I N F O R M A T I K

Alternating Cycles Contribution: a Tour Merging Strategy for the Traveling Salesman Problem

Hisao Tamaki

MPI–I–2003–1-007 March 2003

FORSCHUNGSBERICHT RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Stuhlsatzenhausweg 85 66123 Saarbrücken Germany

Authors' Addresses

Hisao Tamaki
Meiji University
Department of Computer Science
Kawasaki 214-8517, Japan
tamaki@cs.meiji.ac.jp

Abstract

A strategy of merging several traveling salesman tours into a better tour, called ACC (Alternating Cycles Contribution) is introduced. Two algorithms embodying this strategy for geometric instances is implemented and used to enhance Helsgaun's implementaton of his variant of the Lin-Kernighan heuristic. Experiments on the large instances in TSPLIB show that a significant improvement of performance is obtained. These algorithms were used in September 2002 to find a new best tour for the largest instance pla85900 in TSPLIB.

Keywords

Traveling salesman problem, exponential neighborhood local search, tour-merging, alternating-cycles

1 Introduction

In the traveling salesman problem(TSP) [10], we are given a graph and a non-negative cost for each edge of the graph and are asked to construct a cycle of the minimum total cost going through each vertex (often called city) of the graph exactly once. In this paper, we consider the symmetric instances in which the graph is undirected, although our approach is applicable in principle to asymmetric instances (in which the graph is directed and we have to construct a directed cycle). Besides being a classic example of NP-hard problems, TSP is also a notable example of theoretically hard problems for which extremely effective heuristics exist. The instances in the benchmark library TSPLIB [17] have been conquered one by one and today only four remain unsolved. Most of these optimal solutions are due to Applegate, Bixby, Cook and Chvátal [16] including the recent solution for the instance with 15,112 cities. This achievement is made possible by steady progresses in both upper bound and lower bound techniques. We concentrate on the former in this paper, i.e., how to find good tours.

The most important technique for tour-finding is the heuristic due to Lin and Kernighan [9] (the *LK heuristic* henceforth). This status has been reinforced by its chained usage [11], by the modern implementation and thorough experimental analysis by Johnson and McGeoch [7] (see also [2, 8]), and by the powerful variant of Helsgaun [5] based on sequential 5-opt swaps and dynamic edge ranking. Helsgaun's code called LKH [18] has been an indispensable tool in finding the optimal or best known tours for most of the large instances in TSPLIB.

Another important heuristic in tour finding is *tour-merging* due to Applegate *et al* [2] and to Cook and Seymour [4], which plays a complementary role to the LK heuristic. The idea is to merge several good tours (typically obtained by the LK heuristic) into a single graph and solve TSP on this rather sparse graph by rigorous TSP solvers that use branch-and-cut or dynamic programming. This strategy is successfully used in combination with LKH, in finding the optimal tour for the above mentioned 15112-city instance and the best-reported tour for some of the four unsolved instances.

The use of tour-merging has so far been primarily limited to the final stage of tour finding, where the merging is applied to a set of high-quality tours obtained by the LK heuristic. It is conceivable, however, to use this strategy more routinely in the earlier stage of tour finding. This fits in the local search paradigm with *very large scale neighborhoods* [1]. In this paradigm, given the current best solution S, one step of local search consists in finding a best (or sometime simply a good) solution in some gigantic neighborhood of S, which is potentially of exmponential size. The neighborhood of the current tour T_0 in our case is generated by auxiliary tours $T_1, \ldots T_k$ and consists of all the tours in the graph $\bigcup_{i=0}^{k} T_k$. With the previously known techniques of tour-merging, however, such a paradigm is difficult to implement, since the time required for solving a single instance of tour-merging tends to be prohibitively large for such an iterative use, in particular for large instances we are interested in. Our strategy is intended to overcome this difficulty by introducing a somewhat more restricted neighborhood with more structure that can be exploited for efficient solutions.

In the context of genetic algorithms, Nagata and Kobayashi [12] investigated an attractive crossover operation for TSP tours based on alternating cycles. Given two tours, say red and blue, they first partition the disjoint union of these tours into a collection of cycles which alternate between red and blue edges. Then they select randomly or heuristically some subcollection of these alternating cycles and apply them to the red tour, i.e., take the symmetric difference between these cycles and the red tour. The result is a collection of subtours (a 2-factor in graph theoretical terms), which is then greedily connected into one tour. From each pair of parents, several child tours are produced in this manner and the goal in this genetic algorithm context is not only to produce tours with small cost but also to keep diversity of the population.

We adopt this idea of using alternating cycles and turn it into a general strategy of merging tours, which we call ACC (*Alternating Cycles Contribution*). Suppose we are given tours $T_0, T_1, \ldots, T_k$ and seek a good tour in the union $H = \bigcup_{i=0}^{k} T_k$ of these tours. Rather than viewing H as a "flat" graph, we view it as consisting of the *base tour* T_0 and alternating cycles each coming from the symmetric difference between T_0 and some T_i. Then our neighborhood consists of tours that are obtained by applying some subcollection of these alternating cycles to T_0. Though this neighborhood is in general substantially more restricted than the neighborhood consisting of all the tours in H, we speculate that it is sufficiently rich in many cases.

The neighborhood based on alternating cycles have more structure than the neighborhood consisting of all tours of the merged graph. We design algorithms that exploit this structure. In particular, we introduce two algorithms. One is called Greedy ACC and is heuristic. The other is a dynamic program, which is an instance of what we call *Catalanian dynamic programs*, and computes an exactly optimal tour in this neighborhood. Both algorithms have been implemented for 2D Euclidean instances, in which the cities are points in the plane and the cost of an edge is the Euclidean distance between its endpoints. Experiments on the large Euclidean instances of TSPLIB show that these algorithms can be used to enhance the performance of LKH significantly. Most notably, a new best-reported tour for pla85900 (the largest instance in TSPLIB) was found in September 2002 using our method that is about 0.055 % above the best-reported lower bound, a significant improvement over the previous gap of 0.061 % [16, 17, 18][1].

Before concluding this section, we should mention the polynomial time approximation scheme for geometric TSP by Arora [3] as a source of inspiration. The concept of Catalanian dynamic programs is an abstraction of the dynamic program used in his work and we discovered an efficient implementation of this type of dynamic programs in our effort to implement Arora's approximation scheme. More importantly, however, the entire project has been driven by the author's desire to turn Arora's theoretical breakthrough into something tangible in practice.

The rest of this paper is organized as follows. After the preliminaries in Section 2, we describe Greedy ACC in in Section 3 and the dynamic programs in Section 4. In Section 5 we discuss some tools and strategy for filtering alternating cycles so as to make the neighborhood tractable. Then, we describe the application of ACC to enhancing LKH in Section 6 and give experimental results in Section 7. Section 8 is the conclusion.

2 Preliminaries

Throughout this paper, a graph is simple, i.e., without parallel edges or self-loops. The vertex and edge sets of graph G are denoted by $V(G)$ and $E(G)$ respectively. The subgraph of G induced by $U \subseteq V(G)$ is denoted by $G[U]$.

An instance of TSP is a pair (G, c) where G is a graph and $c : E(G) \to \mathrm{R}$ assigns a non-negative cost to each edge of G. A *2-factor* of G is a subgraph of G that contains all vertices of G and in which every vertex has degree 2. A 2-factor consists of one or more cycles and sometimes called a cycle cover (though this may be confusing since a cover in general is allowed to cover each object more than once). A connected 2-factor (a Hamilton cycle) is a *tour*. We often regard each 2-factor T as a set of edges (a subset of $E(G)$). The cost of 2-factor T with respect to the given cost function c, denoted by $c(T)$, is naturally defined to be the sum $\sum_{e \in T} c(e)$ of the costs of edges in T.

[1]This upper bound for pla85900 was subsequently updated by Helsgaun in December 2002

A pair $A = (A^{out}, A^{in})$ of disjoint edge sets A^{out}, $A^{in} \subseteq E(G)$ is called an *alternating cycle* of G, if $A^{in} \cup A^{out}$ forms a cycle of G along which the edges of A^{in} and those of A^{out} appear alternately. We call the edges in A^{out} the out-edges and the edges in A^{in} the in-edges of A. Clearly, the numbers of in-edges and out-edges must be equal and be at least 2.

