A Comparison of Nature Inspired Heuristics on the Traveling Salesman Problem

Thomas Stützle[1,2], Andreas Grün[1], Sebastian Linke[1] and Marco Rüttger[1]

[1] Darmstadt University of Technology, Computer Science Department,
Alexanderstr. 10, 64283 Darmstadt
[2] Université Libre de Bruxelles, IRIDIA,
Avenue Franklin Roosevelt 50, CP 194/6, 1050 Brussels, Belgium

Abstract. The Traveling Salesman Problem is a standard test-bed for algorithmic ideas. Currently, there exists a large number of nature-inspired algorithms for the TSP and for some of these approaches very good performance is reported. In particular, the best performing approaches combine solution modification or construction with the subsequent application of a fast and effective local search algorithm. Yet, comparisons between these algorithms with respect to performance are often difficult due to different implementation choices of which the one of the local search algorithm is particularly critical. In this article we experimentally compare some of the best performing recently proposed nature-inspired algorithms which improve solutions by using a same local search algorithm and investigate their performance on a large set of benchmark instances.

1 Introduction

The task in the Traveling Salesman Problem (TSP) is to find a shorted closed tour through a given set of n cities with known inter-city distances such that each city is visited exactly once and the tour ends at the start city. The TSP has played a central role in the development of many nature inspired heuristics like Evolutionary Computation [6, 7, 19, 22, 28], Ant Colony Optimization [4, 25], Neural Networks [1, 10], or Simulated Annealing [14, 15] to name just the most important. Currently, for several such approaches very good performance is reported. Examples are the genetic algorithms (GAs) by Merz and Freisleben [7, 19], Walters [28], Nagata and Kobayashi [22], and Gorges-Schleuter [8], the iterative partial transcription algorithm of Moebius et al. [21], the renormalization algorithm by Martin and Houdayer [11], Ant Colony Optimization (ACO) algorithms due to Dorigo and Gambardella [5] or Stützle and Hoos [27], and Iterated Local Search (ILS) algorithms [12, 13, 18, 17, 24]. Good computational results are also reported for some Simulated Annealing and Neural Network approaches, but, as described in the extensive review by Johnson and McGeoch [12], they seem not to be competitive with the previously mentioned algorithms.

The current state-of-the-art in the TSP suggests that the best performing algorithms combine solution construction or modification with the subsequent application of fast and effective local search algorithms [12, 24, 28]. Which of the existing possibilities is the most promising one for the TSP? One possible answer would be to look at published results. Yet, often it is difficult to compare these results because of differences in the local search implementation or the use of different local search algorithms, different computing environments like computer configuration, compiler or operating system, different stopping criteria etc. Certainly, if the reported results differ substantially, it is reasonable to conjecture that these differences are intrinsic to the algorithms compared. However, for the above cited approaches differences are typically small and concern tenth of percents of average deviation from optimal solutions or, if the same solution quality is reached, small factors with respect to computation time.

Here, we present a comparison of some of the best performing algorithms for the TSP. When comparing algorithms for the TSP, particularly important is that all algorithms use a same underlying local search algorithm, because its choice has a decisive influence on the final performance. Additionally, apparently small differences in the local search implementation may cause significant differences in the performance of these hybrid algorithms. For the TSP it is well known that the sophisticated Lin-Kernighan local search algorithm [16] often yields best performance with respect to solution quality. Yet, it is known that its run-time is very sensitive to the particular instance and the algorithm is rather complex and requires a lot of fine-tuning to run fast and produce high-quality solutions. On the other side, `2-opt` is rather straightforward to implement but the solution quality it returns is not very satisfactory. Therefore, here we settled on using an efficient implementation of `3-opt` by Tim Walters and that has been used in [28]. As our computational results will show, with the application of the `3-opt` algorithm very good performance is obtained, justifying the choice of this local search algorithm.

The article is structured as follows. In Section 2 we introduce details on the algorithms which are compared here. Then we give details on the experimental setup and the benchmark instances used in Section 3 and report the experimental results in Section 4. We end with some concluding remarks in Section 5.

