Speeding-up the exploration of the 3-OPT neighborhood for the TSP

Speeding-up the exploration of the 3-OPT neighborhood for the TSP

Giuseppe Lancia* Marcello Dalpasso†

1 TSP and the K-OPT neighborhood

The Traveling Salesman Problem (TSP) calls for finding the shortest hamiltonian cycle (tour) in a complete graph $G = (V, E)$ of n nodes, weighted on the arcs. We consider the symmetric TSP, i.e., the graph is undirected and the distance $c(i, j)$ between two nodes i and j is the same irrespective of the direction in which we traverse an edge. A tour is identified by a permutation of vertices $(v_1, \ldots, v_n)$. The length of a tour T, denoted by $c(T)$ is the sum of the lengths of the edges of the tour. More generally, for any set F of edges, we denote by $c(F)$ the value $\sum_{e \in F} c(e)$.

Local search (LS) [6, 1] is often a very effective way to tackle hard combinatorial optimization problems, including the TSP. Assume the problem is $\min_{x \in X} f(x)$. Given a map which associates to every solution x a set $N(x)$ called its *neighborhood*, in LS we start at any solution x^0, set $s := x^0$, and look for a solution $x^1 \in N(s)$ better than s. If found, we replace s with x^1 and iterate the same search. We continue this way until we get to a solution $s = x^i$ such that $f(s) = \min\{f(x) | x \in N(s)\}$. We say that s is a *local optimum*. Replacing x^i with x^{i+1} is called performing a *move* of the search. The total number of moves performed from x^0 to the local optimum is called the *length of the convergence*. If $x \in N(s)$ and $f(x) < f(s)$ we say that the move from s to x is an *improving move*. There are two main strategies of LS, namely *first-improvement* and *best-improvement*. In the first-improvement strategy, x^{i+1} is the first solution that we find in $N(x^i)$ such that $f(x^{i+1}) < f(x^i)$. In best-improvement, x^{i+1} is such that $f(x^{i+1}) = \min\{f(x) | x \in N(x^i) \wedge f(x) < f(x^i)\}$. Neither one of these strategies is better than the other on all instances.

A popular LS neighborhood for the TSP is the so-called K-OPT. Let $K \in \mathbb{N}$ be a constant. A K-OPT move on a tour T consists in first removing a set R of K edges and then inserting a set I of K edges so as $(T \setminus R) \cup I$ is still a tour. A K-OPT move is improving if $c(I) < c(R)$, while it is *best improving* if $c(R) - c(I)$ is the maximum over all possible choices of R, I.

The first use of K-OPT dates back to 1958 with the introduction of 2-OPT in [3]. In 1965 Lin [5] described the 3-OPT neighborhood, and experimented with the $\Theta(n^3)$ algorithm. The instances which could be tackled at the time were fairly small ($n \leq 150$). In 1968, Steiglitz and Weiner [8] described an improvement over Lin's method which made it 2 or 3 times faster, but still cubic in nature.

The exploration of the K-OPT neighborhood, for a fixed K, might be considered "fast" from a theoretical point of view, since there is an obvious polynomial algorithm (complete enumeration, of time $\Theta(n^K)$). However, in practice, complete enumeration makes the use of K-OPT impossible already for $K = 3$, if n is large enough. For a given tour of, say, $n = 5000$ nodes, the time required to try all 3-OPT moves, on a reasonably fast computer, is more than an hour, let alone converging to a local

*DMIF, University of Udine, giuseppe.lancia@uniud.it
†DEI, University of Padova, marcello.dalpasso@unipd.it

optimum. For this reason, 3-OPT has never been really adopted for the heuristic solution of TSP instances of interest.

An important recent result in [4] proves that, under a widely believed hypothesis similar to the $P \neq NP$ conjecture, it is impossible to find the best 3-OPT move with a worst-case algorithm of time $O(n^{3-\epsilon})$ for any $\epsilon > 0$ so that complete enumeration is, in a sense, optimal. However, this gives us little consolation when we are faced with the problem of applying 3-OPT to a large TSP instance. In fact, for complete enumeration the average case and the worst case coincide, and we might wonder if there exists a better practical algorithm, much faster than complete enumeration on the majority of instances but still $O(n^3)$ in the worst case. The algorithm described in this paper is such an example.