We say that an alternating cycle $A = (A^{out}, A^{in})$ decomposes into two alternating cycles $B = (B^{out}, B^{in})$ and $C = (C^{out}, C^{in})$ if A^{out} is the disjoint union of B^{out} and C^{out} and A^{in} is the disjoint union of B^{in} and C^{in}. An alternating cycle is said to be elementary, if it cannot decompose into two alternating cycles. The *gain* of alternating cycle A, with respect to the given cost function c, is defined as

$$\mathrm{gain}_c(A) = \sum_{e \in A^{out}} c(e) - \sum_{e \in A^{in}} c(e).$$

Let T be a 2-factor and A be an alternating cycle. We say that A is *applicable* to T if every out-edge of A belongs to T and no in-edge of A belongs to T. When A is applicable to T, the *result of applying A to T*, denoted by $T \Delta A$, is the 2-factor $(T \setminus A^{out}) \cup A^{in}$ of G. Clearly, the cost of this 2-factor is $c(T) - \mathrm{gain}_c(A)$.

Let $\mathcal{A}$ be a collection of alternating cycles of G and T a 2-factor of G. We say that $\mathcal{A}$ is *simultaneously applicable* to T if

1. Each $A \in \mathcal{A}$ is applicable to T, and
2. for each distinct $A, B \in \mathcal{A}$, A^{in} and B^{in} are mutually disjoint and A^{out} and B^{out} are also mutually disjoint.

When $\mathcal{A}$ is simultaneously applicable to T, then $(T \setminus \bigcup_{A \in \mathcal{A}} A^{out}) \cup \bigcup_{A \in \mathcal{A}} A^{in}$ is a 2-factor, which we denote by $T \Delta \mathcal{A}$.

Let $\mathcal{A}$ be a collection of alternating cycles each of which is applicable to 2-factor T. We define the *neighborhood* of T generated by $\mathcal{A}$ by

$$\mathcal{N}_{\mathcal{A}}(T) = \{T \Delta \mathcal{A}' \mid \mathcal{A}' \subseteq \mathcal{A} : \text{simultaneously applicable to } T\}$$

We are interested in the following two problems.

1. Given a tour T on G, generate a collection $\mathcal{A}$ of alternating cycles applicable to T, such that $\mathcal{N}_{\mathcal{A}}(T)$ is likely to contain a tour better than T and moreover the optimization in $\mathcal{N}_{\mathcal{A}}(T)$ is tractable (in a rigorous or, more modestly, a heuristic sense).
2. Given a tour T and a collection $\mathcal{A}$ of alternating cycles applicable to T, find an optimal (or, more modestly, a good) tour in the neighborhood $\mathcal{N}_{\mathcal{A}}(T)$.

The next two sections address the second problem. For the first problem, our basic approach is to use auxiliary tours constructed by any other methods, such as the LK heuristic, and take the symmetric difference between T and these tours. However, to make the resulting neighborhood tractable, we often need to filter the collected alternating cycles. This is discussed in Section 5

3 Greedy ACC

Suppose a tour T_0 and a collection of alternating cycles $\mathcal{A}$ are given such that each alternating cycle in $\mathcal{A}$ is applicable to T_0. In this section, we give a heuristic algorithm that tries to construct a good tour in a certain neighborhood of T_0 that is larger than $\mathcal{N}_{\mathcal{A}}(T_0)$. Let us first define this neighborhood precisely.

Let T be a 2-factor of G. We call an alternating cycle B a *4-cycle connector* of T if the following conditions hold:

(1) B is applicable to T,
(2) B has exactly four edges, and
(3) the two out-edges of B belong to distinct cycles of T.

Applying a 4-cycle connector to T results in a 2-factor in which two cycles of T are merged into one. A 2-factor T' is called a *completion* of T, if T' is connected and is the result of applying some simultaneously applicable set of 4-cycle connectors of T.

Our enlarged neighborhood $\mathcal{N}^*_{\mathcal{A}}(T_0)$ of T_0 generated by $\mathcal{A}$ consists of all 2-factors in $\mathcal{N}_{\mathcal{A}}(T_0)$ together with all of their completions.

Algorithm: Greedy ACC

> **Splitting phase** Let $A_1, A_2, \dots, A_k$ be the list of alternating cycles in $\mathcal{A}$ with positive gain, in the decreasing order of their gains. For $i = 1$ to k do:
>
> > if A_i is applicable to T_{i-1} then set $T_i = T_{i-1} \Delta A_i$; otherwise set $T_i = T_{i-1}$.
>
> **Patching phase** Set $T = T_k$. While there is an alternating cycle A with non-positive gain in $\mathcal{A}$ such that $T \Delta A$ has smaller number of cycles than T, select such A with the maximum gain (minimum absolute value) and redefine T to be $T \Delta A$.
>
> **Completion phase** If T is a tour then return T. Otherwise, i.e., if T has more than one cycle, try to compute a completion of T as follows.

Let B_1, B_2, ... , B_m be the list of all 4-cycle connectors of T listed in the decreasing order of their gains. Let $S_0 = T$.
For $i = 1$ to m do:

> if B_i is applicable to S_{i-1} and $S_{i-1} \Delta B_i$ has smaller number of cycles than S_{i-1} then set $S_i = S_{i-1} \Delta B_i$; otherwise set $S_i = S_{i-1}$.

If S_m is a tour, then return S_m; otherwise report failure.

The algorithm is greedy in each of the three phases but in different manners. In the splitting phase, it tries to accumulate as much gain as possible in a greedy manner, disregarding the eventual cost of connecting the resulting cycles. In the patching phase, it tries to reduce the number of cycles, using as cheap alternating cycles as possible. The greediness in the completion phase is similar, but is more well grounded in the following sense. Consider the graph in which the cycles of the 2-factor S_0 are vertices and the 4-cycle connectors are edges. Then the completion phase is similar to the Kruskals's algorithm for the minimum spanning tree: if this graph is connected and the 4-cycle connectors are mutually independent (i.e., the application of one does not disable that of another) then we are guaranteed to have the minimum-cost completion. Although in general the 4-cycle connectors may interfere with each other and nothing is theoretically guaranteed, we may still expect good results in non-pathological cases. Though the greediness in the first two phases in contrast lacks formal grounds, experiments (see Section 7) show that Greedy ACC works surprisingly well when applied to two tours.

We note that greedy ACC for two input tours is similar to the crossover operator of Nagata and Kobayashi [12] described in the introduction. The completion phase is common to both and the difference lies in the criteria with which we select alternating cycles in the first two phases (which comprise a single phase in their crossover operation). Their crossover operator uses randomness or heuristics designed to balance the quality of the tour and the diversity of population, which is crucial in pure genetic algorithms. Since we rely on other tour generating methods to provide diversity, we focus on optimizing the tour cost and use two types of greediness (splitting phase and connecting phase) in hope of maximizing the chance of getting better tours.

4 Dynamic programs for the ACC neighborhood

In this section, we describe dynamic programs which, given a tour T and a collection $\mathcal{A}$ of alternating cycles applicable to T, finds the best tour in the neighborhood $\mathcal{N}_{\mathcal{A}}(T)$. These dynamic programs are similar to a standard dynamic program for TSP itself, but exploit the structure of the ACC neighborhood to make the optimization tractable for instances with large $\mathcal{A}$. In the first subsection, we describe a basic dynamic program for the ACC neighborhood. Then in the second subsection, we specialize this dynamic program into a more efficient one for planar or near-planar merged graphs.

4.1 Basic dynamic program

For a tour T in G and a set $\mathcal{A}$ of alternating cycles applicable to T, Let $G(T, \mathcal{A})$ denote the graph on $V(G)$ consisting of all edges of T and of the alternating cycles of $\mathcal{A}$. We first describe a standard dynamic program for TSP on $H = G(T, \mathcal{A})$, in which we regard H as a general graph and disregard its structure imposed by the alternating cycles.

