considering some of the small and medium problem instances.

Table V. Average best tour lengths in percent above the optimal tour length for the experiments on the TSP. In each row, the lowest number has been printed bold

	Optimal	Traditional GA	Standard 1-helper	In-breed control	Niche enforcement	In-breed cntr. +niche enf.
pr107	440303	0.0102	0.0124	0.0115	0.0053∗+	**0.0039**∗+
pr124	59030	0.0023	0.0081−	0.0117−	**0.0000**∗+	**0.0000**∗+
pr136	96772	0.0695	**0.0527**∗+	0.0624	0.0530∗+	0.0561∗+
pr152	73682	0.0048	0.0037∗	0.0093−	0.0022∗	**0.0015**∗+
pr299	48191	0.1601∗	0.1555∗	0.1580∗	**0.1548**∗	0.1601∗
pr439	107217	0.0546	0.0501+	0.0489+	**0.0396**∗+	0.0411∗+
pr1002	259045	3.3346	**2.2969**∗+	2.3872+	2.5308+	2.5795+
rat195	2323	0.2217	0.2174	0.2238	**0.1502**∗+	0.1533∗+
rat575	6773	2.1276	**1.0075**∗+	1.0837+	1.1094+	1.1279+
rat783	8806	2.6488	**1.3761**∗+	1.4833+	1.6169+	1.6167+
d198	15780	0.4089	0.2437∗+	0.2454∗+	0.2444∗+	**0.2413**+
d493	35002	1.3294	**0.8313**∗+	0.8661∗+	0.9419+	0.9632+
d657	48912	1.6217	**0.7695**∗+	0.8318+	0.9008+	0.9005+
d1291	50801	1.3771	**1.0695**∗+	1.0738∗+	1.1250+	1.1264+
d1655	62128	4.0644	**3.4054**∗+	3.5005+	3.5939+	3.6198+
d2103	80450	1.3115	**0.7216**∗+	0.7910+	0.8873+	0.9423+
xqf131	564	0.0213	0.0266	0.0248	0.0142∗+	**0.0124**∗+
xqg237	1019	0.3837	0.3886	0.3925	**0.3553**∗+	0.3660∗
pma343	1368	0.9664	0.1462∗+	0.1520∗+	**0.1382**∗+	0.1389∗+
pka379	1332	1.3011	**0.3423**∗+	0.3611∗+	0.3949+	0.3911+
bcl380	1621	0.1246	0.1487−	0.1474−	0.0993∗+	**0.0913**+
pbl395	1281	0.2311	0.2287	0.2334	**0.1483**∗+	0.1655∗+
pbk411	1343	1.8139	0.3232+	0.3038+	**0.2331**∗+	0.2472∗+
pbn423	1365	0.1656	0.1612	0.1378∗+	0.1253∗+	**0.1216**∗+
pbm436	1443	0.7346	0.7048∗+	0.6979∗+	0.6868∗+	**0.6849**∗+
xql662	2513	0.7310	0.5324∗+	0.5487∗+	0.5487∗+	**0.5273**∗+
rbx711	3115	1.0417	**0.6199**∗+	0.6456∗+	0.6594+	0.6616+
rbu737	3314	2.1919	**1.0202**∗+	1.0278∗+	1.0905+	1.0763+
dkg813	3199	2.0791	**0.9500**∗+	1.0328+	1.1382+	1.1513+
lim963	2789	2.4077	**1.1671**∗+	1.2402+	1.3557+	1.3926+
pbd984	2797	2.2317	**1.0240**∗+	1.0955+	1.1984+	1.5316+
xit1083	3558	3.2532	**1.6526**∗+	1.8527+	1.9753+	1.9938+
dka1376	4666	3.2634	**2.1192**∗+	2.2220+	2.4106+	2.4535+
dca1389	5085	3.1526	**2.1780**∗+	2.2401+	2.4124+	2.4895+
dja1436	5257	3.9855	**2.8516**∗+	2.9873+	3.1413+	3.2277+
icw1483	4416	3.4823	**2.2431**∗+	2.6046+	2.7446+	2.8306+
Average	–	1.4447	**0.8622**	0.9093	0.9507	0.9747

5. Conclusion

This paper has demonstrated that multi-objective methods can be used for improving performance in single-objective optimisation. The notion of *helper-objectives* has been introduced, a helper-objective being an additional objective conflicting with the primary objective, but helpful for diversifying the population and forming good building blocks.