1.1 Our contribution

The TSP is today very effectively solved, even to optimality, by using sophisticated mathematical programming based approaches, such as Concorde [2]. No matter how ingenious, heuristics can hardly be competitive with these approaches when the latter are given enough running time.

However, heuristics such as local search have a great quality: they are simple (to understand, to implement and to maintain) and, in general, very fast, so that they can overcome their simpleminded-ness by being able to sample a huge amount of good solutions in a relatively small time. Of course if too slow, we lose all the interest in using a LS approach despite its simplicity.

This is exactly the situation for the 3-OPT. The goal of our work has been to show that, with a clever implementation of the search for improving moves, it can be actually used since it can become orders of magnitude faster than its standard implementation.

Let us give a flavour of the type of results that we can achieve. Assume we have an average PC and 5hrs of time which we want to devote to local search starting from as many random tours as possible. Assume $n = 1000$ and each convergence goes through 500 intermediate solutions. With the enumerative approach we could sample only one local optimum and would stop while halfway through the second convergence. With our method we would sample more than 300 local optima.

As we will see, another advantage of our method is that while we are approaching the local optimum, our method becomes faster in finding (if it exists) an improving move or the best improving move. The brute force approach, on the other hand, takes costant time for finding the best improving move, or, if it is looking for the first improvement, it does in fact become slower, since the number of candidates to try before finding an improvement becomes larger and larger near the local optimum.

2 Notation

Let $G = (V, E)$ be a complete graph on n nodes, and $c : E \mapsto \mathbb{R}^+$ be a cost function for the edges. Without loss of generality, $V = \{0, 1, \ldots, \bar{n}\}$, where $\bar{n} = n - 1$. We will describe an effective strategy for finding either the best improving or any improving move for a given current tour $(v_1, \ldots, v_n)$ which, wlog, we assume to be $T = (0, 1, \ldots, \bar{n})$.

We will be using modular arithmetic frequently. For convenience, for each $x \in V$ and $t \in \mathbb{N}$ we define

$$x \oplus t := (x + t) \mod n, \qquad x \ominus t := (x - t) \mod n.$$

We define the *forward distance* $d^+(x, y)$ from node x to node y as the unique $t \in \{0, \ldots, n-1\}$ such that $x \oplus t = y$. Similarly, we define the *backward distance* $d^-(x, y)$ from x to y as the $t \in \{0, \ldots, n-1\}$

such that $x \ominus t = y$. Finally, the *distance* between any two nodes x and y is defined by

$$d(x, y) := \min\{d^+(x, y), d^-(x, y)\}$$

A 3-OPT move is fully specified by two sets, i.e., the set of removed and of inserted edges. We call a *removal set* any set of three tour edges, i.e., three edges of type $\{i, i \oplus 1\}$. A removal set is identified by a triple $S = (i_1, i_2, i_3)$ with $0 \leq i_1 < i_2 < i_3 \leq \bar{n}$, where the edges removed are $R(S) := \{\{i_j, i_j \oplus 1\} : j = 1, 2, 3\}$. We call any such triple a *selection*. A selection S is *complete* if $d(i_j, i_h) \geq 2$ for each $j \neq h$, otherwise we say that S is a *partial* selection. Complete selections are more important than partial selections, since there is only a quadratic number of partial selections but a cubic number of complete ones.

Let S be a selection and $I \subset E$ with $|I| = 3$. If $(T \setminus R(S)) \cup I$ is still a tour then I is called a *reinsertion set*. Given a selection S, a reinsertion set I is *pure* if $I \cap R(S) = \emptyset$, and *degenerate* otherwise. Finding the best 3-OPT move when the reinsertions are constrained to be degenerate is $O(n^2)$ (in fact, 3-OPT degenerates to 2-OPT in this case). Therefore, the most computationally expensive task is to determine the best move when *the selection is complete and the reinsertion is pure*. We refer to this kind of moves as *true* 3-OPT. Thus, in the remainder of the paper we will focus on true 3-OPT moves.