The dynamic program is based on a binary decomposition of the graph similar to the branch decomposition [13]. However, for our purposes it is more convenient to consider a recursive decomposition of the vertex set of H, while branch decomposition is a recursive decomposition of the edge set of H.

Let a graph H be fixed. A *binary decomposition* of H is a binary tree B in which each node p of B is associated with a non-empty set V_p of vertices of H such that the two conditions hold.

1. For the root r of B, $V_r = V(H)$.

2. Each non-leaf node p has exactly two child nodes p_1 and p_2 with $V_p = V_{p_1} \cup V_{p_2}$ and $V_{p_1} \cap V_{p_2} = \emptyset$.

For each node p of a binary decomposition B of H, let D_p denote the set of edges between V_p and $V \setminus V_p$. We call the cardinality of D_p the *width* of node p and define the width of the decomposition to be the maximum width over all nodes in the decomposition. We define a *connection type* on p to be a set of disjoint unordered pairs of edges of D_p. A *path partition* of $G[V_p]$ is a set of vertex-disjoint paths on $G[V_p]$ which collectively contain all vertices in V_p. We say that a path partition $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ has connection type $\tau = \{\{e_1, f_1\}, \{e_2, f_2\}, \ldots, \{e_k, f_k\}\}$ if the endvertices of P_i are the endvertices of e_i and f_i in V_p for each $1 \leq i \leq k$.

Based on a binary decomposition B, we can build a dynamic program to obtain an optimal tour on H as follows. At each node p of B, we have a table that records for every connection type τ on p a minimum-cost path partition of p with connection type τ. The computation proceeds in a usual bottom up manner:

1. At each leaf, each table entry is computed by brute force.
2. Let p be a node with children p_1 and p_2. To compute the table entry for a connection type τ on p, we consider all combinations (τ_1, τ_2) of connection types on p_1 and p_2 respectively such that τ_1 and τ_2 are consistent in the choice of edges between V_{p_1} and V_{p_2} and combine into the connection type τ. Choosing the best combination in terms of the summed table values (where the cost of the intermediate edges are also added), we obtain the optimal value for τ.
3. At the root, the optimal tour cost is computed similarly considering all possible combinations of connection types on its child nodes that constitute a connected tour.

Though this dynamic program is tractable when the size of D_p is small for every node p and the size of V_p is small for every leaf p, the table size at p is asymptotically close to $|D_p|!$ and therefore the allowable size of D_p is fairly low.

Now we turn to the optimization in the ACC neighborhood $N_{\mathcal{A}}(T)$. Observe that the above dynamic program for TSP on $G(T, \mathcal{A})$ is readily applicable, if we impose additional constraint on the feasible solution that it must be a result of applying some subcollection of $\mathcal{A}$. A crucial observation is that this constraint allows us to make a fairly large set of vertices a leaf in the decomposition. More specifically, let p be a leaf of some binary decomposition. In the case of general TSP, V_p with more than, say, 20 vertices is practically impossible to deal with, unless we have some efficient algorithm for minimum-cost path partitions. However, in our optimization within $N_{\mathcal{A}}(T)$, we only need to consider the path partitions that are generated by some choice of alternating cycles. If the edges of the induced subgraph $G[V_p]$ comes from k alternating cycles, then there are at most 2^k path partitions to consider, for all the connection types. Thus, we can make p a leaf irrespective of the size of V_p, if k is small, say less than 20. This means that we are able not only to reduce the size of the binary decomposition but also to often avoid large decomposition width by simply not decomposing a large vertex set. This is a huge advantage in constructing a good binary decomposition.

4.2 Catalanian dynamic programs – exploiting planarity

The dynamic program for $N_{\mathcal{A}}(T)$ described in the previous subsection can be made considerably more efficient when $G(T, \mathcal{A})$ is planar. In the following description of how this can be done, we assume that $G(T, \mathcal{A})$ is planar and also assume a fixed plane embedding of $G(T, \mathcal{A})$.

The central idea is to use a binary decomposition with a certain topological structure. We say that a binary decomposition B of $H = G(T, \mathcal{A})$ is *topologically well structured* if, for each node p of B, $H[V_p]$ is connected and all edges in D_p lie in the outer face of $H[V_p]$ (in the assumed plane embedding of H).

Let p be a node in a topologically well structured binary decomposition and let the edges of D_p listed as $e_1, e_2, \ldots, e_k$ in the counterclockwise order they are placed in the outer face of $H[V_p]$. We say that a connection type τ on p is *non-crossing* if, for no pair $(e_{i_1}, e_{i_2}), (e_{j_1}, e_{j_2})$ of pairs in τ we have $i_1 \leq j_1 \leq i_2 \leq j_2$. In other words, τ is non-crossing if we can draw mutually non-crossing curves connecting the specified pairs of endvertices. Since the edges of the graph do not cross each other, it is clear that every path partition on p has a non-crossing connection type and therefore we need only to consider non-crossing connection types in our dynamic program. Arora [3] observes that the number of non-crossing connection types on p is given by

$$f(|D_p|) = \sum_{2 \leq 2m \leq |D_p|} \binom{|D_p|}{2m} C_m$$

where C_m is the number of well-parenthesized strings each consisting of m left and m right parentheses, known as the Catalan number. Since $C_m = \frac{1}{m+1}\binom{2m}{m} = O(m^{-\frac{3}{2}} 2^{2m})$, $f(n)$ is much slower growing than $n!$.

The advantage of topologically well structured decomposition is not only that it drastically reduces the number of connection types to be considered but also that it allows the following implementation technique. The key fact is that non-crossing connection types are quite compactly encoded by a pair of bit strings (σ, π), where σ is the characteristic vector of the subset of D_p we choose for connection, and π encodes the well-parenthesized string specifying the the pairing of chosen edges. This compact encoding allows us efficient lookup of various precomputed tables in the dynamic program that results in a dramatical speedup over naive implementations [15]. Due to this implementation technique it is possible to routinely execute a dynamic program on a binary decomposition of width as large as 16.

To emphasize the practical importance of this implementational advantage, we give a name to dynamic programs that use topologically well-structured decompositions: *Catalanian dynamic programs.*

4.3 Extending Catalanian dynamic program to non-planar cases

It is often too restrictive to require the graph $G(T, \mathcal{A})$ to be planar. This restriction can be somewhat relaxed: edges may cross each other in the plane embedding as long as we are only interested in tours that are not self-crossing. In other words, if edges e and f cross each other in the embedding, our search space excludes any tour in which both e and f are used. Note that, if the original TSP instance is 2D Euclidean and the embedding is the natural one, then this restriction does not rule out any optimal tours.

The trick to allow this extension is simple. We introduce a new vertex for each crosspoint of edges and we divide each edge through this cross point into two edges. The binary decomposition and the dynamic program is constructed for the resulting graph. In executing the dynamic program we impose the constraint that if edges e_1 and e_2 are the result of dividing some common original edge then e_1 and e_2 must be simultaneously used or simultaneously excluded. Incorporating such constraint into the dynamic program is straightforward. We also need to modify the base case of the dynamic program to allow tours to not visit the artificial vertices. This modification is also straightforward.

4.4 Tour-based decomposition

It is clear that the efficiency of the dynamic program described in the previous subsections crucially depends on the binary decomposition we use. It depends linearly on the size and exponentially on the width of the decomposition. So it is crucial to obtain a decomposition with small width.

Since the problem of finding a decomposition with minimum width is NP-hard [14], we need some heuristics. Our approach is to decompose the vertex set in natural accordance with the current tour T_0. In our decomposition, the vertex set V_p associated with each node p in the decomposition is the set of vertices on a single segment (subpath) P_p of T_0. If a non-root node p has two child nodes p_1 and p_2, then their corresponding paths P_{p_1} and P_{p_2} are obtained from P_p by removing an edge. This simple decomposition strategy has the following rationale/advantages.

1. Recall that, in our framework, the given set of alternating cycles is not an absolute mandate. If it turns out to generate an intractable neighborhood, we are willing to prune the set as long as the resulting set remains promising. In contrast, we always need to keep the edges of the base tour. From this perspective, it makes sense to concentrate on a decomposition scheme that gives the minimum decomposition width of 2 for the graph consisting of only the tour edges.