2 Nature-inspired algorithms for the TSP

The goal in the TSP is to find a shortest Hamiltonian circuit through a given set of n points. For our experimental comparison we have chosen four algorithms which showed very good performance and reimplemented them around a given `3-opt` algorithm. In particular, we implemented two GAs, analogous to the GA developed by Merz and Freisleben [7, 19] and one close to that of Walters [28], an ACO algorithm [4] by Stützle and Hoos [27] and an ILS algorithm [18, 17, 24]. Certainly, other previously mentioned approaches which showed very good performance would also be worthwhile investigating. Yet, due to the considerable effort of re-implementing the algorithms we had to limit the number of algorithms compared. It is important to note that the algorithms we applied are not exact reimplementations of the original algorithms. Nevertheless, the reimplementations capture the main characteristics of these algorithms and some of them use additional diversification mechanisms, which improved performance for larger run-times but were not present in the original algorithm descriptions. In some preliminary runs we tuned each of the algorithms to achieve very good performance without removing essential parts of the original algorithm ideas.

2.1 Genetic algorithm using DPX-crossover

The genetic algorithm (GA) by Merz and Freisleben (in the following called GA-DPX) [7, 19] uses a specific crossover operator called DPX which attempts to generate an offspring that has equal distance to both of its parents. DPX works as follows: the content of the first parent is copied to the offspring and all edges which are not in common in the two parents are deleted. The resulting parts of the broken tour are reconnected using a greedy heuristic but taking care not to use non-shared edges of the parents. For the mutation in GA-DPX we use double-bridge (DB) moves [18, 17, 12]. The DB mutation cuts the current tour into four sub-tours $s_1 - s_2 - s_3 - s_4$ (contained in the solution in that order) and reconnects them in the order $s_4 - s_3 - s_2 - s_1$. While Merz and Freisleben use a fine-tuned Lin-Kernighan local search to improve solutions, here we use "only" a `3-opt` local search. This choice may lead to some loss of peak performance; nevertheless, our implementation appears to incur only a slight loss of solution quality.

An additional feature of GA-DPX is that it uses a rather small population. To avoid stagnation of the search we added a diversification mechanism: If in $n/5$ consecutive iterations since the last diversification no improved solution is found, we apply to each solution of the current population 10 random double-bridge moves and continue the search. After each diversification the best solution found since the start of the trial is reinserted into the current population if $0.9 \cdot n/5$ iterations no improved solution is found to refocus the search around this best found solution.

Our algorithm follows roughly the algorithm scheme presented in [7]: a child is generated by applying the DPX-crossover to two randomly chosen parents (in each generation 10 offspring are generated), the child is locally optimized, mutated with a probability of 0.2 (if the child is mutated, again local search is applied to it) and then it replaces an individual in the current population. The population size is chosen as 20. Note that in [19] GA-DPX follows a slightly different outline, but we found the one used here to achieve, in general, better performance.

2.2 Repair-based GA by Walters

The GA by Walters differs substantially from GA-DPX by the fact that in Walters GA the crossover and the mutation may generate infeasible tours which are then repaired by an ingenious repair mechanism; we call our GA using this repair idea GA-Repair. The main idea of the repair algorithm is to replace infeasible edges by edges with a length as close as possible to the original length of the infeasible edge. We refer to [28] for details. An additional difference to GA-DPX is that GA-Repair uses a relatively large population. Additionally, Walters uses a brood selection mechanism [28] which generates several children for a pair of parents and then chooses only the best one or two children as offsprings. We did not implement exactly this mechanism, but in a similar spirit we generated a large number of offspring (three times the population size *m* which is set to $m = 100$ as in [28]) in each generation, choosing parents using a fitness based ranking which gives the fittest solution a three times higher probability to be chosen for the crossover than the worst one. Each offspring is mutated with a probability of $p = 0.5$ by applying two random `2-opt` moves and is then locally optimized. The new population is then composed of the *popsize* best solutions avoiding duplicate solutions. Additionally, we also introduced a diversification mechanism similar to that applied in GA-DPX, yet, without reintroducing the best solution since the start of the trial.