2.1 Reinsertion schemes

Let $S = (i_1, i_2, i_3)$ be a complete selection. When the edges $R(S)$ are removed from a tour, the tour gets broken into 3 consecutive segments which we can label by $\{1, 2, 3\}$ (segment j ends at node i_j). Since the selection is pure, each segment is indeed a path of at least one edge. A reinsertion set patches back the segments into a new tour. If we adopt the convention to start always a tour with segment 1 traversed clockwise, the reinsertion set: (i) determines a new ordering in which the segments are visited along the tour and (ii) may cause some segments to be traversed counterclockwise. In order to represent this fact we use a notation called a *reinsertion scheme*. A reinsertion scheme is a signed permutation of $\{2, 3\}$. The permutation specifies the order in which the segments $2, 3$ are visited after the move. The signing $-s$ tells that segment s is traversed counterclockwise, while $+s$ tells that it is traversed clockwise. For example, the third reinsertion set depicted in figure 1 is represented by the reinsertion scheme $< +3, -2 >$ since from the end of segment 1 we jump to the beginning of segment 3 and traverse it forward. We then move to the last element of segment 2 and proceed backward to its first element. Finally, we close the cycle by going back to the first element of segment 1.

There are potentially $2^2 \times 2! = 8$ reinsertion schemes, but for some of these the corresponding reinsertion sets are degenerate. A scheme for a pure reinsertion must not start with $+2$, nor end with "$+3$", nor be $< -3, -2 >$. This leaves only 4 possible schemes, let them be $r_1, \ldots, r_4$.

Clearly, there is a bijection between reinsertion schemes and reinsertion sets. If r is a reinsertion scheme, we denote by $I(r)$ the corresponding reinsertion set. The enumeration of all true 3-OPT moves can be done as follows: (i) We consider, in turn, each reinsertion scheme $r_1, \ldots, r_4$; (ii) Given r_j, we consider all complete selections $S = (i_1, i_2, i_3)$, obtaining the moves defined by $(R(S), I(r_j))$. The cost of step (ii) by complete enumeration is $\Theta(n^3)$. In the remainder of the paper we will focus on a method for lowering significantly, in practice, the complexity of this step.

2.2 The number of complete selections

For space reasons we state the following theorem without proof. For generality, we consider the complete selections of K-OPT, i.e., k-tuples $(i_1, \ldots, i_n)$ of increasing indices with $d(i_j, i_{j+1}) > 1$ for

two consecutive indices.

Theorem 1. *For each $K = 2, \ldots, \lfloor n/2 \rfloor$ the number of complete K-OPT selections is*

$$\binom{n-K+1}{K} - \binom{n-K-1}{K-2}$$

Corollary 1. *The number of complete 3-OPT selections is*

$$\binom{n-2}{3} - (n-4) = \frac{n^3 - 9n^2 + 20n}{6}$$

From the corollary, and knowing that there are 4 pure reinsertion schemes for each 3-OPT complete selection, we can compute the number $T_3(n)$ of true 3-OPT moves. For example, it is $T_3(1,000) = 660,680,000$, while $T_3(5,000) = 83,183,400,000$ and $T_3(10,000) = 666,066,800,000$ giving a striking example of why the exploration of the 3-OPT neighborhood would be totally impractical unless some effective strategies were adopted.

3 Speeding-up the search: The basic idea

Our method can be used to find either the best improving selection or any improving selection. In the rest of the paper we will focus on the Best-Improvement case, since it is the harder of the two. The changes needed in order to adopt the method for a First-Improvement search are trivial.