2. The ordering of vertices along the tour represents the locality of the vertex set. Our decomposition honors this locality which may help in obtaining a small width of the decomposition, although this hope is only heuristic.

3. Within the class of tour-based decompositions, the decomposition with the minimum width can be computed by a simple dynamic program described below.

Let tour T_0 and the set $\mathcal{A}$ of alternating cycles applicable to T_0 be fixed. In constructing the binary decomposition, we associate a leaf to a vertex set U if at most K alternating cycles of $\mathcal{A}$ have a vertex in U, where K is a constant (for example, 20 is chosen in our experiments). For each subpath P of T_0, let $w(P)$ denote the width of the best possible binary decomposition of the subgraph of $G(T_0, \mathcal{A})$ induced by $V(P)$. Then, $w(P)$ can be determined by a simple recurrence.

1. $w(P) = 0$ if the number of alternating cycles in $\mathcal{A}$ that has at least one vertex on P is at most K;

2. $w(P) = \min_{P_1+P_2=P} \max\{w(P_1), w(P_2), |\partial(V(P_1))|, |\partial(V(P_2)|)\}$, otherwise, where
$\partial(V(P))$ denotes the set of edges of $G(T_0, \mathcal{A})$ between the vertices on P and the vertices not on P, and $P_1 + P_2$ denotes the concatenation of paths P_1 and P_2.

This dynamic program requires a table of size $O(n+m^2)$ and runs in $O(n+m^3)$ time, where n is the number of cities and m is the number of distinct edges in the alternating cycles of $\mathcal{A}$. Note that we are mainly interested in the case where m is smaller than the number of vertices. Nonetheless, time proportional to m^3 is prohibitive in many cases and therefore we adopt a heuristic in the actual implementation. This heuristic consists in selecting $m' < m$ positions in the tour where the paths can be cut in the recursive decomposition. These positions are chosen so as to minimize (again in a heuristic sense) the number of edges across these positions. This reduces the running time of the dynamic program to $O(n + m'^3)$ time, at the cost of possibly missing the real optimal partitioning. We choose the value $m' \leq 400$ in the experiment.

5 Filtering

One important component of the ACC strategy is the filter. Given a tour T_0 and a set $\mathcal{A}$ of alternating cycles applicable to T_0, the role of the filter is to convert $\mathcal{A}$ into $\mathcal{A}'$ such that the neighborhood $\mathcal{N}_{\mathcal{A}'}(T_0)$ is solvable by the dynamic program described in the previous section with tolerable amount of resources. $\mathcal{A}'$ may simply be a subset of $\mathcal{A}$ or the result of some more involved simplifying conversion.

A variety of heuristics for filtering are conceivable. In this section, we describe tools available for such heuristics and the particular heuristic used in our experiments.

A useful feature of the dynamic programming solver described in the previous section is that it can answer the *tractability query*: does the pair $(T_0, \mathcal{A})$ generate a neighborhood that can be solved by the solver roughly within the pre-specified amount of resource? This can be answered by solving the second dynamic program for the decomposability with appropriate parameters of decomposition.

A generic filtering process based on the tractability query would go as follows: we pick alternating cycles from $\mathcal{A}$ one by one with some heuristic ordering, accepting each if and only if adding it to the chosen set keeps the set tractable. In the ordering we consider not only the gain of each alternating cycle but also how it helps in connecting the cycles that would be generated by the applications of large-gain alternating cycles.

Another tool is the greedy ACC algorithm. Suppose the set $\mathcal{A} = \bigcup_{i=0}^{k} \mathcal{A}_i$, where $\mathcal{A}_i$ is the result of decomposing the symmetric difference between T_0 and T_i into elementary alternating cycles. In general, even for a single $\mathcal{A}_i$ the pair $(T_0, \mathcal{A}_i)$ may be intractable when tours T_0 and T_i are quite different. Instead of trying to filter out some alternating cycles from $\mathcal{A}_i$, we may apply the greedy ACC algorithm to the pair (T_0, T_i), let the result be T_i', and obtain the corresponding set of alternating cycles $\mathcal{A}_i'$ from the symmetric difference between T_0 and T_i'. It is empirically observed that T_i' is usually far closer to T_0 than T_i is, and therefore the pair $(T_0, \mathcal{A}'_i)$ is usually tractable. We may also expect that some of the "good" alternating cycles in $\mathcal{A}_i$ is preserved in $\mathcal{A}'_i$, possibly in modified forms, since greedy ACC tries to use all positive-gain alternating cycles in $\mathcal{A}_i$.

The particular heuristic adopted in our experiment uses these tools in the following manner. We are given the base tour T_0 and auxiliary tours $T_1, \ldots, T_k$. For each T_i, $1 \leq i \leq k$, we use greedy ACC to obtain the set $\mathcal{A}'_i$ as described above. Then, we scan the sets $\mathcal{A}'_1, \ldots, \mathcal{A}'_k$ one by one for inclusion in the resulting set. More precisely, we let $\mathcal{B}_0$ to be an empty set and for $1 \leq i \leq k$ we define $\mathcal{B}_i = \mathcal{B}_{i-1} \cup \mathcal{A}'_i$ if the pair $(T_0, \mathcal{B}_{i-1} \cup \mathcal{A}'_i)$ is tractable and $\mathcal{B}_i = \mathcal{B}_{i-1}$ otherwise. $\mathcal{B}_k$ is the output of this heuristic.

6 Applying ACC to enhance the chained LK procedure

This section describes an application of the ACC strategy to enhance the performance of the chained LK (CLK) procedure [11, 7, 2].

An execution of CLK consists of a large number of trials, each of which performs the LK search and produces a local optimal solution. The best result among all the previous trials is maintained as the current best tour. The important feature of the CLK is the dependence of each trial upon the previous trials. Typically, this dependence is introduced by taking the initial tour of the search in each trial to be some randomized perturbation of the current best tour. Helsgaun's LKH [5] differs from this standard approach in that it uses a random initial tour independent of the previous tours but has its search process dependent on the previous tours through a dynamic reordering of the edge list at each vertex. In either case, it has been observed that this dependence almost always results in significantly better tours than would be obtained by the same number of completely independent LK trials.

Call a trial in CLK a success if it finds a better tour than the previous best. In a typical execution of CLK, say consisting of 1000 trials, only a very small number of trials, say 30, are successes. (See the experimental results below). The failed trials are wasteful in the same sense as the money bet on losing horses is wasteful. In our case, however, the races may not have been over. We observe that the failed trials provide the opportunity of applying the ACC strategy: even if the tour produced is longer than the current best, it may contain some good alternating cycles to contribute.

Our enhanced CLK goes as follows. Here, r is a parameter to control the frequency of dynamic programming steps, which is set to 10 in the actual experiments.

1. Set T to any tour.
2. Clear *stack* and set *count* to 0.
3. Repeat the following pre-specified number of times or until a tour of satisfiable quality is obtained:
   (a) Execute a single trial LK search and let the result be T_{LK}
   (b) Apply the greedy ACC algorithm to the pair T, T_{LK} and let the result be T_{GREEDY}.
   (c) If T_{GREEDY} is better than T then rename this tour as T, clear *stack* and go to the next iteration of the repeat loop.
   (d) Otherwise, push T_{LK} to *stack* and increment *count*.
   (e) If *count* $< r$ then skip the following and go to the next iteration.
   (f) Construct a set of alternating cycles $\mathcal{A}$ for T from the tours in the stack, using the filtering heuristic described in the previous section, and compute the best tour in the neighborhood $\mathcal{N}_{\mathcal{A}}(T)$ by the dynamic program. If the result is a tour better than T then update T and clear stack. Reset *count* to 0 in either case.

In the actual implementation, we do not push T_{LK} into the stack but instead push the set of alternating cycles that result from decomposing the symmetric difference between T and T_{GREEDY}, since this set is what is really needed by our filter. If we use other filtering strategies, however, we may need (effectively) to push T_{LK} itself as in the above description.

7 Experimental results

In this section, we describe an implementation of the enhanced CLK procedure and some experimental results.