2.3 Ant colony optimization

In ACO algorithms a colony of artificial ants iteratively constructs solutions to the problem to be solved and communicates indirectly by (artificial) pheromone trails which, in the TSP application, are associated with the edges. The ants construct, starting at some initial city, tours by iteratively choosing from the current city i a next city j with a probability proportional to $\tau_{ij} \cdot \eta_{ij}^{\beta}$. For many problems, best performance is achieved if the ants' solutions, once they are completed, are improved by a local search algorithm. Here, we applied $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) which has been shown the be one of the best performing ACO algorithm for the TSP [24, 27, 25]. In $\mathcal{MM}$AS features of search diversification like pheromone trail re-initialization etc. are used, similar in spirit to the ones we used in GA-DPX; we refer to [27] for more details.

In $\mathcal{MM}$AS we use a colony of 25 ants, $\beta = 2$, and the next city is chosen among a candidate set of the 20 nearest cities. The pheromone trails are reinitialized if in $\max\{100, n/10\}$ consecutive iterations no improved solution is found (for more details see [27]).

2.4 Iterated local search

Iterative improvement algorithms are easily trapped in local minima. The basic idea of ILS is to modify the current solution s applying a *kick-move*, moving it to a point s' beyond the neighborhood searched by the local search algorithm and to continue the local search from s'. Typically, the solution modified in each iteration is the best solution found so far [17, 12]. ILS algorithms and, in particular the iterated Lin-Kernighan version of ILS, are known to be among the best performing algorithms for the TSP [17, 12].

As in most other ILS algorithms for the TSP we apply the double-bridge (DB) kick-move where the cutpoints are chosen randomly within a window of length $\max\{n/2, 500\}$ of the current tour (the same mutation was used in GA-DPX). One problem associated with applying the DB move only to the best solution is that ILS may easily get trapped in specific regions of the search space [24, 26]. One simple possibility to avoid this behavior is to restart ILS from scratch if no improved solution is found for a given number of iterations (ILS-Restart), here chosen as n iterations. Yet, here we apply a more sophisticated mechanism, which is called Fitness-Distance-Based Diversification (ILS-FDD) in [24, 26], to diversify the search. ILS-FDD generates a new starting solution for ILS in an iterative way; in particular, it obtains such a solution by first generating a set of candidate solutions by the standard ILS mechanism and then choosing one candidate solutions which is of high quality as well as rather distant from the current solution. This process is repeated until a solution at a given minimal distance to the current solution is obtained (see [24, 26] for more details).

2.5 The `3-opt` implementation

The `3-opt` implementation we use is based on a code provided by Tim Walters [28]. The `3-opt` algorithm uses the following standard speed-up techniques: It (i) restricts the set of moves which are examined to those contained in a candidate list of the 40 nearest neighbors [2, 16, 23]; (ii) it performs a fixed radius nearest neighbor search [2] and uses *don't look bits* associated with each node [2]. Before applying `3-opt` we set all don't look bits off, that is all cities are considered as a starting city to find an improving move. The `3-opt` algorithm is a first-improvement local search which gives preference to applying `3-opt` moves before the `2-opt` moves which are also checked in a `3-opt` implementation. The `3-opt` implementation takes care that the exchange of partial tours in a move is restricted to the smallest possible part to avoid the cost associated with copying arrays; the distances between all cities are kept in a distance matrix.

3 Experimental setup

We compare the algorithms on a large set of symmetric, Euclidean TSP instances with known optimal solution from TSPLIB, accessible at http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95, ranging in size from 198 cities up to 2392 cities. The number in the instance identifier is the problem size, for example, instance `rat783` has 783 cities. As a stopping criterion we used a CPU-time limit which was chosen large enough such that each of the algorithms could reach limiting behavior and an additional significant increase in solution quality would require a considerably increased CPU-time. In the experiments we noted that all the algorithms could find optimal solution for instances with up to 1000 cities in almost all trials. For these instances we compare the algorithms using run-time distributions (RTDs) [9] which give the cumulative empirical probability of finding an optimal solution as a function of the run-time. The RTDs are based on 25 independent trials of each algorithm. For instances with more than 1000 cities, 10 trials were done if not indicated otherwise. On these larger instances we present standard summary data due to the small number of trials.