Experiments have demonstrated that for the total flow-time job shop scheduling problem, the performance of a traditional GA can be significantly improved by using helper-objectives. Using one dynamic helper-objective is the most promising approach, since using too many helper-objectives at the same time removes the selection pressure in the algorithm. Experiments revealed that controlling the size of the non-dominated set through niching is important in this kind of algorithm, and two kinds of control were investigated. They both improved performance significantly, but the method termed niche enforcement (removing identical solutions) is the most promising.

The idea was also tested on the travelling salesman problem. The experiments revealed that also for this problem performance could be improved significantly by using helper-objectives. The methods for control of the non-dominated set were tested on this problem, but the results were ambiguous; for some problem instances the use of niche enforcement improved performance, while for others the algorithm not using any kind of improved niching had the best performance.

The helper-objectives used on the JSSP and the TSP in this paper were found using clever guesswork and intuition. There is currently no methodology for finding good helper-objectives for a given problem. The objectives used on the TSP in [17] demonstrate that several different types of helpers can be efficient for the same problem. However, other problems may exist for which no helper-objectives can be found. In a preliminary study, the helpers used on the total flow-time JSSP in this paper were tested on a JSSP using makespan as the primary objective. This study revealed no beneficial impact from using the helpers. A methodology for determining a priori if a potential helper-objective is efficient is needed if helper-objectives are to be applied in a wider setting.

Scheduling the use of dynamic helper-objectives is the subject of future research. The experiments of this paper indicate that using one dynamic helper-objective is the best approach, but the question of when to change the helper-objective is still open. Adaptive selection of helpers, perhaps based on monitoring the helpers or the size of the non-dominated set could be an interesting direction of research.

Another direction of future research is applying the methods of this paper to other kinds of problems. In order for the methods to work on a problem, reasonable helper-objectives must be found. Combinatorial problems in which the objective consists of a sum or product of many terms can probably be solved using the same approach used in this paper; a helper-objective can be defined for each term or as a sum terms.

Acknowledgement

The author wishes to thank his colleagues at EVALife and the anonymous reviewers for helpful comments.

Appendix A. Comparing Averages Using Resampling

Traditional statistical testing relies on the data tested to conform to certain well-known distributions. However, it is often necessary to perform statistical testing on data that does not follow any of the standard distributions. Often researchers ignore the fact that data does not comply with the assumptions of traditional tests and apply tests such as ANOVA [21] anyway.

A viable alternative to traditional statistics is non-parametric tests such permutation tests or resampling [12]. These tests do not rely on the data to follow specific distributions. Furthermore, they are asymptotically just as powerful as traditional tests.

The averages produced by the experiments in this paper were compared using a permutation test as described in [12, pp. 36–38]. This test assumes the observations to come from two distributions that only differ by their means (that is $F_A(x) = F_B(x + \mu)$ for some μ, where F_A and F_B are the distribution functions of the observations), but this assumption is weaker than, e.g., the Gaussian assumption of ANOVA, and the test is robust to deviations from it. In this test, two series of observations A and B of equal sizes are compared by creating one large series of observations $C = A \cup B$. The observations in C are then randomly split into two series A'_i and $B'_i = C \setminus A'_i$ a large number of times $i = 1, \ldots, N$. For every A'_i and B'_i created, the test statistic $z_i = z(A'_i, B'_i)$ is calculated. Assuming the observations in A and B to have the same mean, the test statistic $z_0 = z(A, B)$ should not be extremal compared to $z_1, \ldots, z_N$. If z_0 is more extremal than $(1 - \alpha)N$ of the z_i, where α is the significance level of the test, then the hypothesis of equal means is rejected.

The test statistic used was the difference in averages between the two series of observations. Since a priori we have no reason to expect either of the two series to produce a lower average than the other one, the test was two-sided; both low and high values of the statistic were considered critical to the hypothesis of identical averages. The significance level was chosen to be $\alpha = 0.05$, and in each test $N = 20000$ random permutations were created.

References

1. Abbass, H. A., Sarker, R., and Newton, C.: PDE: A Pareto-frontier differential evolution approach for multi-objective optimization problems, in *Proceedings of CEC 2001*, Vol. 2, 2001, pp. 971–976.
2. Bentley, J. L.: Fast algorithms for geometric traveling salesman problems, *ORSA J. Comput.* **4**(4) (1992), 387–411.