We use LKH(version 1.2) [5], Helsgaun's implementation of CLK, as the target of enhancement. The source code [18] (which is written in C) of LKH is slightly modified to allow the call of a single LK trial from outside. Otherwise, no change is made to LKH and default parameters are used throughout the experiments.

Our implementation of Catalanian dynamic programs is specialized to 2D Euclidean instances, and uses the natural embedding of the merged graph. Extension to general instances would require a planar embedding procedure. In contrast, our implementation of Greedy ACC can be more easily adapted for general instances. The main procedure as well as Greedy ACC and the dynamic programming algorithm are coded in Java with JNI (Java Native Interface) as the interface to LKH. A compute server consisting of 44 SPARC III+ CPUs (900MHz) is used, although each experiment is performed using a single CPU. In all the experiments, 3600MB is allocated to the Java heap memory.

7.1 Large TSPLIB instances

This subsection describes detailed results of the experiments on some of the large TSPLIB instances, namely pla7397, usa13509, d15112, pla33810, and pla85900. All of these are essentially 2D Euclidean instances, although the distances are approximated by integers. The numbers in the instance name represent the number of cities. The first three are the largest instances in TSPLIB for which the optimal solutions are known and the last two are the largest instances in TSPLIB.

For each instance, the procedure is applied in three modes:

LK_ONLY Both the greedy ACC and the dynamic program are disabled.

LK_AND_GREEDY The dynamic program is disabled.

ALL Full procedure as described in the previous section.

The LK_ONLY mode is essentially equivalent to the original LKH, except for the slight overhead of the Java main procedure. In the ALL mode, the parameter r is set to 10, so the dynamic program is executed at most once in 10 steps.

Each execution consists of 1000 LK trials and, for each s which is a positive multiple of 100, the following quantities are measured: the number of successes in the first s trials, the quality of the tour after the first s trials, the time consumed in the first s trials. For the smaller three instances, these quantities are averaged over 10 trials with different random seeds. For the larger two, they are averaged over 5 or 3 trials. In the case of pla7397, where the optimal solution is found in some trials, the number of such trials is counted. The trials are continued even after they have found the optimal solution, so as to keep the averaging meaningful. The tour quality is in terms of the percent in excess over the optimal solution, for the first three instances, and over the best reported lower bound for the larger two.

The initialization time for LKH and for ACC procedures are measured separately. For LKH, this includes the time for initial edge ordering, though the full ascent computation is avoided using the pi-file provided with LKH distribution. This is roughly 50, 120, 150, 1050, 13000 seconds for the used instances in the ascending order of their size. The initialization time for ACC is mainly that for setting up instance-independent precomputed tables used in our dynamic programs, and roughly 240 seconds for all the instances.

Tables 1-5 show the results. In almost all cases, the addition of Greedy ACC and/or the dynamic program enhances the performance of LKH significantly. This is most prominent for pla85900, the largest instance. The LKH without these enhancement produced a tour of 0.2024 % above the lower bound after 1000 trials using 76325 seconds. With Greedy ACC, the result after 100 trials (in 12094 seconds) has already much better 0.1371 % quality. Using almost the same amount of time in 600 trials, a tour of 0.0984 % excess is produced with Greedy ACC and a tour of 0.0964 % excess is produced with both Greedy ACC and the dynamic program. For smaller instances, similar tendency is observed, although the time overhead of the dynamic program is more conspicuous in the smaller instances.

The advantage of adding the dynamic programming step may not appear very large from the tables. However, this small difference becomes more important when the tour quality approaches the optimum solution and it becomes more and more difficult to find improved tours. Indeed, we were able to construct a tour for pla85900 that was considerably shorter than the previously best reported tour, using the procedure described above with parameter $r = 1$ (i.e., dynamic programming after every LK trial). We collected 7 results from independent executions of this procedure, each running less than a week, and merged these tours using again both Greedy ACC and the dynamic program. The resulting tour has length 142385237 and is about 0.055 % above the best reported lower bound of length 142307500. The previously best reported tour had been produced by LKH and has length 142393738 (0.061 % above the lower bound). This record was however broken again, three months later, with a tour of length 142383704 (0.054 % above the lower bound) by Helsgaun using his new LKH code.

7.2 DIMACS challenge instances

This subsection lists the results on DIMACS-challenge instances [19, 8]. Table 6–8 follow the format of the challenge. Only geometric instances are included as our

Table 1: pla7397: average of 10 trials, opt is no. times out of 10

	LK_ONLY				LK_AND_GREEDY				ALL			
trials	succ.	opt /10	quality (%)	time (sec)	succ.	opt /10	quality (%)	time (sec)	succ.	opt /10	quality (%)	time (sec)
100	16.3	0	0.0456	140	17.0	0	0.0526	245	17.1	0	0.0520	256
200	20.8	0	0.0348	276	22.2	2	0.0188	495	22.6	2	0.0199	536
300	23.3	0	0.0261	424	23.3	3	0.0145	745	23.5	4	0.0167	836
400	25.3	0	0.0216	569	24.6	4	0.0099	1014	25.1	5	0.0110	1143
500	27.1	0	0.0183	714	25.2	6	0.0079	1282	25.6	6	0.0087	1471
600	28.5	0	0.0158	860	25.2	6	0.0079	1548	25.6	6	0.0087	1850
700	29.8	0	0.0137	1006	25.2	6	0.0079	1806	26.2	6	0.0074	2280
800	30.3	0	0.0132	1153	25.9	6	0.0038	2071	27.0	6	0.0043	2735
900	31.1	1	0.0121	1303	26.2	6	0.0035	2341	27.7	7	0.0028	3218
1000	31.5	1	0.0119	1447	26.5	7	0.0031	2611	27.7	7	0.0028	3831

Table 2: usa13509 : average of 10 trials

	LK_ONLY			LK_AND_GREEDY			ALL		
trials	success	quality (%)	time (sec)	success	quality (%)	time (sec)	success	quality (%)	time (sec)
100	20.1	0.0535	420	29.5	0.0411	604	26.4	0.0367	1011
200	26.7	0.0408	746	39.5	0.0291	1129	35.6	0.0244	2336
300	31.3	0.0354	1067	46.3	0.0236	1646	40.5	0.0176	4091
400	34.3	0.0325	1382	51.4	0.0185	2158	43.2	0.0154	6076
500	36.3	0.0313	1692	54.4	0.0175	2658	44.2	0.0152	8419
600	38.6	0.0288	1989	56.4	0.0157	3157	46.1	0.0136	10496
700	41.2	0.0275	2292	58.4	0.0147	3655	47.2	0.0127	12768
800	43.3	0.0261	2587	59.5	0.0144	4149	48.3	0.0122	14978
900	44.0	0.0259	2904	60.9	0.0138	4637	49.9	0.0114	17494
1000	45.7	0.0248	3205	62.2	0.0136	5142	50.4	0.0109	20005

Table 3: d15112: average of 10 trials

	LK_ONLY			LK_AND_GREEDY			ALL		
trials	success	quality (%)	time (sec)	success	quality (%)	time (sec)	success	quality (%)	time (sec)
100	14.2	0.0513	788	18.9	0.0355	971	18.1	0.0183	2398
200	18.5	0.0407	1413	25.1	0.0270	1819	24.2	0.0108	5289
300	21.5	0.0348	1993	28.3	0.0248	2647	26.0	0.0094	8564
400	23.2	0.0322	2589	31.1	0.0218	3444	27.5	0.0087	12035
500	24.6	0.0302	3155	32.9	0.0208	4235	29.6	0.0076	15403
600	26.2	0.0285	3709	35.5	0.0191	4998	30.2	0.0073	18949
700	28.1	0.0267	4260	37.5	0.0177	5755	31.2	0.0069	22463
800	29.4	0.0252	4774	39.9	0.0166	6513	31.7	0.0067	26105
900	30.4	0.0239	5303	41.7	0.0157	7260	32.2	0.0066	29825
1000	31.2	0.0233	5815	42.7	0.0153	8033	33.0	0.0064	34172