To eliminate some factors which might affect final performance, all algorithms were coded in "C" and, where possible, they used similar data structures (for example, tour representation by arrays, same local search algorithm etc.) and were compiled and run on the same machine, a double processor Pentium III 450MHz machine running Redhat Linux 6.1 with 256 MB RAM. Due to the sequential implementation of the algorithms only one single processor is used.

4 Experimental results

4.1 Experimental comparison with run-time distributions

Among the instances in TSPLIB with $n \leq 1000$ we have chosen those proposed in the First and the Second Contest on Evolutionary Optimization [3]. Of these instances we do not present results for `eil51` and `kroA100` because they were often solved in the initialization phase of the algorithms. Additionally, we run for each algorithm 25 trials on the instances `pr1002`, `pcb1173`, and `d1291`.

Figure 1 gives the resulting RTDs. For most instances, ILS-FDD shows best performance which can be noted at the fact that for the same computation time it reaches a higher probability of finding optimal solutions than the competitors and on some instances (`att532` and `d1291`) it is the only algorithm which always finds optimal solutions in the given run-time. Yet, in general the relative ranking of the algorithms is strongly instance dependent. For example, GA-Repair is the worst performing algorithm on instance `pcb442`, but among the best performing ones on instances `rat783` and `pr1002`. Additionally, it can be observed that $\mathcal{MM}$AS has particular problems of finding optimal solutions on instance `pr1002` which, in contrast, is solved easily by the other algorithms.

An interesting observation from the RTDs is that the algorithms need an algorithm and instance specific initialization time t_{init} before they find the first optimal solution. t_{init} appears to be particularly large for GA-Repair and $\mathcal{MM}$AS. In GA-Repair this effect is mainly due to the large population size compared to GA-DPX or even ILS-FDD, which uses only one single solution; $\mathcal{MM}$AS has been specifically designed to be an ACO algorithm with a strong initial exploration of the search space, explaining this behavior.