3. Bentley, J. L., Kung, H. T., Schkolnick, M., and Thompson, C. D.: On the average number of maxima in a set of vectors and applications, *J. ACM* **25** (1978), 536–543.
4. Bleuler, S., Brack, M., Thiele, L., and Zitzler, C.: Multiobjective genetic programming: Reducing bloat using SPEA2, in *Proceedings of CEC'2001*, 2001, pp. 536–543.
5. Coello, C. A. C. and Aguirre, A. H.: Design of combinatorial logic circuits through an evolutionary multiobjective optimization approach, *Artificial Intelligence for Engineering, Design, Analysis and Manufacture* **16** (2002), 39–53.
6. Corne, D., Jerram, N., Knowles, J., and Oates, M.: PESA-II: Region-based selection in evolutionary multiobjective optimization, in L. Spector et al. (eds), *Proceedings of GECCO 2001: Genetic and Evolutionary Computation Conference*, 2001, pp. 283–290.
7. de Jong, E. D., Watson, R. A., and Pollack, J. B.: Reducing bloat and promoting diversity using multi-objective methods, in L. Spector et al. (eds), *Proceedings of GECCO 2001*, 2001, pp. 11–18.
8. Deb, K., Pratab, A., Agarwal, S., and Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II, *IEEE Trans. Evolut. Comput.* **6**(2) (2002), 182–197.
9. Ekárt, A. and Németh, S. Z.: Selection based on the Pareto nondomination critierion for controlling code growth in genetic programming, *Genetic Programming and Evolvable Machines* **2** (2001), 61–73.
10. Fisher, H. and Thompson, G. L.: Probabilistic learning combinations of local job-shop scheduling rules, in J. F. Muth and G. L. Thompson (eds), *Industrial Scheduling*, Prentice-Hall, 1963, pp. 225–251.
11. Giffler, B. and Thompson, G. L.: Algorithms for solving production scheduling problems, *Oper. Res.* **8** (1960), 487–503.
12. Good, P.: *Permutation Tests*, Springer, New York, 2000.
13. Jensen, M. T.: Robust and flexible scheduling with evolutionary computation, Ph.D. thesis, Department of Computer Science, University of Aarhus, 2001.
14. Jensen, M. T.: Guiding single-objective optimization using multi-objective methods, in G. Raidl et al. (eds), *Applications of Evolutionary Computation*, Lecture Notes in Comput. Sci. 2611, 2003, pp. 268–279.
15. Jensen, M. T.: Reducing the run-time complexity of multi-objective EAs: The NSGA-II and other algorithms, *IEEE Trans. Evolut. Comput.* **7**(5) (2003), 503–515.
16. Johnson, D. S. and McGeoch, L. A.: The travelling salesman problem: A case study, in E. Aarts and J. K. Lenstra (eds), *Local Search in Combinatorial Optimization*, Wiley, 1997, Chapt. 8.
17. Knowles, J. D., Watson, R. A., and Corne, D. W.: Reducing local optima in single-objective problems by multi-objectivization, in E. Zitzler et al. (eds), *Proceedings of the First International Conference on Evolutionary Multi-criterion Optimization (EMO'01)*, 2001, pp. 269–283.
18. Lawrence, S.: *Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*, Graduate School of Industrial Administration, Carnegie-Mellon University, 1984.
19. Louis, S. J. and Rawlins, G. J. E.: Pareto optimality, GA-easiness and deception, in S. Forrest (ed.), *Proceedings of ICGA-5*, 1993, pp. 118–123.
20. Scharnow, J. S., Tinnefeld, K., and Wegener, I.: Fitness landscapes based on sorting and shortest paths problems, in J. J. M. Guervós et al. (eds), *Proceedings of PPSN VII*, Lecture Notes in Comput. Sci. 2439, 2002, pp. 54–63.
21. Sokal, R. R. and Rohlf, F. J.: *Biometry*, W. H. Freeman and Company, 1995.
22. Starkweather, T., McDaniel, S., Mathias, K., Whitley, D., and Whitley, C.: A comparison of genetic sequencing operators, in R. Belew and L. Booker (eds), *Proceedings of the 4th International Conference on Genetic Algorithms*, 1991, pp. 69–76.
23. Storer, R. H., Wu, S. D., and Vaccari, R.: New search spaces for sequencing problems with applications to job shop scheduling, *Management Sci.* **38**(10) (1992), 1495–1509.

24. Wright, M.: Subcost-guided search – Experiments with timetabling problems, *J. Heuristics* **7** (2001), 251–260.
25. Zitzler, E. and Thiele, L.: Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach, *IEEE Trans. Evolut. Comput.* **3**(4) (1999), 257–271.