Table 4: pla33810: average of 5 trials

	LK_ONLY			LK_AND_GREEDY			ALL		
trials	success	quality (%)	time (sec)	success	quality (%)	time (sec)	success	quality (%)	time (sec)
100	12.8	0.2132	4530	33.6	0.1308	5508	34.4	0.1269	5851
200	15.6	0.1962	9033	43.0	0.1195	11275	42.2	0.1188	12499
300	17.8	0.1788	12974	46.8	0.1174	17307	44.6	0.1171	20002
400	18.0	0.1761	17303	48.2	0.1170	23194	47.2	0.1148	27524
500	19.0	0.1706	21316	49.8	0.1163	29208	50.8	0.1135	34459
600	19.4	0.1683	25570	51.2	0.1160	35830	51.4	0.1133	41642
700	20.6	0.1657	29468	51.6	0.1158	41747	53.2	0.1108	49585
800	22.0	0.1605	33539	52.0	0.1156	48077	55.4	0.1082	57187
900	22.4	0.1602	37869	52.6	0.1154	54287	57.8	0.1062	65064
1000	23.2	0.1581	42157	53.0	0.1144	60186	58.6	0.1061	73570

Table 5: pla85900: average of 3 trials

	LK_ONLY			LK_AND_GREEDY			ALL		
trials	success	quality (%)	time (sec)	success	quality (%)	time (sec)	success	quality (%)	time (sec)
100	19.3	0.3149	7827	64.3	0.1371	12094	64.3	0.1371	11854
200	26.0	0.2744	15756	94.0	0.1225	23938	94.7	0.1223	24140
300	29.3	0.2531	23298	109.7	0.1114	35189	116.3	0.1084	35856
400	32.0	0.2383	30873	120.3	0.1014	47325	127.3	0.1044	49101
500	33.3	0.2327	38291	126.0	0.1001	59060	137.7	0.0970	61734
600	33.7	0.2286	45829	132.3	0.0984	71176	141.0	0.0964	75207
700	35.7	0.2078	53704	140.0	0.0954	82818	144.0	0.0935	90080
800	36.0	0.2046	61293	149.7	0.0923	94958	146.3	0.0931	104218
900	36.7	0.2041	68665	152.3	0.0922	106438	153.0	0.0863	117983
1000	37.0	0.2024	76325	157.7	0.0889	117476	156.0	0.0854	133085

implementation is specialized for those. Instances of size 100000 and above are also excluded, because of the large initialization time for LKH. We have results for all geometric instances for which the results of LKH are reported. Table 6 shows the results of our implementation in LK_AND_GREEDY mode, while Table 7 shows the results in ALL mode. Table 8 lists the result of unmodified LKH(version 1.2 with the default setting) run on the same machine, which is included for comparison. For each instance with n cities, the number of LKH trials is $n/20$ for LK_AND_GREEDY and ALL. It is $n/5$ for LKH-1.2, except for the largest two instances pla33810 and pla85900, where it is set to $n/10$. The purpose of these differences is to roughly balance the running time and enable the comparison of the tour qualities obtained with similar amount of time.

For smaller instances, no significant improvement is observed on the quality of the tours. On the other hand, the overhead of ALL mode on both time and memory is obvious. This is due to the large precomputed tables used for the dynamic programming. For instances with 10000 cities and above, the advantage of our methods starts to be observed. Our implementation in ALL mode produced better tours than the pure LKH for 13 instances out of the 16 instances with 10000 cities or more. For 6 of those 13 successful instances, it did so with less amount of time than the pure LKH. Our implementation in LK_AND_GREEDY mode produced better tours than the pure LKH for 11 instances out of those 16 instances, for 6 of which it did so with less amount of time than the pure LKH. The tendency is more obvious for instances with more than 30000 cities. For all of 6 such instances, our implementation in ALL mode produced better tours than the pure LKH, with less time for 5 of those. For 5 of those 6 instances, our implementation in LK_AND_GREEDY mode produced better tours with less time than the pure LKH. Although the memory consumption of our implementation is significantly larger than the pure LKH, it is in an affordable range.

8 Concluding remarks

We have introduced a strategy of tour merging called ACC and two algorithms, Greedy ACC and a dynamic program, that embody this strategy. It is experimentally demonstrated that these algorithms can be used to significantly enhance the performance of other methods based on LK heuristics for large geometric instances. These algorithms were also used in combination with LKH to find a new best tour for the largest TSPLIB instance, at the time of the experiment.

The experimental results reported in this paper should be interpreted as an example that shows the potential of the ACC approach. It can be combined with any other methods in principle, including the future implementations of LK heuristics with possible further improvements. In the other direction, we may consider applying ACC to "lighter" LK implementations, which are not able to produce tours of quality comparable to LKH for large instances but run considerably faster, trying to enhance them to outperform LKH. This direction is left for further explorations.

The current implementation is specialized for 2D Euclidean instances. Extensions to the general case as well as to asymmetric instances are left as future work.

Acknowledgment

Part of this work was done while the author was visiting the Max-Planck-Institut für Informatik at Saarbrücken. He thanks the algorithms group of the institute for their hospitality. He thanks Keld Helsgaun for his permission to modify and use the source code of LKH. He also thanks Kazutoshi Takahashi for his involvement in the early stage of this research. The tables in the experiment section are prepared with ExpLab [6].

Table 6: DIMACS challenge: LK_AND_GREEDY, 0.05N trials

instance	tour length	time(sec)	memory	instance	tour length	time(sec)	memory
E1k.0	23368355	15.7	20.4M	E1k.1	22987959	17.5	20.7M
E1k.2	23023351	14.9	20.4M	E1k.3	23149856	13.3	20.4M
E1k.4	22698717	14.6	20.4M	E1k.5	23192391	11.6	20.4M
E1k.6	23369551	17.4	20.4M	E1k.7	22886793	15.9	20.4M
E1k.8	23036837	14.9	20.6M	E1k.9	23356256	13.2	20.4M
E3k.0	40636973	236	22.6M	E3k.1	40315287	203.3	22.1M
E3k.2	40304456	217.2	22.6M	E3k.3	40598973	194.3	22.1M
E3k.4	40758964	257.3	22.6M	E10k.0	71876963	3197.8	34.6M
E10k.1	72059083	3763.2	34.6M	E10k.2	71833206	3113.6	34.5M
E31k.0	127334557	77535.5	75.9M	E31k.1	127488634	75201.8	75.9M
C1k.0	11388227	18.9	20.4M	C1k.1	11377365	22.5	20.4M
C1k.2	10855033	28.5	20.4M	C1k.3	11886966	17.3	20.4M
C1k.4	11504210	17.3	20.4M	C1k.5	11399266	28.7	20.4M
C1k.6	10166992	31.7	20.4M	C1k.7	10676547	20.9	20.4M
C1k.8	11605723	16.0	20.4M	C1k.9	10916260	25.3	20.4M
C3k.0	19198289	184.5	22.3M	C3k.1	19156859	280.1	22.2M
C3k.2	19549251	250.4	22.6M	C3k.3	19121346	197.2	22M
C3k.4	19144467	193.2	22.6M	C10k.0	33014565	3999.4	34.7M
C10k.1	33206060	3096.2	34.6M	C10k.2	33163546	3070	34.8M
C31k.0	59579944	43551	76.3M	C31k.1	59459105	44735.7	88.3M
dsj1000	18660436	19.6	20.4M	pr1002	259460	13.3	20.4M
u1060	224307	16.7	20.7M	vm1084	239432	14.5	20.8M
pcb1173	56892	18.9	21.7M	d1291	50888	25.3	22.3M
rl1304	254228	18.9	22M	rl1323	270675	29.4	22.3M
nrw1379	56643	28.5	22.6M	fl1400	20167	53.7	22.9M
u1432	152970	29.5	23.2M	fl1577	22264	97.3	24.3M
d1655	62129	42.0	24.6M	vm1748	336743	43.4	25.3M
u1817	57229	50.6	26.2M	rl1889	316549	51.1	26.7M
d2103	80533	92.7	20M	u2152	64295	71.3	20.2M
u2319	234256	132.5	21M	pr2392	378032	106.7	20.7M
pcb3038	137708	190.9	21.8M	fl3795	28883	542.6	23.3M
fnl4461	182571	429.6	24M	rl5915	565775	706.2	26.6M
rl5934	556822	717	26.6M	pla7397	23260728	2102	29.7M
rl11849	923686	4311.9	38.3M	usa13509	19987965	8820.3	41M
brd14051	469454	10069.1	42.8M	d15112	1573211	11733.7	44.5M
d18512	645339	19469.8	50.5M	pla33810	66061284	124974.7	81M
pla85900	142430559	844353.4	184.6M				