According to the enumerative strategy outlined in the previous section, suppose we have fixed a reinsertion scheme r, and want to find the best selection for it. Our goal is to provide an alternative, much faster, way to do it than the following, classical, "nested-for" approach over all indices:

```
for ( i1 = 0; i1 ≤ n̄ − 4; i1++ )
    for ( i2 = i1 + 2; i2 ≤ n̄ − 2 − P(i1 = 0); i2++ )
        for ( i3 = i2 + 2; i3 ≤ n̄ − P(i1 = 0); i3++ )
            evaluateMove(i1, i2, i3, r); [* check if move is improving. possibly update best *]
```

(The expression $\mathcal{P}(A)$, given a predicate A returns 1 if A is true and 0 otherwise).

Our idea for speeding-up the search is based on this consideration. Suppose there is a magic box which knows all the pairs of indices that belong to some best improving selection, and that we can inquire the box by specifying two labels (e.g., "i_2" and "i_3", etc.). The box, in time $O(1)$ would return us a pair of values (e.g., v_2 and v_3) such that there exists at least one best improving 3-OPT move in which the two specified indices have those particular values. At this point we could enumerate the values for the missing index and determine the best completion possible. This way, finding a best improving 3-OPT move would take $\Theta(n)$ time rather than $\Theta(n^3)$.

The bulk of our work has then been to simulate, heuristically, a similar magic box, i.e., a data structure that can be queried and should return two out of the three indices of a best improving selection much in a similar way as described above. In our heuristic version, the box, rather than returning a pair of indices that are certainly in a best improving solution, returns a pair of indices that *are likely to be in a best improving solution.* As we will see, this can already greatly reduce the number of possible selections candidate to be best improving. In order to assess the likelihood of two specific indices to be in a best solution, we will use suitable two-arguments functions described in the next sections.

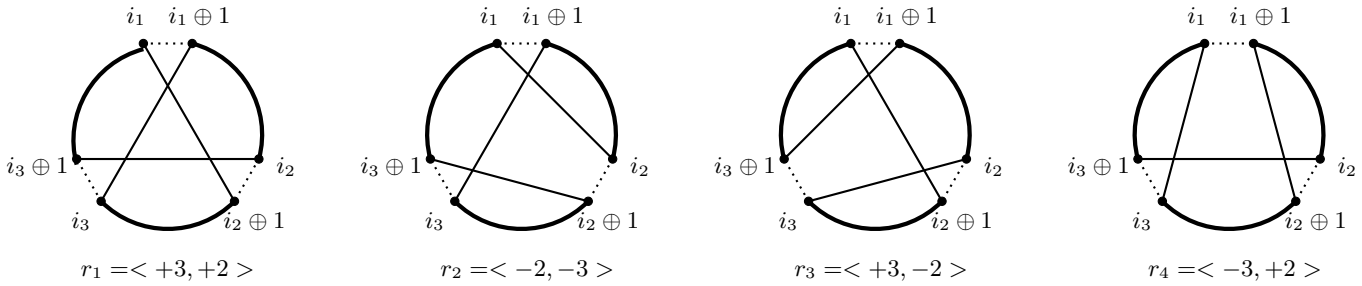

Figure 1: The pure reinsertion schemes of 3-OPT

3.1 The fundamental quantities τ^+ and τ^-

We define two functions of $V \times V$ into $\mathbb{R}$ which, loosely speaking, will be used to determine, for each pair of indices of a selection, the contribution of that pair to the value of a move. The rationale is that, the higher the contribution, the higher the probability that a particular pair is in a best selection.

The two functions are called $\tau^+()$ and $\tau^-()$. For each $a, b \in \{0, \ldots, \bar{n}\}$, we define: (1) $\tau^+(a, b)$ to be the difference between the cost from a to its successor and to the successor of b, and (2) $\tau^-(a, b)$ to be the difference between the cost from a to its predecessor and to the predecessor of b:

$$\tau^+(a, b) = c(a, a \oplus 1) - c(a, b \oplus 1), \qquad \tau^-(a, b) = c(a, a \ominus 1) - c(a, b \ominus 1)$$

Clearly, each of these quantities can be computed in time $O(1)$, and computing their values for a subset of possible pairs can never exceed time $O(n^2)$.

