COMBINING 2-OPT, 3-OPT AND 4-OPT WITH K-SWAP-KICK PERTURBATIONS FOR THE TRAVELING SALESMAN PROBLEM

Andrius Blazinskas, Alfonsas Misevicius

Kaunas University of Technology, Department of Multimedia Engineering, Studentu St. 50–401, 416a, Kaunas, Lithuania, andrius.blazinskas@ktu.lt, alfonsas.misevicius@ktu.lt

Abstract. The Traveling Salesman Problem (TSP) is a famous NP-hard problem typically solved using various heuristics. One of popular heuristics class is *k*-opt local search. Though these heuristics are quite simple, combined with other techniques in an iterated local search (ILS) framework they show promising results. In this paper, we propose to combine the 2-opt, 3-opt and 4-opt local search algorithms with so called *k-swap-kick* perturbations, which are a generalization of the well known *double-bridge* (random 4-opt) move. To reduce the run time of our 2-opt, 3-opt and 4-opt implementations, we apply some enhancements like a candidate list (based on *k-d* tree), search cuts, greedy starts, *two-level* tree data structure and others. We provide experimental results of the implemented algorithms with the TSPLIB instances.

Keywords: traveling salesman problem, 2-opt, 3-opt, 4-opt, k-swap-kick.

1 Introduction

The traveling salesman problem (TSP) [6] can be formulated as follows. Given a set of cities and distances between them the task is to find shortest tour (Hamiltonian cycle) visiting every city only once. The problem where distance between two cities does not depend on the direction is called symmetric TSP (STSP) and asymmetric TSP (ATSP) otherwise. TSP is considered to be NP-hard problem. The total number of possible tours for STSP is equal to $(n-1)!/2$ (where n is the problem size – the number of cities (nodes)) [7]. It is a relatively complicated task to find good tours in an acceptable time for such a search space, especially when n is large.

There are two main ways of solving the TSP: exact methods and heuristics. The exact methods are too time-consuming for larger n, thus heuristics typically are used. One of the leading heuristics for the STSP is *Lin-Kernighan* [9]. Effective implementations of it exist (see, for example, *Concorde*[*] [2], Helsgaun's code[**] (LKH) [7]).

In this paper, we consider only STSP. In the experiments, we use *k*-opt heuristics, which are closely related to a more robust *Lin-Kernighan* heuristic, since both use *k*-opt sub-moves [8]. In Section 2, we describe several ways how to optimize 2-opt, 3-opt and 4-opt heuristics to make them run faster. In Section 3, *k-swap-kick* perturbations are described and in Section 4 experimental results for combining 2-opt, 3-opt and 4-opt heuristics with *k-swap-kick* perturbations are presented. We also provide conclusions and future research ideas in Section 5.

2 Improving 2-opt, 3-opt and 4-opt speed

While 2-opt algorithm is simple and its naive form involves repeated breaking of two edges and reconnecting them in other (cost decreasing) way (see Figure 3b) until no positive gain 2-opt move can be made, when it comes to practice, there is more specifics to consider and different authors implement it differentially. For example, some authors check all possible flips and perform only the best one [11], while others simply make the first found positive gain flip (see *DIMACS Implementation Challenge for STSP*[***] on variations). Several other possible implementation choices for 2-opt heuristic are also indicated in [3]. Such choices typically significantly impact the cost and timings. Most of these considerations also apply for 3-opt and 4-opt, but even more cases and specifics needs to be evaluated.

The time complexity for naive 2-opt, 3-opt and 4-opt is $O(n^2)$, $O(n^3)$ and $O(n^4)$ respectively, however this can be greatly improved by using various speedup techniques. These techniques typically slightly sacrifice true local optimality (resulting tours are not really *k*-optimal), but in some cases combination of them allows reaching nearly $O(n)$ complexity [1].

We use several important improvements for our fast 2-opt, 3-opt and 4-opt modifications (2-opt-f, 3-opt-f and 4-opt-f). We are outlining them shortly in the subsequent subsections.

---

[*] *Concorde TSP solver*, http://www.tsp.gatech.edu/concorde/

[**] *Helsgaun's Lin-Kernighan implementation*, http://www.akira.ruc.dk/~keld/research/LKH/

[***] *DIMACS Implementation Challenge for STSP*, http://www2.research.att.com/~dsj/chtsp/

2.1 *K-d* tree based candidate list

Nearest node candidate lists (*CL*) are widely used for improving *k*-opt heuristics [1]. *K-d* tree usage for *CL* identification is much more efficient than naive *CL* approach. In particular, naive *CL* can be generated in $O(n^2 \log n)$ time [11], while *k-d* tree – in $O(Kn + n\log n)$ [4] (where *K* is the number of dimensions and in our case always $K = 2$). Once generated, naive *CL* does not require any additional processing to extract lists (it simply contains them in arrays), while *k-d* tree needs to be searched every time a list is needed. This, of course, can be solved by using caching, but for large TSP instances caching requires a lot of memory which grows in $O(mn)$ (where m – *CL* size) and for storing *k-d* tree this number is roughly $O(n)$ (efficient *Concorde k-d* tree implementation uses $52n$ bytes of memory). Searching in *k-d* tree (*bottom-up* technique) can be done in almost $O(1)$ time [4]. Thus efficient *k-d* tree implementation beats naive *CL* practically in all cases. Benefits of using *k-d* tree for *CL* are obvious, for example, on our test machine (see Section 4 for specifications) for *pla85900* TSP problem instance construction of *CL* in naive way takes roughly 15 minutes, while construction of *k-d* tree with searching and caching of *CL* – about 3 seconds (*CL* size in both cases is equal to 80). It must be noted, that usage of *k-d tree* limits TSP solver to Euclidean instances.

2.2 Stop search if $b_1 = a_2$

Instead of a popular *don't look bits* idea [1], we propose using search cut if a_1 nearest candidate b_1 is already the next node to a_1 (that is, $a_2 = b_1$) in the current tour (see Figure 2 for pseudo code). A similar cut is used for 2-opt *fixed-radius* search in [3]. We also use such cuts for 3-opt and 4-opt when $b_2 = c_1$ and $c_2 = d_1$. Our experiments indicate that these improvements greatly shorten running times without significantly increasing cost.

2.3 For 3-opt and 4-opt consider cases where only 3 and 4 edges are exchanged respectively

While breaking *k* edges in a tour, there are $(k-1)!2^{k-1}$ ways to reconnect it (including the initial tour) to form a valid tour [6], typically not all of these cases are considered [15]. In our implementation, we consider only those cases where all *k* edges are new. For example, breaking 3 edges in 3-opt there are in total 8 cases for reconnection (Figure 1), but only 4 of them (e, f, g, h) actually introduce all new edges (all others, except initial one, are simple 2-opt moves). Similarly for 4-opt, breaking 4 edges we have 48 ways to reconnect, but only 25 cases offer all 4 new edges. This does not affect 2-opt heuristic, since there is only one way to reconnect to a valid tour.

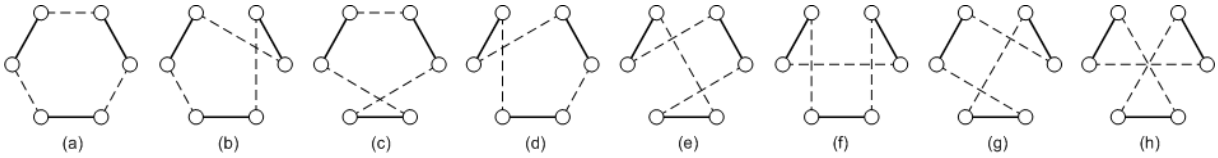

Figure 1. All possible 3-opt reconnection cases

2.4 Cascading 3-opt and 4-opt

We have noticed that cascading *k*-opt heuristics reduces overall running time. We exploit this feature in this way: for 3-opt-f, we additionally preprocess tours with 2-opt-f and for 4-opt-f preprocessing is done with 2-opt-f and 3-opt-f in this order. Optimization is started from random node every time.

2.5 Greedy multi-fragment initial tours

Multi-fragment heuristic (simply called *Greedy* heuristic) is an effective tour construction heuristic proposed in [3]. It is known to be particularly good as a preprocessor for *k*-opt heuristics and more effective than traditional *nearest-neighbour* heuristic [1, 3]. Using such greedy starting tours in *k*-opt, improvement is observed for both – tour quality and running times. In our implementation, *multi-fragment* tour construction is backed with earlier described *k-d* tree for efficiency reasons. Complexity for this heuristic is $O(n\log n)$ [3]. It should be noted, that it makes sense to use this heuristic only once for particular problem, since there is no different starting points like with the *nearest-neighbor* and thus the resulting tour is always the same.

2.6 *Two-level* tree data structure

Two-level tree [5] is one of the most popular data structures for tour representation in the TSP [1, 7]. It offers $O(\sqrt{N})$ complexity for reversing (flipping) the sub-tour – one of the most common operations in the TSP. It is useful for problems where $n \leq 10^6$, otherwise *Splay* trees seems to be a better choice [5]. It should be noted, that usage of *two-level* tree is helpful only within the optimized *k*-opt heuristics, where time required for flip operation becomes dominant. In our experiments for naive 2-opt, only changing array with *two-level* tree resulted in higher running times: traversing through nodes in *two-level* tree is more costly, since it requires accessing parent nodes and performing additional checks (though still $O(1)$, the constant is actually larger), while

for simple array representation it is only a matter of increasing or decreasing the array index. Considering above mentioned improvements, a rough pseudo code for the fast 4-opt implementation is provided in Figure 2.

```
procedure fast4Opt(tour, cl, costs, startNode) begin
localyOptimal := FALSE;
while (not localyOptimal) begin
        localyOptimal := TRUE;
        a1 := startNode;
        repeat
                a2 := next(a1);
                for (∀b1 ∈ cl(a1)) begin
                        if (a2 == b1) break; // speedup
                        b2 := next(b1);
                        for (∀c1 ∈ cl(b1)) begin
                                if (b2 == c1) break; // speedup
                                c2 := next(c1);
                                for (∀d1 ∈ cl(c1)) begin
                                        if (c2 == d1) break; // speedup
                                        d2 := next(d1);
                                        findBestReconnection(a1, a2, b1, b2, c1, c2, d1, d2, costs);
                                        if (better reconnection exists) begin
                                                reconnect(a1, a2, b1, b2, c1, c2, d1, d2, tour);
                                                localyOptimal := false;
                                        end
                                end
                        end
                end
                a1 := next(a1);
        until (a1 == startNode);
end
end fast4Opt
```

Figure 2. General pseudo code for fast 4-opt implementation

3 K-swap-kick perturbation

The simplest non-sequential *k*-opt move is a *double-bridge* move (4-opt move) [8] first mentioned in [9] and later recognized as an effective mutation operator [10]. It is well known move, random version of it is used in one of the most powerful heuristics – *Iterated Lin-Kernighan* [1]. A generalization of this move is a *k-swap-kick* (a concatenation of *k* segments: s_1, s_k, s_{k-1},...,s_2) introduced in [14] as an improvement for LKH. Because LKH considered improving *k*-opt moves of size $k \leq 5$, it was proposed to use *k-swap-kicks* of size $k \geq 6$. Since we will be applying *k-swap-kicks* to our fast 2-opt, 3-opt and 4-opt heuristics, we will omit this restriction. We will consider all same pattern having moves with $k \geq 3$ and simple 2-*opt* move ($k = 2$). Graphical 2-*opt* move and *k-swap-kicks* illustration for $k = 3, 4$ is provided in Figure 3.

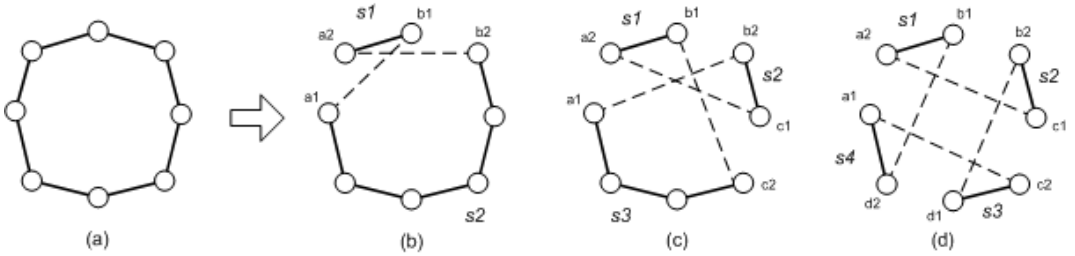

Figure 3. 2-opt move and *k-swap-kick* perturbations: a) initial tour, b) 2-opt move, c) 3-opt move (k=3) and d) *double-bridge* or 4-opt move ($k = 4$)

All *k-swap-kicks* are special cases of *k*-opt moves. Clearly for *k-swap-kick* to be performed, a condition of $n \geq 2k$ is enough.

We will be analyzing only random 2-opt and *k-swap-kick* moves, the ones which do not consider the effect on tour cost during perturbation and may typically lead to worse tours. This is opposite to the moves used in *k*-opt heuristics, which typically perform only tour-cost-decreasing moves.

Random *double-bridge* move previously was also combined with 3-opt heuristic, forming *Iterated 3-opt* [1]. In some cases, it even yielded better results to that of *Lin-Kernighan* and was slightly worse than *Iterated Lin-Kernighan*. We continue by extending this idea. We perform an experiment combining our 2-opt-f, 3-opt-f and

4-opt-f implementations with random 2-opt move and random *k-swap-kicks* (k = 3...15). Pseudo code on how this combination is implemented is provided in Figure 4b.

```
z* := ∞;
repeat
    Generate a greedy randomized solution t;
    Find local optimum t_l with local search starting from t;
    if (z(t_l) < z*) begin
        z* := z(t_l);
        t* := t_l;
    end
until a stopping criterion is satisfied;
```

(a)

```
Generate a greedy solution t;
Find local optimum t_l with local search starting from t;
z* := z(t_l);
repeat
    Perturb t_l tour using random k-swap-kick forming t_lp;
    Find local optimum t_lpl with local search starting from t_lp;
    if (z(t_lpl) < z*) begin
        z* := z(t_lpl);
        t* := t_l := t_lpl;
    end
until a stopping criterion is satisfied;
```

(b)

Figure 4. General pseudo codes for GRASP (a) and *k-swap-kick* perturbation (b) frameworks (*t – best solution found and *z** = z(*t**), where *z* – objective function)**

It should be noted that *k-swap-kick* ILS framework is quite similar to other well known procedure – GRASP (see Figure 4a, also see [12]). The main distinction between them is that GRASP starts from a totally different initial tour in every iteration, rather than perturbing existing one.

4 Experimental results

All algorithms are fully implemented using Java (JDK 6). Implementations of *two-level* tree, *k-d* tree and *multi-fragment* heuristic are based on *Concorde* C code. Experiments were performed on the machine with Intel Core i7-860 processor and 8GB of RAM.

Below provided tables and graphs summarize the results of our fast 2-opt, 3-opt and 4-opt implementations (2-opt-f, 3-opt-f and 4-opt-f). To emphasize the advantage of perturbations, we also performed two additional experiments without using perturbations: one with random starts and other with greedy starts. In these experiments, we used similar to GRASP procedure, but it did not fully comply with basic GRASP scheme (see Figure 4a): random start experiment did not use greedy randomized heuristic for initial solution and greedy start experiment had no randomization (since we were using original *multi-fragment* heuristic). The only randomization in later case was different start point (node) in fast *k*-opt local search. All other consecutive columns provide the results for random 2-opt (k = 2) and *k-swap-kick* perturbations with fixed k. In both frameworks, the number of iterations was set to 1000 (stopping criterion). Deviations from the optimal tour length (optimal tour lengths can be found in TSPLIB [13]) were estimated using formula: $\delta = 100(z^* - z_{opt})/z_{opt}\,[\%]$, where z^* – obtained best value of the objective function and z_{opt} denotes the provably optimal objective function value. All experiments were repeated for 10 times and averages calculated. The problem data for experiments were taken from the TSP library – TSPLIB [13].

Table 1. Experimental results for 4-opt-f: tour quality (average deviation from optimal of 10 runs, %)

Problem name	4-opt-f		4-opt-f with greedy start and *k-swap-kicks*							
	Random start	Greedy start	*k*							Best found (*k*)
			2	3	4	5	6	10	15	
eil51	0.00	1.17	0.02	0.07	0.05	0.02	0.12	0.00	0.02	**0** (all)
st70	0.00	3.26	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**0** (all)
kroE100	0.03	0.00	0.00	0.00	0.03	0.02	0.00	0.00	0.02	**0** (all)
kroB150	0.19	2.11	0.11	0.02	0.04	0.04	0.02	0.03	0.03	**0** (all)
ts225	0.21	1.35	0.02	0.00	0.00	0.00	0.00	0.00	0.00	**0** (all)
gil262	1.73	2.11	0.40	0.46	0.13	0.14	0.23	0.20	0.27	**0** (4, 6)
a280	1.81	1.98	0.64	0.48	0.02	0.05	0.20	0.03	0.04	**0** (all)
lin318	1.66	2.09	0.54	0.49	0.36	0.42	0.35	0.37	0.54	**0** (2, 4)
rd400	2.42	2.10	0.57	0.33	0.24	0.33	0.29	0.51	0.54	**0.07** (4, 6)
u574	2.98	2.50	1.18	0.53	0.62	0.63	0.69	0.79	1.06	**0.13** (3)
rat783	3.96	2.58	1.43	1.00	1.11	1.13	1.19	1.55	1.78	**0.48** (3)
vm1084	3.05	1.92	0.87	0.47	0.47	0.60	0.50	0.77	0.98	**0.15** (3, 4)
pcb1173	4.92	3.78	2.10	1.76	1.45	1.77	1.82	2.15	2.54	**1.17** (4)
vm1748	3.68	2.89	1.24	0.81	0.85	0.96	1.08	1.23	1.52	**0.50** (4)
d2103	6.33	2.46	1.20	0.91	0.82	1.22	1.22	1.53	1.80	**0.27** (4)
fnl4461	4.96	2.57	2.21	2.19	2.17	2.44	2.47	2.76	2.74	**1.92** (2)
rl5934	5.71	2.95	2.06	1.82	1.81	2.21	2.08	2.69	3.07	**1.47** (3)

Problem name	4-opt-f		4-opt-f with greedy start and *k-swap-kicks*							
	Random start	Greedy start	*k*							Best found (*k*)
			2	3	4	5	6	10	15	
pla7397	4.85	2.94	2.01	1.74	1.62	2.04	1.86	2.28	2.71	**1.34** (4)
rl11849	6.30	2.85	2.52	2.58	2.52	2.82	2.88	3.14	3.13	**2.23** (4)
usa13509	5.11	3.07	2.61	2.72	2.59	2.98	2.93	3.21	3.19	**2.47** (2)
brd14051	5.30	3.03	2.77	2.89	2.94	3.07	3.09	3.06	3.27	**2.66** (2)
d15112	5.25	2.92	2.71	2.82	2.87	2.94	3.01	3.10	3.16	**2.58** (2)
d18512	5.37	2.84	2.68	2.82	2.84	2.94	2.94	3.04	3.07	**2.54** (2)

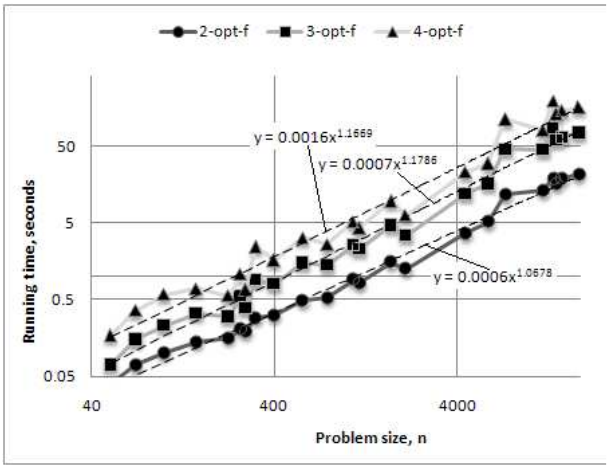

Figure 5. Running time dependence on problem size (*k-swap-kick* size = 4, logarithmic scale)

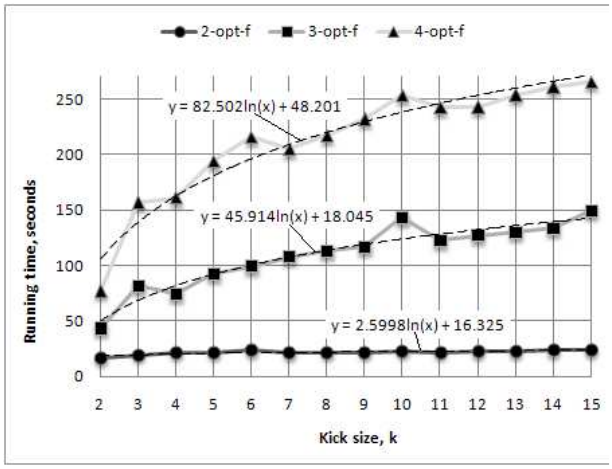

Figure 6. Running time dependence on *k-swap-kick* size (problem d18512)

Table 2. Experimental results for 4-opt-f: running time (average of 10 runs, seconds)

Problem name	4-opt-f		4-opt-f with greedy start and *k-swap-kicks*						
	Random start	Greedy start	*k*						
			2	3	4	5	6	10	15
eil51	0.51	0.33	0.07	0.16	0.17	0.2	0.21	0.29	0.32
st70	1.04	0.36	0.19	0.3	0.36	0.35	0.41	0.53	0.67
kroE100	1.98	0.86	0.27	0.46	0.58	0.6	0.76	0.93	1.26
kroB150	2.79	1.53	0.37	0.58	0.69	0.71	0.87	1.08	1.45
ts225	3.09	0.73	0.32	0.56	0.56	0.66	0.78	1.05	1.39
gil262	5.74	1.73	0.57	1.01	1.06	1.18	1.47	1.8	2.54
a280	5.15	1.73	0.37	0.65	0.67	0.73	0.98	1.33	1.82
lin318	10.46	4.52	1.3	2.26	2.45	2.71	3.17	4.63	5.58
rd400	9.52	3.37	0.87	1.65	1.63	1.89	2.28	3.35	4.07
u574	17.71	9.28	1.74	3.12	3.19	3.55	4.35	6.59	7.74
rat783	18.21	6.32	1.31	2.49	2.61	2.97	3.59	5.46	6.56
vm1084	31.18	16.08	2.76	5.26	5.2	5.98	7.07	10.72	12.57
pcb1173	34.78	13.31	2.38	4.5	4.31	5.32	6.39	9.8	12.19
vm1748	61.29	33.66	5.51	10.04	9.65	11.34	13.78	20.37	21.04
d2103	65.52	16.4	3.58	6.35	6.29	7.24	8.69	12.12	14.63
fnl4461	175.72	59.5	11.44	22.77	22.74	26.38	32.31	38.78	44.42
rl5934	238.15	88.53	16.11	31.66	29.63	37.66	42.11	56.06	67.38
pla7397	964.15	377.13	50.73	118.11	111.6	143.28	170.82	222.82	278.1
rl11849	716.1	241.73	39.93	86.49	80.5	101.43	119.85	149.98	172.37
usa13509	1197.66	619.04	89.98	186.62	195.33	222.92	269.8	318.19	356.73
brd14051	894.35	347.39	62.32	132.13	130.7	143.25	176.97	204.42	228.78
d15112	1018.65	474.92	70.16	144.26	144.54	163.4	195.59	233.7	264.27
d18512	1159.27	510.4	77.3	156.8	160.93	193.64	215.89	253.28	265.59

5 Conclusions and future research

In this paper, we have proposed efficient implementations of 2-opt, 3-opt and 4-opt heuristics for solving the traveling salesman problem. We were also concerned with combining these heuristics with the random 2-opt move and *k-swap-kick* perturbations. We provide the results of the experiments with these heuristic variants as well. To show efficiency of the perturbations-based approach, a simplified GRASP-like procedure was, in addition, used in the comparison of the heuristics.

As can be seen from the experimental results, the bigger problem size is – the less effect (in cost terms) perturbations have and simply increasing the perturbation size only increases cost and running time. Overall perturbation algorithm processing time dependence on problem size is approximately $O(n^{1.2})$, while dependence on kick size – logarithmic (see Figure 5 and Figure 6). Because of absence of randomness in *multi-fragment* heuristic, such greedy starts for *k*-opt heuristics, without using perturbations, are not so effective for smaller problem sizes. This can be seen on the instances with $n < 400$, where simple non-improved random starts give much better results, however going beyond that limit, greedy starts surpass.

Though, as expected, the *k-swap-kick* of size 4 (i.e. *double-bridge* move) proven to be one of the most effective perturbations, overall best solutions were also often found by using simple random 2-opt move and 3-opt move perturbations (see Table 1, last column). This proposes idea for further experiments, where *double-bridge* moves could be combined with 2-opt and 3-opt moves in some reasonable manner. Another interesting extension would be usage of *quadrant nearest neighbor CL* [6] instead of typical *CL*. However, this may not be effective with the cut optimization described in this paper (Section 2.2), so further improved approaches may be needed.

References

[1] **Aarts E., Lenstra J. K.** Local search in combinatorial optimization. *John Wiley & Sons, Inc*, 1997.

[2] **Applegate D. L., Bixby R. E., Chvátal V., Cook W. J.** The traveling salesman problem: A computational study, *Princeton University Press*, 2007.

[3] **Bentley J. L.** Fast Algorithms for Geometric Traveling Salesman Problems, *ORSA J. Comput.*, 1992, vol. 4-4, 387-411.

[4] **Bentley J. L.** *K-d* trees for Semidynamic Point Sets. *Proceedings of the 6th Annual ACM Symposium on Computational Geometry*, 1990, 187-197.

[5] **Fredman M. L., Johnson D. S., Mcgeoch L. A., Ostheimer G.** Data Structures for Traveling Salesmen. *Journal of Algorithms*, 1995, vol. 18-3, 432-479.

[6] **Gutin G., Punnen A. P.** The Traveling Salesman Problem and Its Variations, *Kluwer*, 2002.

[7] **Helsgaun K.** An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 2000, vol.126, 106-130.

[8] **Helsgaun K.** General *k*-opt submoves for the Lin-Kernighan TSP heuristic. *Mathematical Programming Computation*, 2009, 119-163.

[9] **Lin S., Kernighan B. W.** An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 1973, vol. 21, 498-516.

[10] **Martin O., Otto S. W., Felten E. W.** Large-Step Markov Chains for the Traveling Salesman Problem. *Complex Systems*, 1991.

[11] **Misevičius A., Ostreika A., Šimaitis A., Žilevičius V.** Improving Local Search for the Traveling Salesman Problem. *Information Technology And Control, Kaunas, Technologija*, 2007, Vol. 36, No. 2, 187 – 195.

[12] **Pitsoulis L. S. and Resende M. G. C.** Greedy randomized adaptive search procedures. In *P.M. Pardalos and M.G.C. Resende, editors, Handbook of Applied Optimization*, *Oxford University Press*, 2002, 178-183.

[13] **Reinelt G.** TSPLIB — A traveling salesman problem library. *ORSA Journal on Computing*, 1991, vol.3-4, 376-385. Access via the Internet:: <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/>.]

[14] **Richter D., Goldengorin B., Jager G., Molitor P.** Improving the Efficiency of Helsgaun's Lin-Kernighan Heuristic for the Symmetric TSP. *In Proceedings of the 4th conference on Combinatorial and algorithmic aspects of networking, 2007.*

[15] **Sierksma G.** Hamiltonicity and the 3-Opt procedure for the traveling salesman problem. *Applicationes Mathematicae*, 1994, vol. 22-3, 351–358.