Table 7: DIMACS challenge: ALL, 0.05N trials

instance	tour length	time(sec)	memory	instance	tour length	time(sec)	memory
E1k.0	23367551	245.4	475.2M	E1k.1	22990419	249.1	437.1M
E1k.2	23023874	243.3	475.2M	E1k.3	23149930	244.6	475M
E1k.4	22711233	243.8	475.2M	E1k.5	23192391	243.2	437.1M
E1k.6	23370821	248.3	475M	E1k.7	22903918	245.9	474.9M
E1k.8	23027801	246	475M	E1k.9	23368573	246.3	475.2M
E3k.0	40639961	403.1	494M	E3k.1	40319674	397.4	480.2M
E3k.2	40304851	398.3	474.9M	E3k.3	40595818	379.5	472.6M
E3k.4	40761661	423.3	476M	E10k.0	71874624	5372.6	766.9M
E10k.1	72042807	7005.6	738.2M	E10k.2	71825995	7208.9	721M
E31k.0	127306523	95421.6	878.9M	E31k.1	127475412	97529.5	877.4M
C1k.0	11452455	251	475.1M	C1k.1	11378767	249	437.1M
C1k.2	10855033	258.7	475M	C1k.3	11886966	249	437.4M
C1k.4	11504196	253.2	475.2M	C1k.5	11407886	252.2	475M
C1k.6	10169722	256.6	475.2M	C1k.7	10671355	251.1	475.2M
C1k.8	11609179	248.8	437.5M	C1k.9	10912675	254.8	475M
C3k.0	19199749	357.8	472.2M	C3k.1	19020912	419	474.9M
C3k.2	19558908	410.4	479.7M	C3k.3	19109418	389.6	473.2M
C3k.4	18864816	365.1	472.3M	C10k.0	33021514	3077.6	683.8M
C10k.1	33250622	2751.2	693M	C10k.2	33162187	4973.8	719.6M
C31k.0	59565503	45637.4	825.1M	C31k.1	59492148	41178.8	842.7M
dsj1000	18690556	250.4	475.2M	pr1002	259048	248.4	437.1M
u1060	224094	248.4	472.6M	vm1084	239592	246.9	468.2M
pcb1173	56950	249.9	440.5M	d1291	50936	253.1	470.7M
rl1304	253464	249.3	433.6M	rl1323	270639	257.2	470.4M
nrw1379	56643	255.9	472.4M	fl1400	20167	264.4	473M
u1432	152970	261.9	471.5M	fl1577	22267	286.2	474.6M
d1655	62135	265.6	474.7M	vm1748	336592	265.9	480.5M
u1817	57229	273.7	483.4M	rl1889	317591	279	484M
d2103	80533	292.9	476.6M	u2152	64337	286.6	477M
u2319	234256	316.2	469.7M	pr2392	378074	300.1	470.5M
pcb3038	137713	395	479.9M	fl3795	28886	541.3	481.1M
fnl4461	182575	932.7	638.6M	rl5915	566050	721.2	596.2M
rl5934	556808	716.7	566.1M	pla7397	23261853	1428	658.2M
rl11849	923895	5454.4	722.6M	usa13509	19986988	10619.5	757.5M
brd14051	469407	16365.7	789.4M	d15112	1573371	21255.3	864.4M
d18512	645251	34719.4	803.6M	pla33810	66058732	100887.7	938.5M
pla85900	142435540	636710.4	1034.5M				

Table 8: DIMACS challenge: LKH-1.2, 0.2N trials (0.1N for pla33810 and pla85900)

instance	tour length	time(sec)	memory	instance	tour length	time(sec)	memory
E1k.0	23360648	16.4	4.9M	E1k.1	22985695	14.7	4.9M
E1k.2	23025429	9.4	4.9M	E1k.3	23149856	7.2	4.9M
E1k.4	22698717	9.6	4.9M	E1k.5	23192391	8.2	4.9M
E1k.6	23349803	12.4	4.9M	E1k.7	22883024	8.0	4.9M
E1k.8	23029608	9.8	4.9M	E1k.9	23356256	7.8	4.9M
E3k.0	40637390	142.6	5.1M	E3k.1	40317738	147.1	5.1M
E3k.2	40306853	167.5	5.1M	E3k.3	40600328	160.6	5.2M
E3k.4	40757980	219.9	5.1M	E10k.0	71873305	2603.6	12M
E10k.1	72046162	3365.3	12.1M	E10k.2	71835083	3132	12M
E31k.0	127334671	104254.6	33.4M	E31k.1	127528654	117054.1	33.4M
C1k.0	11387430	21.9	4.9M	C1k.1	11385449	15.6	4.9M
C1k.2	10855033	44.7	4.9M	C1k.3	11886966	18.3	4.9M
C1k.4	11503892	22.7	4.9M	C1k.5	11395014	21.4	4.9M
C1k.6	10167085	38.6	4.9M	C1k.7	10665684	26.0	4.9M
C1k.8	11605723	14.5	4.9M	C1k.9	10924048	20.3	4.9M
C3k.0	19209357	214.6	5.2M	C3k.1	19018172	307.5	5.2M
C3k.2	19548677	316.2	5.2M	C3k.3	19108964	193.4	5.2M
C3k.4	18864046	137.6	5.2M	C10k.0	33035573	3256.8	12.3M
C10k.1	33285274	2910	12.2M	C10k.2	33269823	4391.1	12.2M
C31k.0	59569831	41658.6	34.4M	C31k.1	60250134	50059.5	34.3M
dsj1000	18668986	20.6	4.9M	pr1002	259045	7.7	4.9M
u1060	224121	28.9	5.2M	vm1084	239297	17.0	5.3M
pcb1173	56892	12.9	5.8M	d1291	50989	21.1	6.4M
rl1304	253383	12.2	6.6M	rl1323	270199	14.2	6.7M
nrw1379	56643	15.7	7.1M	fl1400	20164	85.7	7.2M
u1432	152970	26.0	7.4M	fl1577	22561	83.7	8.4M
d1655	62128	29.2	8.9M	vm1748	336636	26.5	9.6M
u1817	57205	41.2	10.2M	rl1889	316553	33.8	10.8M
d2103	80529	125.2	3.9M	u2152	64307	68.7	4.1M
u2319	234256	189.4	4.3M	pr2392	378032	69.8	4.4M
pcb3038	137728	144.5	5M	fl3795	28881	911.5	5.7M
fnl4461	182579	271.5	6.5M	rl5915	565728	442.1	7.8M
rl5934	556611	657.6	7.8M	pla7397	23264478	4101.1	9.3M
rl11849	923854	4363.9	13.8M	usa13509	19984407	6713.1	15.8M
brd14051	469434	9484.3	16.3M	d15112	1573308	11134.2	17.3M
d18512	645366	23488	20.8M	pla33810	66102906	154831.6	35.4M
pla85900	142622651	764018.6	87M				

References

[1] R.K. Ahuja, Ö. Ergun, J.B. Orlin and A.P. Punnen, A Survey of Very Large-Scale Neighborhood Search Techniques, to appear in Discrete Applied Mathematics, 2002.

[2] D. Applegate, R. Bixby, V. Chvátal and W. Cook, Finding tours in the TSP, Report Number 99885, Research Institute for Discrete Mathematics, Universitat Bonn, 1999.

[3] S. Arora, Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems, J. ACM, 45(5):753–782, 1998.

[4] W. Cook and P.D. Seymour, Tour merging via branch-decomposition, to appear in INFORMS Jounal on Computing, 2003

[5] K. Helsgaun, An effective implementation of the Lin-Kernighan traveling salesman heuristic, European Journal of Operations Research, 12:106–130, 2000.