4 A 3-phase procedure for searching the 3-OPT neighborhood

The pure 3-OPT reinsertion schemes are four (see figure 1), namely : $r_1 =< +3, +2 >$; $r_2 =< -2, -3 >$; $r_3 =< +3, -2 >$; and $r_4 =< -3, +2 >$.

Notice that r_3 and r_4 are symmetric to r_2. Therefore, we can just consider r_1 and r_2 since all we say about r_2 can be applied, *mutatis mutandis*, to r_3 and r_4 as well.

Given a reinsertion scheme r, the cost $\Delta(i_1, i_2, i_3)$ of a move with selection $S = (i_1, i_2, i_3)$ is the difference between the cost of the removed edges $\{\{i_1, i_1 \oplus 1\}, \{i_2, i_2 \oplus 1\}, \{i_3, i_3 \oplus 1\}\}$ and the cost of the reinsertion set $I(r)$. A key observation is that we can break-up the function $\Delta()$, that has $\Theta(n^3)$ possible arguments, into a sum of functions of *two parameters each* (therefore, $\Theta(n^2)$ possible arguments). That is, we'll have

$$\Delta(i_1, i_2, i_3) = f^1(i_1, i_2) + f^2(i_2, i_3) + f^3(i_1, i_3) \tag{1}$$

for suitable functions $f^1(), f^2(), f^3()$, each representing the contribution of a particular pair of indices to the value of the move. The domains of these functions are subsets of $\{0, \ldots, \bar{n}\} \times \{0, \ldots, \bar{n}\}$ which limit the valid input pairs to values obtained from two specific elements of a selection. Let $\mathcal{S}$ be the set of all complete selections. For $a, b \in \{1, 2, 3\}$, let us define

$$\mathcal{S}_{ab} := \{(x, y) : \exists (v_1, v_2, v_3) \in \mathcal{S} \text{ with } v_a = x \text{ and } v_b = y\} \tag{2}$$

Then the domain of f^1 is $\mathcal{S}_{12}$, the domain of f^2 is $\mathcal{S}_{23}$ and the domain of f^3 is $\mathcal{S}_{13}$.

Below, we describe these functions for r_1 and r_2 (remember that r_3 and r_4 are symmetric to r_2):

[r_1:] We have $I(r) = \{\{i_1, i_2 \oplus 1\}, \{i_2, i_3 \oplus 1\}, \{i_1 \oplus 1, i_3\}\}$ (see Figure 1) and

$$\Delta(i_1, i_2, i_3) = \tau^+(i_1, i_2) + \tau^+(i_2, i_3) + \tau^+(i_3, i_1)$$

$$f^1 : (x, y) \in \mathcal{S}_{12} \mapsto \tau^+(x, y); \quad f^2 : (x, y) \in \mathcal{S}_{23} \mapsto \tau^+(x, y); \quad f^3 : (x, y) \in \mathcal{S}_{31} \mapsto \tau^+(x, y).$$

[r_2:] We have $I(r) = \{\{i_1, i_2\}, \{i_2 \oplus 1, i_3 \oplus 1\}, \{i_1 \oplus 1, i_3\}\}$ (see Figure 1) and

$$\Delta(i_1, i_2, i_3) = \tau^+(i_1, i_2 \ominus 1) + \tau^-(i_2 \oplus 1, i_3 \oplus 2) + \tau^+(i_3, i_1)$$

$$f^1 : (x, y) \in \mathcal{S}_{12} \mapsto \tau^+(x, y \ominus 1); \; f^2 : (x, y) \in \mathcal{S}_{23} \mapsto \tau^+(x \oplus 1, y \oplus 2); \; f^3 : (x, y) \in \mathcal{S}_{31} \mapsto \tau^+(x, y).$$

The two-parameter functions f^1, f^2, f^3 can be used to discard very quickly from consideration all triples such that no two of the indices give a sufficiently large contribution to the total. Better said, we keep in consideration only candidate triples for which at least one contribution of two indices is large enough. Assume we want to find the best selection and we currently have a selection $S^* = (\bar{v}_1, \bar{v}_2, \bar{v}_3)$ of value $V := \Delta(S^*)$ (the current "champion"). We make the trivial observation that for a selection (i_1, i_2, i_3) to beat S^* it must be

$$\left(f^1(i_1, i_2) > \frac{V}{3}\right) \vee \left(f^2(i_2, i_3) > \frac{V}{3}\right) \vee \left(f^3(i_1, i_3) > \frac{V}{3}\right)$$

These are not exclusive, but possibly overlapping conditions, which we will consider in turn with a three-phase algorithm. For $j = 1, 2, 3$, in the j-th phase, we will restrict our search to the selections (i_1, i_2, i_3) which satisfy the j-th condition. Furthermore, we will not enumerate those selections with a complete enumeration, but rather *from the most promising to the least promising*, stopping as soon as we realize that no selection of the phase has still the possibility of being the best selection overall.

The best data structure for performing this kind of search (which hence be used for our heuristic implementation of the magic box described in section 3) is the *Max-Heap*. A heap is perfect for taking the highest-valued elements from a set, in decreasing order. It can be built in linear time with respect to the number of its elements and has the property that the largest element can be extracted in logarithmic time, while still leaving a heap.

We then build a a max-heap H^1 in which, for each $(x, y) \in \mathcal{S}_{12}$ such that $f^1(x, y) > V/3$, we put the triple $(x, y, f^1(x, y))$. The heap is organized according to the value of f^1. Assume H^1 has L elements. Building the heap has cost $O(L)$, and then extracting the element of maximum $f^1()$ value, while keeping the heap structure, has cost $O(\log L)$. We will now start extracting the elements from the heap. Let us denote by $(x[j], y[j], f^1(x[j], y[j]))$ the j-th element extracted. Heuristically, we might say that $x[1]$ and $y[1]$ are the most likely values that the pair of indices i_1 and i_2 can take in a best improving selection, since these values give the largest possible contribution (as far as i_1 and i_2 are concerned) to the move value (1). We will keep extracting the maximum $(x[j], y[j], f^1(x[j], y[j]))$ from the heap as long as $f^1(x[j], y[j]) > V/3$. This does not mean that we will extract all the elements of H^1, since the value of V could change (namely, increase) during the search and hence the extractions might terminate before the heap is empty.

Each time we extract the heap maximum, we have that $x[j]$ and $y[j]$ are two possible indices (i.e., i_1 and i_2) out of three for a candidate selection to beat S^*. With a linear-time scan, we can search

the third missing index (i.e., i_3) and see if we get indeed a better selection than S^*. To find i_3 we run a for-cycle with $y[j] + 2 \leq i_3 \leq \bar{n}$, checking each time if $(x[j], y[j], i_3)$ is a better selection than S^*. Whenever this is the case, we update S^*. Notice that we also update V so that the number of elements still in the heap for which $f^1(x, y) > V/3$ may decrease considerably. Say that, overall, M elements are extracted from the heap. Then the cost of the phase is $O(n^2 + L + M(n + \log L))$ which, on our tests, was is in practice growing as a quadratic function of n. Worst-case, this is $O(n^3)$ like complete enumeration but, as we will show in our computational experiments, it is *much* smaller in practice. This is because the Mn triples which are indeed evaluated for possibly becoming the best selection have a much bigger probability of being good than a generic triple, since two of the three indices are guaranteed to help the value of the move considerably.

After this phase of probing the heap H^1, we run an analogous phase in which we probe a heap H^2 containing all the triples $(x, y, f^2(x, y))$ for which $(x, y) \in \mathcal{S}_{23}$ and $f^2(x, y) > V/3$. For each element $(x[j], y[j])$ extracted from the heap, we look for the missing index i_1, with $0 \leq i_1 \leq x[j] - 2$. Notice that the value V determining which elements belong to H^1 is the value of the current best solution, i.e., it is not the value that V had at the start of the previous phase, but the value it had at its end.

A third and final phase probes a heap H^3 containing all the triples $(x, y, f^3(x, y))$ for which $(x, y) \in \mathcal{S}_{13}$ and $f^3(x, y) > V/3$, each time looking for the missing index i_2 in the range $x[j] + 2 \leq i_2 \leq y[j] - 2$.

5 Computational results

Here we describe some preliminary computational results. More experiments and a full detailed discussion of the computational experience are delayed to a journal version of this paper.

In a first set of experiments we have tested the idea on random instances in which the edge costs are uniformly distributed over the interval $[1, \ldots, n^2]$. The following table refers to 10 random graphs for each choice of n from 1,000 to 3,000 with increments of 500. For each instance, we generated a random tour and looked for the best 3-OPT move both with the complete enumeration (C) and our heap-based method (H) :

	$n = 1000$	$n = 1500$	$n = 2000$	$n = 2500$	$n = 3000$
min C	39.416	146.936	396.748	805.064	1452.312
max C	39.936	147.624	385.872	810.576	1463.624
avg C	39.712	147.272	392.300	807.712	1457.236
avg H	0.312	0.688	1.672	2.448	3.336
min H	0.184	0.560	1.560	2.060	2.500
max H	0.372	0.748	1.684	2.936	4.624

Times are in seconds on a Intel Pentium G645 @2.9GHz. We put the rows of avg next to each other and it is easy to see that the speedup of our method goes from about 100× to more than 300× for increasing n. This behavior was true for all instances tried. By fitting the growth of the time required to perform a move from a random permutation over a larger set of graph sizes than what shown here, we determined the quadratic polinomial function $t(n) = An^2 + Bn + C$ microseconds, with $A = 0.097$, $B = 9.18$ and $C = 0$.

When n gets large, comparing the two approaches becomes impractical since the time for complete enumeration gets too long. We can however abort complete enumeration after, say 10,000,000 triples and estimate the final time quite accurately as $\bar{t} \times T_3(n)/10,000,000$, where $\bar{t}$ is the time at abortion

and $T_3(n)$ was given after Corollary 1. This way, for example, we can show that the time for finding the best 3-OPT move from a random solution when $n = 5,000$ is more than 10 hours, while with our method (which we carry out without aborting) it is about 30 seconds.

The same type of behavior is true for instances from the TSPLIB [7], although the speedup percentage seems lower than for uniform-cost random graphs. We sampled 94 instances of TSPLIB and performed a best move from a random starting solution (10 times). The instance sizes went from $n = 21$ to $n = 2392$. For small n it is practically impossible to distinguish the running times and our method starts to emerge as a clear winner for $n \geq 100$. The larger instances are reported below:

instance	`u1817`	`d2103`	`u2152`	`u2319`	`pr2392`
n	1817	2103	2152	2319	2392
avg C	261.560	425.936	460.812	600.008	654.436
avg H	3.500	2.624	6.124	3.312	6.040

6 Conclusions and future directions

We have described a practical method for finding the best 3-OPT move which exhibits, empirically, a quadratic running time. A (probably quite hard) question to study would be to mathematically prove that the expected running time of this algorithm on random instances is lower than cubic. A direction of future research would also be to assess the effectiveness of 3-OPT, now that we can use it, on the TSP, i.e., to study the quality of 3-OPT local optima (for both best- and first- improvement LS).

Finally, we are developing similar ideas to speed-up the search of the 4-OPT neighborhood, which, from preliminary results, appear very promising.

References

[1] Aarts, E. and Lenstra, J.K. editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[2] Applegate, D.L., Bixby, R E. , Chvatl, V., and Cook, WJ. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[3] Croes, G.A. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[4] de Berg, M., Buchin, K., Jansen, B., and Woeginger, GJ. Fine-grained complexity analysis of two classic TSP variants. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 5:1–5:14, 2016.

[5] Lin S. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*, 44(10):2245–2269, Dec 1965.

[6] Papadimitriou C.H. and Steiglitz K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 01 1982.

[7] Reinelt, G. TSPLIB - a traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991.

[8] Steiglitz K. and Weiner P. Some improved algorithms for computer solution of the traveling salesman problem. In *Proc. 6th annual Allerton Conf. on System and System Theory*, pages 814–821. University of Illinois, Urbana, 1968.