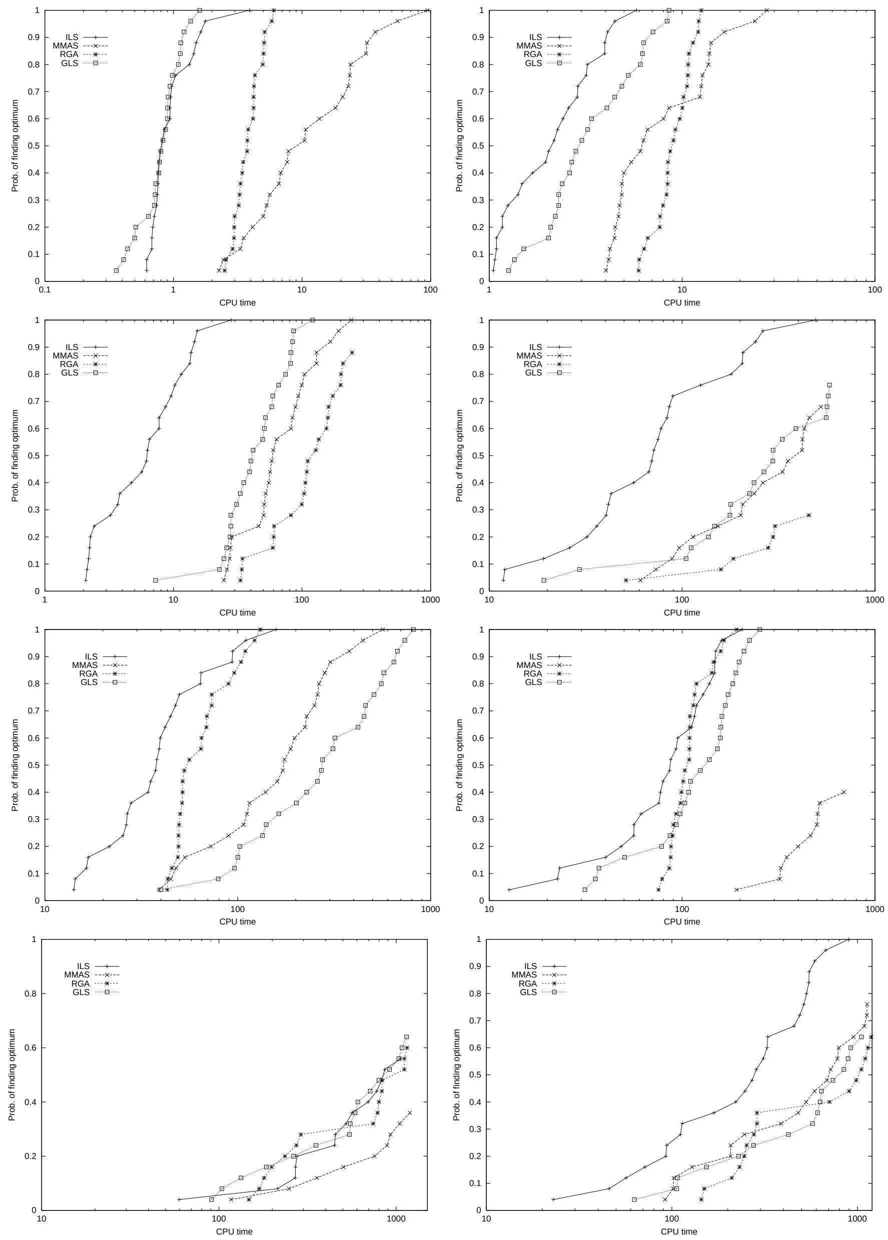

Fig. 1. Comparison of the algorithms using run-time distributions which give the cumulative empirical probability of finding optimal solutions on the TSP instances `d198` (upper left), `lin318` (upper right), `pcb442` (second row, left), `att532` (second row, right), `rat783` (third row left), `pr1002` (third row, right), `pcb1173` (fourth row, left), and `d1291` (fourth row, right). On each instance 25 independent trails have been run.

Yet, once this initialization time is passed, within some additional computation time the algorithms find a large number of additional optimal solutions. This effect is more notable on the smaller instances with a few hundreds of cities, while on most larger instances it is significantly harder to find optimal solutions and the variance of the solution times appear to increase. For example, on instance `lin318` GA-Repair finds the first optimal solution after 5.9 seconds and in the longest of the 25 trials it takes 12.6 seconds to find the optimal solution. An interesting question regarding t_{init} is its scaling behavior. This behavior has direct consequences on the usefulness of algorithm restarts (which would be the case for ILS-Restart) when compared to other diversification mechanisms which avoid such algorithm restarts (this is the case in ILS-FDD).

4.2 Experimental comparison on large instances

To obtain a more complete picture of the algorithms' performance we extend our experiments to a larger set of TSPLIB instances comprising all Euclidean TSPLIB instances with $1000 \leq n \leq 2392$ cities. On most instances 10 trials are run; the experimental results are given in Table 1. In general, for all algorithms very good performance is obtained and all algorithms found, with only few exceptions, on average solutions within only few tenths percent of the optimal solution. The computational results on these larger instances also confirm the findings in the previous section. On most instances ILS-FDD gives the best average performance and finds the largest number of optimal solutions. Yet, in general, which of the algorithms shows best or second best performance is again strongly instance dependent. Noteworthy is the very good performance of GA-Repair on two of the largest instances, `d2103` and `pr2392`, where it was the only algorithm which was able to find in all runs optimal solutions. Yet, on the two instances `u2152` and `u2319` both GA show rather poor results, which may be rather due to some particularities of these instances than because of a loss in the performance of the two GAs. Regarding the influence of the instance characteristics on the performance we noticed that ILS-FDD compared very favorable to the other algorithms on clustered instances. Extreme examples of such clusters of cities are `fl1400` and `fl1577`, but to some less extent such clusters are also available on some other instances. Yet, many other factors like the number of optimal solutions etc. may be responsible for the performance differences.

The surprisingly good performance of ILS-FDD is certainly due to the diversification features of ILS-FDD (note that the diversification mechanism applied in the two GAs and the ACO algorithms also improve their performance on most instances). If instead we apply ILS without any such features (called ILS-standard), that is if the DB kick-move is always applied to the best solution found since the start of the trial, in the same computation time ILS-standard could find only 9 optimal solutions on `pr1002`, 3 on `pcb1173`, 6 on `d1291`, 1 on `fl1577` and none on `pr2392` and the average percentage deviation increased to 0.15%, 0.11%, 0.082%, 0.47%, and 0.21%, respectively. Yet, already by the very simple ILS-Restart extension, a much improved performance over ILS-standard can be obtained finding 25, 2, 22, 4, and 1 optimal solutions, respectively and reducing the average percentage deviation to 0.0%, 0.015%, 0.0035%, 0.0054% and 0.11%, respectively. In general we noticed that the advantage of ILS-FDD over ILS-Restart strongly increases with instance size, certainly also due to the large initialization time for larger instances. Additionally, also the population-based algorithms compare much more favorably to ILS-Restart: on most instances at least one of the three population-based algorithms performs

Table 1. Comparison of the algorithms on a large set of TSPLIB instances with $1000 \le n \le 2392$. For each instance we report how often the optimal solution is found in a given number of trials (n_{opt}/No. of trials), the average percentage deviation from the optimum (Δ_{avg}), and the average CPU-time t_{avg} to find the best solution in a trial. The maximally allowed CPU-time for each trial has been limited to 1,200 sec. for $1,000 \le n \le 1,500$, 2,400 sec. for $1,500 < n \le 2,000$, and 3,600 sec. for $n > 2,000$. Best results with respect to n_{opt} or Δ_{avg} are indicated in **boldface**, second best in *italics*.

Instance	DPX `3-opt`			Repair			$\mathcal{MM}$AS			ILS-FDD		
	n_{opt}	Δ_{avg}	t_{avg}	n_{opt}	Δ_{avg}	t_{avg}	n_{opt}	Δ_{avg}	t_{avg}	n_{opt}	Δ_{avg}	t_{avg}
`dsj1000`	0/10	**0.0076**	607.7	0/10	0.63	772.8	0/10	0.16	826.6	0/10	*0.0091*	604.2
`pr1002`	**25/25**	**0.0**	132.8	**25/25**	**0.0**	111.51	10/25	0.021	509.1	**25/25**	**0.0**	93.5
`u1060`	0/10	0.031	662.5	0/10	*0.012*	500.4	0/10	0.13	968.0	**4/10**	**0.009**	671.8
`vm1084`	0/10	0.020	225.9	0/10	0.045	708.2	*1/10*	**0.011**	730.1	**4/10**	**0.011**	340.8
`pcb1173`	**16/25**	0.015	571.4	*15/25*	*0.0032*	524.3	9/25	0.0040	522.5	14/25	**0.0011**	493.8
`d1291`	16/25	0.034	396.1	16/25	0.021	535.7	*19/25*	*0.006*	527.2	**25/25**	**0.0**	320.5
`rl1304`	**10/10**	**0.0**	45.5	**10/10**	**0.0**	223.9	**10/10**	**0.0**	161.7	**10/10**	**0.0**	23.6
`rl1323`	*1/10*	*0.009*	104.5	0/10	0.010	283.0	*1/10*	0.017	679.3	**10/10**	**0.0**	148.3
`nrw1379`	0/10	*0.067*	744.3	0/10	0.087	686.1	0/10	0.14	912.7	0/10	**0.051**	867.3
`fl1400`	0/10	0.12	787.9	*1/10*	*0.040*	677.8	0/10	0.53	640.9	**10/10**	**0.0**	218.2
`u1432`	0/10	0.45	787.1	0/10	*0.099*	742.5	0/10	0.25	640.9	**4/10**	**0.039**	763.4
`fl1577`	0/10	0.16	1283.4	0/10	*0.089*	1430.0	0/10	0.30	1716.3	**10/10**	**0.0**	648.5
`d1655`	*9/10*	*0.0003*	809.6	*9/10*	0.0008	999.7	1/10	0.049	1314.7	**10/10**	**0.0**	803.1
`vm1748`	*2/10*	*0.045*	713.8	0/10	0.11	1892.6	0/10	0.090	1472.9	**3/10**	**0.031**	1267.4
`u1817`	0/10	0.14	1470.6	0/10	*0.093*	1637.9	0/10	0.099	1267.8	0/10	**0.056**	1316.3
`rl1889`	*7/10*	*0.041*	1102.8	0/10	0.17	1742.2	0/10	0.17	1401.2	**8/10**	**0.0064**	1101.4
`d2103`	6/10	0.016	1476.8	**10/10**	**0.0**	1085.8	0/10	0.045	2487.9	*8/10*	*0.001*	1747.7
`u2152`	0/10	0.18	2149.7	0/10	0.18	2385.6	0/10	*0.16*	2494.6	0/10	**0.061**	2401.9
`u2319`	0/10	1.24	2322.3	0/10	0.59	2231.5	0/10	*0.29*	1387.6	0/10	**0.11**	1233.8
`pr2392`	*4/10*	0.075	842.9	**10/10**	**0.0**	572.7	0/10	0.16	1946.2	*4/10*	*0.004*	2079.2

better than ILS-Restart.

One last concern is how our computational results compare to the previously published results of the original algorithms. The performance of $\mathcal{MM}$AS is, except on instance `fl1577` significantly better than those reported with `3-opt` local search in [27] which is mainly due to the better performance of the `3-opt` applied here. For ILS-FDD the results in [26] appear, when adjusting for the different computer speed, to be slightly better than those reported on the larger instances, while they are worse on the smallest five instances. Again, this is probably due to slight differences of the `3-opt` implementation and other implementation details. Also our GA-DPX implementation has shown very good performance. It appears to perform roughly comparable on the instances with $n \le 1000$ to the results published in [19]; better results are reported by the same authors for a somewhat modified MA in [20]. Yet, with increasing instance size our GA-DPX appears to perform slightly worse which may be due to the more powerful Lin-Kernighan local search algorithm applied in [19, 20] (yet, a detailed comparison is not possible here because the authors report only results for two instances with $n > 1000$). Regarding GA-Repair, our implementation performs comparable to the implementation of [28] with the exception of instance `fl1577`, where the computational results reported in [28] are significantly better. Our implementation achieves somewhat better results on other instances like, for example, `rat783` (we found optimal solutions in every run, while in [28] only a solution probability of 0.8 was reached).

5 Conclusions

In this paper we have experimentally compared some of the best performing algorithms for the TSP. Because the particular local search algorithm chosen has a very significant influence on the final performance, we re-implemented two GAs, one ACO algorithm, and one ILS algorithm, around a same `3-opt` algorithm. For all algorithms very good computational results could be obtained, sometimes even better than in the original publications. We found ILS-FDD to be the best performing algorithm on most instances. This is quite remarkable, because typically iterated local search algorithms are relatively easy to implement and have less parameters to be tuned than other, more complex algorithms. This fact also suggests that a population of solutions is not necessary to obtain a very high performing algorithm for the TSP. Nevertheless, it should be noted that the ILS-FDD variant uses some extra features like a specific diversification mechanism. More straightforward ILS extensions like ILS-Restart still show very good performance, but they compare less favorable to the three population-based approaches. Hence, one interesting conclusion is that if algorithms which manipulate only a single solution include appropriate diversification mechanisms, they may be very competitive with high performing population-based algorithms. This observation raises interesting research issues. One such issue is, how the relative performance of the algorithms applied here differs if weaker (for example, `2-opt`) or stronger (for example, the Lin-Kernighan heuristic) local search algorithms are applied. Another issue is on which types of practically relevant problems population-based algorithms achieve more advantage over algorithms modifying single solutions.

Acknowledgements

We would like to thank Tim Walters for making available a version of his `3-opt` and remarks on this work and Peter Merz for comments on the paper.

References

1. E. H. L. Aarts and J. Korst. *Simulated Annealing and Boltzman Machines*. John Wiley & Sons, Chichester, 1989.
2. J. L. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing*, 4(4):387–411, 1992.
3. H. Bersini, M. Dorigo, S. Langerman, G. Seront, and L. Gambardella. Results of the first international contest on evolutionary optimisation. In *Proc. of ICEC'96*, pages 611–615, 1996.
4. M. Dorigo and G. Di Caro. The Ant Colony Optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw Hill, 1999.
5. M. Dorigo and L. M. Gambardella. Ant Colony System: A cooperative learning approach to the traveling salesman problem. *IEEE Trans. on Evolutionary Computation*, 1:53–66, 1997.
6. D. B. Fogel. Applying evolutionary programming to selected travelling salesman problems. *Cybernetics and Systems*, 24:27–36, 1993.

7. B. Freisleben and P. Merz. New genetic local search operators for the traveling salesman problem. In *Proc. of PPSN-IV*, volume 1141 of *LNCS*, pages 890–900. Springer, 1996.
8. M. Gorges-Schleuter. Asparagos96 and the travelling salesman problem. In *Proc. of ICEC'97*, pages 171–174, 1997.
9. H. H. Hoos and T. Stützle. Evaluating Las Vegas algorithms — pitfalls and remedies. In *Proc. of the 14th Conference on Uncertainty in AI*, pages 238–245. Morgan Kaufmann, 1998.
10. J. J. Hopfield and D. Tank. Neural computations of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.
11. J. Houdayer and O. C. Martin. Renormalization for discrete optimization. *Physical Review Letters*, 83(5):1030–1033, 1999.
12. D. S. Johnson and L. A. McGeoch. The travelling salesman problem: A case study in local optimization. In E.H.L. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, Chichester, England, 1997.
13. K. Katayama and H. Narihisa. Iterated local search approach using genetic transformation to the traveling salesman problem. In *Proc. of GECCO'99*, pages 321–328. Morgan Kaufmann, 1999.
14. S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
15. J. Lee and M. Y. Choi. Optimization by multicanonical annealing and the traveling salesman problem. *Physical Review E*, 50(2):651–654, 1994.
16. S. Lin and B. W. Kernighan. An effective heuristic algorithm for the travelling salesman problem. *Operations Research*, 21:498–516, 1973.
17. O. Martin and S. W. Otto. Combining simulated annealing with local search heuristics. *Annals of Operations Research*, 63:57–75, 1996.
18. O. Martin, S. W. Otto, and E. W. Felten. Large-step Markov chains for the traveling salesman problem. *Complex Systems*, 5(3):299–326, 1991.
19. P. Merz and B. Freisleben. Genetic local search for the TSP: New results. In *Proc. of ICEC'97*, pages 159–164. IEEE Press, 1997.
20. P. Merz and B. Freisleben. Fitness landscapes and memetic algorithm design. In D. Corne, M. Dorigo, and F.Glover, editors, *New Ideas in Optimization*, pages 244–260. McGraw Hill, 1999.
21. A. Möbius, B. Freisleben, P. Merz, and M. Schreiber. Combinatorial optimization by iterative partial transcription. *Physical Review E*, 59(4):4667–4674, 1999.
22. Y. Nagata and S. Kobayashi. Edge assembly crossover: A high-power genetic algorithm for the traveling salesman problem. In *Proc. of ICGA'97*, pages 450–457. Morgan Kaufmann, 1997.
23. G. Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*, volume 840 of *LNCS*. Springer, 1994.
24. T. Stützle. *Local Search Algorithms for Combinatorial Problems — Analysis, Improvements, and New Applications*. PhD thesis, Darmstadt University of Technology, Department of Computer Science, 1998.
25. T. Stützle and M. Dorigo. ACO algorithms for the traveling salesman problem. In K. Miettinen et al., editor, *Evolutionary Algorithms in Engineering and Computer Science*, pages 163–183. Wiley, 1999.
26. T. Stützle and H. H. Hoos. Analyzing the run-time behaviour of iterated local search for the TSP. Technical Report IRIDIA-4-00, IRIDIA, Université Libre de Bruxelles, 2000.
27. T. Stützle and H. H. Hoos. $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System. *Future Generation Computer Systems*, 16(8):889–914, 2000.
28. T. Walters. Repair and brood selection in the traveling salesman problem. In *Proc. of PPSN-V*, volume 1498 of *LNCS*, pages 813–822. Springer, 1998.