[6] S. Hert, L. Kettner, T. Polzin and G. Schäfer, A tool set for computational experiments, http://explab.sourceforge.net

[7] D.S. Johnson and L.A. McGeoch, The traveling salesman problem: a case study. In *Local Search in Combinatorial Optimization (eds.E. Aarts and J.K. Lenstra)*, 215–310, Wiley, Chichester, 1997.

[8] D.S. Johnson and L.A. McGeoch, Experimental Analysis of Heuristics for the STSP, In *The Traveling Salesman Problem and its Variations (eds. G. Gutin and A. Punnen)*, Kluwer Academic Publishers, 2002.

[9] S. Lin and B.W. Kernighan, An Effective Heuristic Algorithm for the Traveling Salesman Problem, Operations Research, vol. 21, 498-516, 1973.

[10] E.L. Lawler and J.K. Lenstra and A.H.G. Rinnooy Kan and D.B. Shmoys, The traveling salesman problem, John Wiley, 1985.

[11] O. Martin, S.W. Otto and E.W. Felten, Large-step Markov chains for the traveling salesman problem, Complex Systems, 5(3):299–326, June 1991.

[12] Y. Nagata and S. Kobayashi, Edge Assembly Crossover: a high-power genetic algorithm for the traveling salesman problem, Proc. 7th International Conference on Genetic Algorithms, 1997.

[13] N. Robertson and P.D. Seymour, Graph minors X. Obstructions to tree-decomposition, Journal of Combinatorial Theory, Series B, 153-190, 1991

[14] P.D. Seymour and R. Thomas, Call routing and the ratcatcher. Combinatorica, 14(2), 217-241, 1994

[15] K. Takahashi, Approximation algorithms for the traveling salesman problem in the plane based on dynamic programming, Masters Thesis (in Japanese), Department of Computer Science, Meiji University, February 2001.

[16] TSP home page: http://www.math.princeton.edu/tsp/

[17] TSPLIB: http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

[18] LKH: http://www.dat.ruc.dk/ keld/research/LKH/

[19] DIMACS TSP implementation challenge: http://www.research.att.com/ dsj/chtsp

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2003-NWG2-002	F. Eisenbrand	Fast integer programming in fixed dimension
MPI-I-2003-NWG2-001	L.S. Chandran, C.R. Subramanian	Girth and Treewidth
MPI-I-2003-4-001	M. Tarini, H.P.R. Lensch, M. Goesele, H. Seidel	3-D Aqusition of Mirrowing Objects
MPI-I-2003-2-001	P. Maier	Compositional Circular Assume-Guarantee Rules Cannot Be Sound And Complete
MPI-I-2003-1-010	H. Tamaki	Soon to be published
MPI-I-2003-1-009	B. Csaba	On the Bollobás – Eldridge conjecture for bipartite graphs
MPI-I-2003-1-008	P. Sanders	Soon to be published
MPI-I-2003-1-007	H. Tamaki	Alternating cycles contribution: a strategy of tour-merging for the traveling salesman problem
MPI-I-2003-1-006	H. Tamaki, M. Dietzfelbinger	On the probability of Rendezvous in Graph
MPI-I-2003-1-005	M. Dietzfelbinger, P. Woelfel	Almost Random Graphs with Simple Hash Functions
MPI-I-2003-1-004	E. Althaus, T. Polzin, S.V. Daneshmand	Improving Linear Programming Approaches for the Steiner Tree Problem
MPI-I-2003-1-003	R. Beier, B. V÷*cking*	Random Knapsack in Expected Polynomial Time
MPI-I-2003-1-002	P. Krysta, P. Sanders, B. V÷*cking*	Scheduling and Traffic Allocation for Tasks with Bounded Splittability
MPI-I-2003-1-001	P. Sanders, R. Dementiev	Asynchronous Parallel Disk Sorting
MPI-I-2002-4-002	F. Drago, W. Martens, K. Myszkowski, H. Seidel	Perceptual Evaluation of Tone Mapping Operators with Regard to Similarity and Preference
MPI-I-2002-4-001	M. Goesele, J. Kautz, J. Lang, H.P.A. Lensch, H. Seidel	Tutorial Notes ACM SM 02 A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2002-2-008	W. Charatonik, J. Talbot	Atomic Set Constraints with Projection
MPI-I-2002-2-007	W. Charatonik, H. Ganzinger	Symposium on the Effectiveness of Logic in Computer Science in Honour of Moshe Vardi
MPI-I-2002-1-008	P. Sanders, J.L. Trõff	The Factor Algorithm for All-to-all Communication on Clusters of SMP Nodes
MPI-I-2002-1-005	M. Hoefer	Performance of heuristic and approximation algorithms for the uncapacitated facility location problem
MPI-I-2002-1-004	S. Hert, T. Polzin, L. Kettner, G. Schõfer	Exp Lab A Tool Set for Computational Experiments

MPI-I-2002-1-003	I. Katriel, P. Sanders, J.L. Trõff	A Practical Minimum Scanning Tree Algorithm Using the Cycle Property
MPI-I-2002-1-002	F. Grandoni	Incrementally maintaining the number of l-cliques
MPI-I-2002-1-001	T. Polzin, S. Vahdati	Using (sub)graphs of small width for solving the Steiner problem
MPI-I-2001-4-005	H.P.A. Lensch, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web
MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-006	H. Nivelle, S. Schulz	Proceeding of the Second International Workshop of the Implementation of Logics
MPI-I-2001-2-005	V. Sofronie-Stokkermans	Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators
MPI-I-2001-2-004	H. de Nivelle	Translation of Resolution Proofs into Higher Order Natural Deduction using Type Theory
MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes
MPI-I-2001-2-002	P. Maier	A Set-Theoretic Framework for Assume-Guarantee Reasoning
MPI-I-2001-2-001	U. Waldmann	Superposition and Chaining for Totally Ordered Divisible Abelian Groups
MPI-I-2001-1-007	T. Polzin, S. Vahdati	Extending Reduction Techniques for the Steiner Tree Problem: A Combination of Alternative-and Bound-Based Approaches
MPI-I-2001-1-006	T. Polzin, S. Vahdati	Partitioning Techniques for the Steiner Problem
MPI-I-2001-1-005	T. Polzin, S. Vahdati	On Steiner Trees and Minimum Spanning Trees in Hypergraphs
MPI-I-2001-1-004	S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel	An Adaptable and Extensible Geometry Kernel
MPI-I-2001-1-003	M. Seel	Implementation of Planar Nef Polyhedra
MPI-I-2001-1-002	U. Meyer	Directed Single-Source Shortest-Paths in Linear Average-Case Time
MPI-I-2001-1-001	P. Krysta	Approximating Minimum Size 1,2-Connected Networks
MPI-I-2000-4-003	S.W. Choi, H. Seidel	Hyperbolic Hausdorff Distance for Medial Axis Transform
MPI-I-2000-4-002	L.P. Kobbelt, S. Bischoff, K. Kõhler, R. Schneider, M. Botsch, C. R÷*ssl, J. Vorsatz*	Geometric Modeling Based on Polygonal Meshes
MPI-I-2000-4-001	J. Kautz, W. Heidrich, K. Daubert	Bump Map Shadows for OpenGL Rendering
MPI-I-2000-2-001	F. Eisenbrand	Short Vectors of Planar Lattices Via Continued Fractions
MPI-I-2000-1-005	M. Seel, K. Mehlhorn	Infimaximal Frames: A Technique for Making Lines Look Like Segments
MPI-I-2000-1-004	K. Mehlhorn, S. Schirra	Generalized and improved constructive separation bound for real algebraic expressions
MPI-I-2000-1-003	P. Fatourou	Low-Contention Depth-First Scheduling of Parallel Computations with Synchronization Variables
MPI-I-2000-1-002	R. Beier, J. Sibeyn	A Powerful Heuristic for Telephone Gossiping
MPI-I-2000-1-001	A branch and cut algorithm for the optimal solution of the side-chain placement problem	

MPI-I-1999-4-001	J. Haber, H. Seidel	A Framework for Evaluating the Quality of Lossy Image Compression
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages