# An $\mathcal{O}(n \log n)$ Algorithm for the $K$-Template Traveling Salesman Problem

Jack A.A. van der Veen * Gerhard J. Woeginger †
Shuzhong Zhang ‡

November 1995

**Abstract**

In this paper a one-machine scheduling model is analyzed where $n$ different jobs are classified into $K$ groups depending on which additional resource they require. The change-over time from one job to another consists of the removal time or of the set-up time of the two jobs. It is sequence-dependent in the sense that the change-over time is determined by whether or not the two jobs belong to the same group. The objective is to minimize the makespan. This problem can be modeled as an asymmetric Traveling Salesman Problem with a specially structured distance matrix. For this problem we give a polynomial time solution algorithm that runs in $O(n \log n)$ time.

**Keywords**: Single Machine Scheduling, Traveling Salesman Problem, Polynomial Time Algorithm.

*Nijenrode University, The Netherlands School of Business, Straatweg 25, 3621 BG Breukelen, The Netherlands.

†TU Graz, Institut für Mathematik B, Steyrergasse 30, A-8010 Graz, Austria.

‡Econometric Institute, Erasmus University Rotterdam, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands.

# 1 Introduction

In this paper we consider the following problem. There are $n$ jobs, $J_1, \ldots, J_n$, to be processed without interruption on a single machine. In order to process a job, a certain additional resource is required, e.g. a fixture or, as we will call it in this paper, a *template*. In total there are $K$ different templates ($K \leq n$) each with a unique form. It is assumed that the machine always works with one template at a time. Moreover, it is assumed that a template can contain only one job each time and jobs can only be processed in one (job-specific) template. So, the jobs can be partitioned according to the template they require. The set of indices that belong to jobs requiring the $k$-th template is denoted by $N_k$ and its cardinality by $n_k := |N_k|$, $1 \leq k \leq K$. Furthermore, there is a *change-over time* between the processing of two jobs. This change-over time is determined by the following two rules.

(R1) After a job has been processed, it has to remain in its template for a certain time period to allow some after-processing, e.g. cooling. For job $J_i$ this time period is denoted by $a_i$ ($i = 1, \ldots, n$ ). However, it is not necessary that the after-processing is done on-line, i.e. that the template remains placed on the machine. If a different template is to be used for the next job, the template, with the job still in it, can be removed from the machine in no time and the after-processing can take place off-line. It is assumed that the after-processing time is negligible compared to the processing times of the jobs on the machine. This implies that the template is free to be used whenever that is required (or, in other words, that the machine never needs to wait idle for a template becoming available). On-line after-processing is necessary only when the following job requires the same template.

(R2) If the same template is not used for the next job, then a set-up time is needed. The length of this set-up time depends on the job to be processed next. For job $J_i$ this set-up time is denoted by $b_i$, $1 \leq i \leq n$.

In order to get an expression for the change-over time, suppose that job $J_j$ is scheduled directly after job $J_i$, and that $i \in N_p$ and $j \in N_q$. If both jobs require the same template (i.e. $p = q$) then the change-over time is given by $a_i$, because the after-processing of job $J_i$ is done on-line, so only after that time $J_j$ can be placed in the template for processing. If $J_j$ requires another template than $J_i$ (i.e. $p \neq q$) then the change-over time is given by $b_j$ because in this case the template with $J_i$ can be taken away from the machine (the after-processing can take place off-line) and the machine has to be set up in order to process job $J_j$.

Our objective is to determine a sequence of the jobs (a *schedule*) such that all jobs are processed on the machine in a minimal amount of time, i.e. we want to minimize the *makespan.* (Note that in the problem described above, the makespan includes the set-up time of the first job and does not include the removal time of the last job). Clearly, this is equivalent to minimizing the summation of change-over times. It is well-known (and easy to see) that by introducing a dummy job $J_0$, the sequencing problem with job-dependent change-over times can be reformulated as an (asymmetric) *Traveling Salesman Problem* (TSP). The distance matrix of the corresponding TSP is given by $D = (d\,[i, j])$ where $d\,[i, j]$ denotes the change-over time

if job $J_j$ is scheduled directly after job $J_i$ $(i, j = 0, \ldots, n)$. In general, the TSP is *NP*-hard. However, this hardness result does not necessarily carry over to our case, since in the situation described above the distance matrix has a rather special structure,

$$d[i,j] = \begin{cases} a_i & \text{if } i \text{ and } j \text{ belong to the same group} \\ b_j & \text{if } i \text{ and } j \text{ belong to different groups} \end{cases}$$

where the groups are given by $N_1, \ldots, N_K$ and a dummy group $N_0$ containing only job $J_0$. Remark that if we assume $a_0 = 0$ and $b_0 = 0$, the minimal length of the above TSP equals the minimal makespan of the $K$ -template one machine scheduling problem. Therefore, in the remainder of this paper we will not take the dummy job into consideration but will, without loss of generality, concentrate on solving the TSP on the jobs $J_1, \ldots, J_n$.

The main objective of this paper is to show that due to the special structure of the distance matrix, the problem allows a *polynomial-time* exact solution algorithm. To be more precise: It will be shown that the TSP restricted to distance matrices satisfying the above property (which will be referred to as the *$K$–template TSP*) is solvable in $O(n \log n)$ time.

The scheduling problem discussed in this paper has several characteristics, including:

- One machine scheduling with sequence dependent change-over times
- Makespan minimization
- Polynomial-time solvability
- Job-groups.

Note that the first two characteristics make it possible to reformulate the problem as a TSP. In the following, we will briefly discuss results in the literature that possess at least three of the above characteristics.

An excellent survey on polynomially solvable special cases of the TSP is given in Gilmore *et al.* [8]. More recent results can be found e.g. in Van Dal [15], in Van der Veen [16] and in the recent survey paper by Burkard *et al.* [5]. One of the few polynomially solvable cases of the TSP that relate directly to the first two characteristics mentioned above, is the special case discussed in Gilmore & Gomory [7]. In that paper, a problem of sequencing jobs for heat-treatment in a furnace is discussed. The change-over time between two consecutive jobs is determined by the pre-specified required starting and ending temperature of the furnace for these jobs. In [7] it is shown that for a certain structure on the time-requirements for heating and cooling the furnace, the resulting problem is solvable in $O(n \log n)$ time.

Polynomially solvable single machine scheduling problems with job-groups are discussed in many papers of which we mention Ahn & Hyun [1], Gupta [9], Gupta *et al.* [10], Monma & Potts [12] and Potts [13].

The following two problems have all four characteristics mentioned above in common with the $K$–template TSP.

The first problem concerns the so called *Aircraft Sequencing Problem* (ASP), see e.g. Psaraftis [14] and Bianco *et al.* [3, 4]. Assume that there are $n$ airplanes waiting for permission to land on a single runway. According to the number of passenger seats, the airplanes can

be divided into $K$ classes (from small to large). Safety regulations require a certain time gap between the landing of two airplanes. This time gap depends on the size of the two airplanes. Assume that for size $k$, $1 \leq k \leq K$, there are $n_k$ airplanes waiting for permission to land and that the flight controller wants to determine the sequence in which all $n$ airplanes will land such that the total time (landing time plus safety time for all airplanes) is minimized. Let $c[p,q]$ denote the required safety time if an airplane from class $N_q$ is to land directly after an airplane from class $N_p$. Clearly, the ASP can be reformulated as a TSP on the $n$ airplanes where the distance matrix $D = (d[i,j])$ is given by

$$d[i,j] = c[p,q]$$

if $i \in N_p$ and $j \in N_q$ for all $i,j \in \{1,\ldots,n\}$ and all $p,q \in \{1,\ldots,K\}$. Psaraftis [14] proposed a dynamic programming approach for the ASP, whereas Bianco *et al.* ([3, 4]) suggested a branch-and-bound method for this problem. In Van der Veen & Zhang [17] (see also Cosmadakis & Papadimitriou [6]) it is shown that for any *fixed* value $K$ (that is not part of the input), the ASP is solvable in $O(n)$ time.

The second problem occurs in *printed circuit board* (PCB) design. Assume that $n$ parts are to be inserted by a robot arm at specified points on a PCB. Each of the parts belongs to one of $K$ part-types. All parts of a given type are stored in a bin. The $K$ bins are located along one side of the PCB. Furthermore, it is assumed that the robot arm can only move sequentially in either horizontal or vertical direction, so that the robot arm travel time from one location to another is proportional to the rectilinear distance between the two locations. The *Robot Tour Problem* (RTP) is to find the optimal sequence of insertions of the parts in the PCB, given the locations of the $n$ insertion points and the locations of the $K$ bins. For more details and extensions see Hamacher [11].

For fixed locations of the bins, the RTP can be reformulated as a TSP on the $n$ insertion points. Obviously, the distance from one insertion point to another equals the distance from the first insertion point to the bin containing the part to be inserted into the second insertion point plus the distance from this bin to the second insertion point. Note that the second part of this distance is fixed, in the sense that independent of the sequence this distance has to be travelled. However, the first part (the distance from first insertion point to bin for the second insertion point) is sequence-dependent. Since all bin-locations are located at the same side of the PCB and since distances are measured rectilinearly, the only part of this distance that is sequence dependent is the distance from the projection of the insertion point on the side of the PCB where the bins are located to the next bin. Let $N_k$ be the set of all parts of type $k$ ($k = 1,\ldots,K$ ). If a part from $N_k$ is to be inserted after a given part $i$, the sequence dependent "distance" $d[i,j]$ to travel from $i$ to $j \in N_k$ has the same value for all $j \in N_k$. It follows that the distance matrix $D = (d[i,j])$ of the TSP modelling the RTP is given by

$$d[i,j] = a^k[i] \quad \text{if} \quad j \in N_k$$

for all $i,j \in \{1,\ldots,n\}$, where $a^1, a^2, \ldots, a^K$ are given $n$-dimensional vectors. A polynomial time algorithm for solving this RTP has been given in Ball & Magazine [2].

The rest of this paper is organized into sections as follows. Section 2 gives some definitions and states an illustrating example for the $K$–template TSP. Sections 3, 4 and 5 provide the combinatorial and algorithmical tools for the polynomial time solution algorithm. The solution algorithm itself is given in Section 6 together with an application to a more general special case of the TSP. Finally, the paper is concluded in Section 7.

## 2 The K-Template TSP

In the following sections, we will derive an $O(n \log n)$ time solution algorithm for the $K$–template TSP. An instance of this problem is fully described by the numbers $n$ and $K$ with $1 \leq K \leq n$, by two vectors $a = [a_1, \ldots, a_n]$ and $b = [b_1, \ldots, b_n]$, and by the partition $N_1, \ldots, N_K$ of the jobs. For notational simplicity, we assume without losing generality that all elements in the vectors $a$ and $b$ are pairwise distinct (i.e. $a_i \neq a_j$ and $b_i \neq b_j$ for $i \neq j$, and $a_i \neq b_j$ for $1 \leq i, j \leq n$). Notice that vectors $a$ and $b$ are not necessarily integral. Since the proposed algorithm is strongly polynomial, by perturbation the above conditions can always be satisfied without changing the complexity of the algorithm. Furthermore, we assume that the jobs are ordered according to the template they require, i.e. $N_k = \{m_k + 1, \ldots, m_k + n_k\}$ for $k = 1, \ldots, K$, where $m_1 := 0$ and $m_k := \sum_{t=1}^{k-1} n_t$ for $k = 2, \ldots, K$. Since we are going to solve the scheduling problem via a traveling salesman formulation, we will use the terms *job* and *city* interchangeably.

Throughout the paper, we will use the following example to illustrate the introduced concepts and algorithms.

**Example**. Let there be $n = 9$ cities and $K = 3$ templates, where $N_1 = \{1, 2, 3\}$; $N_2 = \{4, 5\}$ and $N_3 = \{6, 7, 8, 9\}$. Furthermore, let

$$a = [34 \;\; 67 \;\; 83 \;\; 55 \;\; 68 \;\; 94 \;\; 56 \;\; 65 \;\; 88]$$

and

$$b = [73 \;\; 23 \;\; 54 \;\; 77 \;\; 81 \;\; 39 \;\; 50 \;\; 45 \;\; 29].$$

It follows that the distance matrix is given by

$$D = \begin{bmatrix} 34 & 34 & 34 & 77 & 81 & 39 & 50 & 45 & 29 \\ 67 & 67 & 67 & 77 & 81 & 39 & 50 & 45 & 29 \\ 83 & 83 & 83 & 77 & 81 & 39 & 50 & 45 & 29 \\ 73 & 23 & 54 & 55 & 55 & 39 & 50 & 45 & 29 \\ 73 & 23 & 54 & 68 & 68 & 39 & 50 & 45 & 29 \\ 73 & 23 & 54 & 77 & 81 & 94 & 94 & 94 & 94 \\ 73 & 23 & 54 & 77 & 81 & 56 & 56 & 56 & 56 \\ 73 & 23 & 54 & 77 & 81 & 65 & 65 & 65 & 65 \\ 73 & 23 & 54 & 77 & 81 & 88 & 88 & 88 & 88 \end{bmatrix}. \qquad \square$$

## 3 The Combinatorial Structure of Tour-Sets

Our first step towards a solution of the $K$–template TSP is triggered by the following observation. Given a tour, it is trivial to find the corresponding set of entries in the distance matrix that are "used" in this tour. Such a set of entries corresponding to a tour consists of exactly $n$ entries in the two vectors $a$ and $b$ (since each element in $a$ and $b$ can only be used once). However, the converse of this statement in general does not hold. There are sets of exactly $n$ entries in the two vectors $a$ and $b$ for which no tour exists that exactly uses this set of entries.

**Example** (continued). The tour that visits the cities $(1, 3, 5, 7, 9, 8, 6, 4, 2)$ in this order uses the nine distinct entries $\{a_1, a_2, a_7, a_8, a_9, b_2, b_4, b_5, b_7\}$. On the other hand it is easy to verify that there is no tour that uses e.g. the nine entries $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9\}$. $\square$

This observation rules out simple solution strategies for the $K$ –template TSP, like selecting the smallest $n$ values from the two vectors $a$ and $b$ and constructing a tour with these values. Moreover, they lead to the problem of finding necessary and sufficient conditions under which a given subset of $n$ values from the vectors $a$ and $b$ corresponds to a tour. Our first goal is to find a solution for this problem.

We start with some definitions. Let

$$\begin{aligned} A_k &:= \{a_{m_k+1}, \ldots, a_{m_k+n_k}\} \\ B_k &:= \{b_{m_k+1}, \ldots, b_{m_k+n_k}\} \\ V_k &:= A_k \cup B_k \end{aligned}$$

for $k = 1, \ldots, K$ and $V := V_1 \cup \ldots \cup V_K = \{a_1, \ldots, a_n; b_1, \ldots, b_n\}$. Also, $\sigma_k$ and $\rho_k$ are defined as permutations of $m_k + 1, \ldots, m_k + n_k$ such that

$$\begin{aligned} a_{\sigma_k(m_k+1)} &\leq \ldots \leq a_{\sigma_k(m_k+n_k)} \quad \text{and} \\ b_{\rho_k(m_k+1)} &\leq \ldots \leq b_{\rho_k(m_k+n_k)} \end{aligned}$$

for $k = 1, \ldots, K$. Furthermore, for a subset $S$ of $V$, define

$$\begin{aligned} x_k(S) &:= |S \cap A_k| \\ y_k(S) &:= |S \cap B_k| \end{aligned}$$

for $k = 1, \ldots, K$.

**Definition 3.1** *A set $S \subseteq V$ is called a* tour-set *if it satisfies the following four conditions for all $k \in \{1, \ldots, K\}$:*

**(T1)** $x_k(S) + y_k(S) = n_k$*;*

**(T2)** $y_k(S) \leq \sum_{t \neq k} y_t(S)$*;*

**(T3)** $y_k(S) \geq 1$*;*

**(T4)** *Either $y_k(S) = n_k$ or there is an $i \in N_k$ such that both $a_i \in S$ and $b_i \in S$.* □

Observe that Condition (T4) in fact implies Condition (T3). However, Conditions (T1), (T2) and (T3) together are concerned only with the cardinalities of the intersection sets $|S \cap A_k|$ and $|S \cap B_k|$, whereas Condition (T4) reveals further structure of the intersections. In later analysis this feature will be used in the construction of related tour-sets and value functions.

The *length of a tour-set $S \subseteq V$*, denoted by $d(S)$, is defined as the sum of the values of its elements, i.e.

$$d(S) := \sum_{s \in S} s.$$

A tour $\tau$ on the cities $\{1, \ldots, n\}$ is denoted by the order in which the cities are visited, i.e. $\tau = (i_1, i_2, \ldots, i_n)$. The length of a tour $\tau$ is given by

$$d(\tau) := d[i_n, i_1] + \sum_{t=1}^{n-1} d[i_t, i_{t+1}].$$

Furthermore, the successor of a city $i$ is denoted by $\tau(i)$.

There is a clear link between the just defined tour-sets and "real" TSP tours, as will be shown in the following two lemmas.

**Lemma 3.2** *For every tour $\tau$, there exists a tour-set $S_\tau$ such that $d(S_\tau) = d(\tau)$.*

Proof.

Let $\tau$ be a tour. Define $S_\tau := \{\ d[i, \tau(i)]\ :\ i = 1, \ldots, n\}$. Clearly, $d(S_\tau) = d(\tau)$ holds and it remains to verify the Conditions (T1)–(T4) in Definition 3.1.

(T1) The tour $\tau$ visits the group $N_k$ exactly $n_k$ times. Every time $\tau$ enters $N_k$ while coming from another group $N_\ell$ with $\ell \neq k$, this contributes another value to $S_\tau \cap B_k$. Every time $\tau$ enters $N_k$ while coming from $N_k$ itself, this contributes another value to $S_\tau \cap A_k$. Since there are no other possibilities for getting numbers from $A_k$ and $B_k$ into $S_\tau$, $|S_\tau \cap A_k| + |S_\tau \cap A_k| = n_k$ holds, hence Condition (T1) is fulfilled.

(T2) Note that $y_k(S_\tau)$ equals the number of times tour $\tau$ enters $N_k$ from some other group, which in turn must equal the number of times $\tau$ leaves $N_k$ to some other group. In case $y_k(S_\tau)$ is greater than the sum of all other $y_t(S_\tau)$, this would imply that the number of times $N_k$ is entered from another group is more than the number of possibilities to leave for other groups from $N_k$, which is clearly impossible.

(T3) The tour $\tau$ connects all cities and hence must enter every group $N_k$ at least once while coming from another group.

(T4) If $\tau$ enters $N_k$ always while coming from another group, then $x_k(S_\tau) = 0$ and $y_k(S_\tau) = n_k$. Otherwise, $\tau$ must contain a configuration of the following form: $\tau$ enters $N_k$ from outside, visits a city $i \in N_k$ and then remains within $N_k$ to visit another city $j \in N_k$. Clearly, this implies $a_i \in S_\tau$ and $b_i \in S_\tau$. □

**Lemma 3.3** *For every tour-set $S$, there is a tour $\tau_S$ such that $d(\tau_S) = d(S)$. The tour $\tau_S$ can be computed from $S$ in $O(n)$ time.*

Proof.

Let $S$ be a tour-set. We will construct a tour $\tau_S$ from set $S$ in two phases.

In the *first phase*, consider an initially empty graph on $K$ vertices that are labeled by the $K$ groups $N_1, \ldots, N_K$. Our goal is to compute a set of directed edges $E$ over this vertex set such that (i) the resulting graph is a loopless, strongly connected, Eulerian directed multigraph, and (ii) for $1 \leq k \leq K$, the vertex corresponding to $N_k$ has in-degree $y_k(S)$. We claim that for a tour-set $S$ such a set of edges always exists and that we can compute it in $O(n)$ time. We start with a vector $y = [y_1, \ldots, y_K]$ with $y_k = y_k(S) \geq 1$ and $y_k \leq \sum_{t \neq k} y_t$ by Conditions (T3) and (T2) for $1 \leq k \leq K$. Consider the following procedure for computing a graph that satisfies conditions (i) and (ii).

**Step 1** While there are at least two entries in the vector $y$ that have value greater than one, perform the following operation:

Let $y_a$ and $y_b$ be the largest two entries in $y$. Decrement $y_a$ by one, decrement $y_b$ by one and create an edge going from the vertex $N_a$ to the vertex $N_b$ and another edge from $N_b$ to $N_a$ (i.e. create a simple directed cycle of length two between $N_a$ and $N_b$).

**Step 2** (After performing Step 1, in the vector $y$ there are $K-1$ entries with value 1 and there is one entry $y_c$ with value $1 \leq y_c \leq K-1$. W.l.o.g. we assume that $c = K$).

Create a simple directed cycle of length two between $N_K$ and $N_i$, for $1 \leq i \leq y_K - 1$.

Finally, create a directed simple cycle through the vertices $N_{y_K}, \ldots, N_K$.

First observe that in Step 1, the properties $y_k \geq 1$ and $y_k \leq \sum_{t \neq k} y_t$ are maintained all the time (since we only decrement entries that have value at least two, and since we always decrease the largest value). Also, observe that Step 1 must terminate (since every execution of the while-loop decreases the sum of the $y_i$ by two), and that the only possibility to terminate is that all $y_i$ with at most one exception have value one. By condition (T2), the value of this exception ($y_c$) is less or equal to the sum of the other values (i.e. $\leq K-1$), and this justifies the comment in the beginning of Step 2.

We claim that the algorithm indeed constructs a Eulerian graph. In the beginning the edge set is empty, and hence for every vertex its in-degree equals its out-degree. Step 1 and Step 2 only add directed cycles that increase the in-degree and the out-degree of every vertex by the same amount. Moreover, the directed cycles created in Step 2 strongly connect all vertices to $N_{y_K}$. Summarizing, the resulting graph is strongly connected, all vertices have in-degree equal to out-degree, and thus the graph is Eulerian. Moreover, it is easy to verify that in the end, vertex $N_k$ indeed has in-degree $y_k(S)$ and that no loops have been created. Concluding, the graph constructed by the above procedure satisfies the conditions (i) and (ii).

Finally, we investigate the time complexity of the procedure. In the beginning, $\sum_{i=1}^{K} y_i \leq n$ holds. Every execution of the while-loop decreases this sum by two, and thus the loop is executed $O(n)$ times. Since the $y_i$ only take integer values between 1 and $n$, they can be stored in a linear array of length $n$ where the $j$-th entry counts the number of $y_i$ that are currently equal to $j$. With this data-structure, it is easy to find the two largest values, decrease and re-insert them into the array in constant time per execution of the loop. Since Step 2 can be implemented trivially in $O(n)$ time, this yields the claimed overall time complexity of $O(n)$.

In the *second phase* of the construction of the tour $\tau_S$ , we first consider the groups $N_k$, $1 \leq k \leq K$. The cities in any group can be partitioned into four parts.

$$\begin{aligned} OI_k(S) &:= \{i \in N_k : \ a_i \in S \text{ and } b_i \in S\} \\ II_k(S) &:= \{i \in N_k : \ a_i \in S \text{ and } b_i \notin S\} \\ OO_k(S) &:= \{i \in N_k : \ a_i \notin S \text{ and } b_i \in S\} \\ IO_k(S) &:= \{i \in N_k : \ a_i \notin S \text{ and } b_i \notin S\}. \end{aligned}$$

In these sets "$O$" stands for "outside" and "$I$" for "inside". For example, the set $OI_k(S)$ consists of all $i \in N_k$ for which $b_i \in S$ (hence $\tau_S^{-1}(i) \notin N_k$ must hold in $\tau_S$) and $a_i \in S$ (hence $\tau_S(i) \in N_k$ must hold in $\tau_S$), i.e. the predecessor of $i \in N_k$ is *outside* $N_k$ and the successor of $i$ is *inside* $N_k$. The other sets are defined in a similar way. Note that some of these sets may be empty.

Combining Condition (T1) with an easy counting argument shows that $|OI_k(S)| = |IO_k(S)|$. Therefore, we can arbitrarily match the cities in $OI_k(S)$ and $IO_k(S)$ with each other.

Next label the edges of the Eulerian graph constructed in phase one in such a way that the $y_k(S)$ edges going into $N_k$ are labeled by the numbers $b_i$ in $S \cap B_k$. We compute a Eulerian circuit through the Eulerian graph and derive from this Eulerian circuit a *partial tour* that

visits the cities in the order of the indices of the labels of the edges in the Eulerian circuit. The remaining cities that do not occur in the partial tour have to be patched into this partial tour. Consider a group $N_k$. If $y_k(S) = n_k$, all cities in $N_k$ are already contained in the partial tour. Otherwise, we deduce from Condition (T4) that $OI_k(S)$ is non-empty. After every city in $OI_k(S)$, we insert its matched city from $IO_k(S)$ into the partial tour. In case the set $II_k(S)$ is non-empty, we patch all cities in $II_k(S)$ in arbitrary order between a city from $OI_k(S)$ and its matched city from $IO_k(S)$. Repeating this procedure for every group $N_k$ finally yields the desired tour $\tau_S$.

It will now be shown that the constructed tour $\tau_S$ indeed is a tour of length $d(S)$. In every group $N_k$, $1 \leq k \leq K$, all cities in $OO_k(S)$ and $OI_k(S)$ occur as index of some label and thus are visited sometime when the Eulerian circuit enters $N_k$. The above construction patches all cities in $II_k(S)$ and $IO_k(S)$ into the partial tour. Hence, $\tau_S$ visits all cities exactly once. By construction, $d(\tau_S) = d(S)$.

Computing the Eulerian circuit and patching the remaining cities into the partial tour can be easily performed in $O(n)$ time. □

The algorithm for constructing a tour from a given tour-set (as it was discussed in the proof of Lemma 3.3) is illustrated in the following example.

**Example** (continued). Let $S = \{a_2, a_4, a_7, a_8; b_2, b_3, b_4, b_7, b_9\}$, i.e. $d(S) = 476$. It is easy to check that

$$\begin{array}{lll} x_1(S) = 1 & x_2(S) = 1 & x_3(S) = 2 \\ y_1(S) = 2 & y_2(S) = 1 & y_3(S) = 2 \end{array}$$

and that $S$ is a tour-set. The construction in phase one yields a Eulerian multigraph on $\{N_1, N_2, N_3\}$ that consists of a two-cycle $N_1, N_3$ and a directed cycle $N_1, N_2, N_3$. One of the possible Eulerian cycles in this graph is labeled $b_4, b_9, b_3, b_7, b_2$ and yields a partial tour $(4, 9, 3, 7, 2)$. Furthermore,

$$\begin{array}{lll} II_1(S) = \emptyset & II_2(S) = \emptyset & II_3(S) = \{8\} \\ IO_1(S) = \{1\} & IO_2(S) = \{5\} & IO_3(S) = \{6\} \\ OI_1(S) = \{2\} & OI_2(S) = \{4\} & OI_3(S) = \{7\} \\ OO_1(S) = \{3\} & OO_2(S) = \emptyset & OO_3(S) = \{9\}\,. \end{array}$$

Hence, after City 2 we insert City 1, after City 4 we insert City 5 and after City 7 we insert Cities 8 and 6 which yields the tour $\tau_S = (4, 5, 9, 3, 7, 8, 6, 2, 1)$ with $d(\tau_S) = 476 = d(S)$. □

# 4 From Tour-Sets to Convex Functions

By Lemmas 3.2 and 3.3, the $K$–template TSP is equivalent to the problem of finding a minimum length tour-set. Instead of searching for an optimal tour, we will try to find such an optimal tour-set. Consider the set $Y$ that is defined as follows:

$$Y = \{[y_1, \ldots, y_K] : \;\; y_k \leq \textstyle\sum_{t \neq k} y_t,\; 1 \leq y_k \leq n_k,\; y_k \text{ integers},\; k = 1, \ldots, K\}.$$

For any vector $y = [y_1, \ldots, y_k] \in Y$, one may define values $x_k := n_k - y_k$ for $1 \leq k \leq K$. This yields numbers $x_k$ and $y_k$, $1 \leq k \leq K$, that obviously satisfy Conditions (T1)–(T3) in

Definition 3.1. It is easy to see that there always exists a tour-set $S$ such that $x_k(S) = x_k$ and $y_k(S) = y_k$ hold for all $k$. Since we are interested in a tour-set of minimum length, the following problem arises.

> Given a vector $y = [y_1, \ldots, y_K] \in Y$, determine a tour-set $S_y$ such that $y_k(S_y) = y_k$ for $k = 1, \ldots, K$ and such that $d(S_y) \leq d(S)$ for any tour-set $S$ with $y_k(S) = y_k$ for $k = 1, \ldots, K$.

We describe a simple algorithm for solving this problem.

**Step 0** Compute $x_k := n_k - y_k$ for $k = 1, \ldots, K$. Set $k := 1$.

**Step 1** If $x_k = 0$ then set $S_k := \{b_i : i \in N_k\}$ and go to Step 4, else go to Step 2.

**Step 2** Let the set $S_k$ consist of the $x_k$ smallest elements in $A_k$ and the $y_k$ smallest elements in $B_k$, i.e.

$$S_k := \{a_{\sigma_k(m_k+1)}, \ldots, a_{\sigma_k(m_k+x_k)}\} \cup \{b_{\rho_k(m_k+1)}, \ldots, b_{\rho_k(m_k+y_k)}\}.$$

If there is an $i \in N_k$ such that both $a_i \in S_k$ and $b_i \in S_k$ (i.e. Condition (T4) in Definition 3.1 is satisfied) then go to Step 4, else go to Step 3.

**Step 3** Compute the values

$$\begin{aligned} \delta_a &:= a_{\sigma_k(m_k+x_k+1)} - a_{\sigma_k(m_k+x_k)} \quad \text{and} \\ \delta_b &:= b_{\rho_k(m_k+y_k+1)} - b_{\rho_k(m_k+y_k)}. \end{aligned}$$

If $\delta_a \leq \delta_b$ then update

$$S_k := \left(S_k \setminus \left\{a_{\sigma_k(m_k+x_k)}\right\}\right) \cup \left\{a_{\sigma_k(m_k+x_k+1)}\right\}$$

else update

$$S_k := \left(S_k \setminus \left\{b_{\rho_k(m_k+y_k)}\right\}\right) \cup \left\{b_{\rho_k(m_k+y_k+1)}\right\}.$$

Go to Step 4.

**Step 4** If $k = K$ then define the output set $S_y := \bigcup_{k=1}^{K} S_k$ , else return to Step 1 with $k := k + 1$.

**Example** (continued). We start with $y = [2, 2, 1] \in Y$. In Step 0, we find $x_1 = 1$, $x_2 = 0$ and $x_3 = 3$. The tour-set $S$ is now computed as follows.

**(k=1)** In Step 2, we find $S_1 = \{a_1\} \cup \{b_2, b_3\}$, i.e. there is no $i \in N_1 = \{1, 2, 3\}$ such that both $a_i$ and $b_i$ are in $S_1$. In Step 3 we find $\delta_a = 33 > 19 = \delta_b$ and hence $S_1 = \{a_1\} \cup \{b_1, b_2\}$.

**(k=2)** Since $x_2 = 0$ we have $S_2 = \{b_4, b_5\}$.

**(k=3)** From Step 2 we have $S_3 = \{a_7, a_8, a_9\} \cup \{b_9\}$. Both $a_9$ and $b_9$ are in $S_3$.

So, $S = \{a_1, a_7, a_8, a_9, \} \cup \{b_1, b_2, b_4, b_5, b_9\}$ and $d(S) = 526$. □

It is convenient to denote the output set of the above algorithm that is assigned to an input vector $y \in Y$ by $S_y$. The following lemma shows that the above algorithm solves the problem of finding a minimum length tour-set for a given vector $y \in Y$ to optimality.

**Lemma 4.1** *For every $y = [y_1, \ldots, y_K] \in Y$, the set $S_y$ is a tour-set. Moreover, for any tour-set $S$ satisfying $y_k(S) = y_k$ for $k = 1, \ldots, K$ it holds that $d(S_y) \leq d(S)$.*

Proof.

First we show that $S_y$ satisfies Conditions (T1)–(T4). Conditions (T2) and (T3) follow by the definition of $Y$. Condition (T1) is satisfied by the definition of $x_k$ in Step 0. Step 3 guarantees, due to $x_k + y_k = n_k$, that if $x_k > 0$ then $\{i : a_i \text{ and } b_i \in S_k\} \neq \emptyset$. Hence, (T4) is also satisfied.

To see that $d(S_y)$ is the minimum length tour-set that satisfies $y_k(S) = y_k$, observe that each group $N_k$ can be handled separately. Obviously, in each set $V_k$ the smallest $n_k$ elements that satisfy Condition (T4) in Definition 3.1 are taken into $S_y$. □

Let $f_k(y_k)$ denote the total length of the $n_k$ elements in $S_y \cap V_k$ . Since the above algorithm for computing $S_y$ is relatively simple, it is possible to explicitly write down the values $f_k(y_k)$ for $1 \leq y_k \leq n_k$ and $1 \leq k \leq K$. This is done as follows:

$$f_k(y_k) := \; b_{\rho_k(m_k+1)} + \ldots + b_{\rho_k(m_k+y_k)} + a_{\sigma_k(m_k+1)} + \ldots + a_{\sigma_k(m_k+n_k-y_k)} \tag{1}$$

if

$$\{\rho_k(m_k+1), \ldots, \rho_k(m_k+y_k)\} \cap \{\sigma_k(m_k+1), \ldots, \sigma_k(m_k+n_k-y_k)\} \neq \emptyset \tag{2}$$

and

$$\begin{aligned} f_k(y_k) := \quad & b_{\rho_k(m_k+1)} + \ldots + b_{\rho_k(m_k+y_k)} + a_{\sigma_k(m_k+1)} + \ldots + a_{\sigma_k(m_k+n_k-y_k)} + \\ & + \min \left\{ \; b_{\rho_k(m_k+y_k+1)} - b_{\rho_k(m_k+y_k)} \; ; \; a_{\sigma_k(m_k+n_k-y_k+1)} - a_{\sigma_k(m_k+n_k-y_k)} \; \right\} \end{aligned} \tag{3}$$

otherwise.

For ease of reference, the condition stated in (2) is called Condition (NE). Since each group $N_k$ can be handled separately, we have the following observation.

**Observation 4.2** *For every $y = [y_1, \ldots, y_K] \in Y$, the tour-set $S_y$ fulfills*

$$d(S_y) = \sum_{k=1}^{K} f_k(y_k).$$

□

**Example** (continued). The values for $f_k(y_k)$ are:

| | | |
|---|---|---|
| $f_1(1) = 124$ | $f_2(1) = 132$ | $f_3(1) = 238$ |
| $f_1(2) = 130$ | $f_2(2) = 158$ | $f_3(2) = 195$ |
| $f_1(3) = 150$ | | $f_3(3) = 174$ |
| | | $f_3(4) = 163.$ |

So, from $y = [2, 2, 1]$ we obtain $f_1(2) + f_2(2) + f_3(1) = 526$, which is exactly the length of the tour-set $S_y$ that we computed above for the input vector $[2, 2, 1]$. □

In the remainder of this section, we investigate the combinatorial structure of the functions $f_k(\cdot)$ and we show how to compute them efficiently. We start with the following definition for convexity of functions on ordered discrete domains. A function $h : \{1, 2, \ldots, n\} \to \mathbb{R}$ is called *convex*, if for all $i \in \{2, \ldots, n-1\}$, the inequality

$$h(i) - h(i-1) \;\leq\; h(i+1) - h(i)$$

holds. The following lemma reveals the structure of the functions $f_k(\cdot)$.

**Lemma 4.3** *For any $k \in \{1, \ldots, K\}$, the function $f_k$ is convex on $\{1, \ldots, n_k\}$.*

Proof.

We will prove the convexity of $f_1$ on $\{1, \ldots, n_1\}$. The arguments for $k = 2, \ldots, K$ are analogous. To ease the notation, we drop the index 1 from $f_1$ and $n_1$ and simply write $f$ and $n$ instead.

Note that if $n \leq 2$, then the lemma trivially holds. Hence, assume $n \geq 3$. Define $\Delta(i) := 2f(i) - f(i-1) - f(i+1)$ for $i = 2, \ldots, n-1$. Clearly, our goal is to shown that $\Delta(i) \leq 0$ for all $i$. Define

$$A(i) := \sum_{j=1}^{n-i-1} a_{\sigma(j)} \text{ and } B(i) := \sum_{j=1}^{i-1} b_{\rho(j)}.$$

According to the definition of functions $f_k(\cdot)$, we call the function $f$ to be in *State I* at $j$ if Condition (NE) holds, i.e. if

$$\{\rho(1), \ldots, \rho(j)\} \cap \{\sigma(1), \ldots, \sigma(n-j)\} \neq \emptyset,$$

and to be in *State II* at $j$, otherwise.

Observe that the function $f$ takes higher (or equal) value in State II than the value it would take in State I at the same point. It is clear that in order to prove the convexity relation the most stringent case is if $f$ is in State I at $i-1$ and at $i+1$, and it is in State II at $i$. In that case we find:

$$\begin{aligned} f(i-1) &= A(i) + B(i) + a_{\sigma(n-i)} + a_{\sigma(n-i+1)} \\ f(i) &= A(i) + B(i) + a_{\sigma(n-i)} + b_{\rho(i)} + \min\{b_{\rho(i+1)} - b_{\rho(i)}, a_{\sigma(n-i+1)} - a_{\sigma(n-i)}\} \\ f(i+1) &= A(i) + B(i) + b_{\rho(i)} + b_{\rho(i+1)}. \end{aligned}$$

Therefore,

$$\begin{aligned} \Delta(i) \;=\; & a_{\sigma(n-i)} + b_{\rho(i)} - a_{\sigma(n-i+1)} - b_{\rho(i+1)} \\ & + 2\min\{b_{\rho(i+1)} - b_{\rho(i)}, a_{\sigma(n+i+1)} - a_{\sigma(n-i)}\}. \end{aligned}$$

If $b_{\rho(i+1)} - b_{\rho(i)} \leq a_{\sigma(n+i+1)} - a_{\sigma(n-i)}$, we find

$$\Delta(i) = (b_{\rho(i+1)} - b_{\rho(i)}) - (a_{\sigma(n+i+1)} - a_{\sigma(n-i)}) \leq 0$$

and if $b_{\rho(i+1)} - b_{\rho(i)} > a_{\sigma(n+i+1)} - a_{\sigma(n-i)}$, we get

$$\Delta(i) = (a_{\sigma(n+i+1)} - a_{\sigma(n-i)}) - (b_{\rho(i+1)} - b_{\rho(i)}) < 0.$$

Hence, even in the most stringent case we have $\Delta(i) \leq 0$ for all $i \in \{2, \ldots, n-1\}$. Since the other ("easier") cases follow by analogous arguments, the lemma is proved. □

We now show how the functions $f_k(y_k)$ can be computed efficiently.

**Lemma 4.4** *From the vectors $a$ and $b$, all values $f_k(y_k)$ with $1 \leq k \leq K$ and $1 \leq y_k \leq n_k$ can be computed in $O(n \log n)$ overall time.*

Proof

It is sufficient to show how to compute all values $f_1(y_1)$ with $1 \leq y_1 \leq n_1$ in $O(n_1 \log n_1)$ overall time. Then the statement of the lemma follows by analogous calculations for the functions $f_k$ with $2 \leq k \leq K$.

First we compute in $O(n_1 \log n_1)$ time permutations $\sigma$ and $\rho$ such that $a_{\sigma(1)} \leq \ldots \leq a_{\sigma(n_1)}$ and $b_{\rho(1)} \leq \ldots \leq b_{\rho(n_1)}$ hold. Then we determine for every $y_1$ $(1 \leq y_1 \leq n_1)$ whether or not Condition (NE) is fulfilled for $y_1$. This is done as follows. We start with $y_1 = 1$ and consider the set

$$\{\ b_{\rho(1)}\ \} \cup \{\ a_{\sigma(1)}, \ldots, a_{\sigma(n_1-1)}\ \}.$$

For every city $i$, we introduce a counter that counts the number of entries from vectors $a$ and $b$ that appear in the above set and correspond to city $i$ (obviously, such a counter only may take the values zero, one and two). Moreover, we introduce three numbers $q_\ell$, $0 \leq \ell \leq 2$, where number $q_\ell$ stores the number of counters that currently take value $\ell$. Clearly, Condition (NE) is fulfilled if and only if $q_2 \geq 1$. When we move on from $y_1$ to $y_1 + 1$, one $a$-entry is removed from and one $b$-entry is added to the above set. The corresponding two counters and the numbers $q_\ell$ can be updated in constant time. We keep on increasing the value $y_1$, updating the counters and $q_\ell$, and in the end we know after an overall $O(n_1)$ number of operations whether or not Condition (NE) is fulfilled for each $y_1 \in \{1, \ldots, n_1\}$.

Then we compute in $O(n_1)$ time the $2n_1$ numbers $A(j)$ and $B(j)$ for $1 \leq j \leq n_1$ by

$$A(j) = \sum_{i=1}^{j} a_{\sigma(i)} \qquad \text{and} \qquad B(j) = \sum_{i=1}^{j} b_{\rho(i)},$$

i.e. $A(j)$ and $B(j)$ are the sums of the $j$ smallest values in vector $a$ , respectively vector $b$. From the values $A(j)$ and $B(j)$, every value $f_1(y_1)$ can be computed in constant time according to equations (1 ) and (3) depending on whether or not Condition (NE) is fulfilled for $y_1$. □

# 5 An Integer Program and its Solution

We now continue with the solution of the $K$–template TSP. By Lemmas 3.2 and 3.3, the $K$–template TSP is equivalent to the problem of finding a minimum length tour-set, which we now may formulate as

$$\min\{d(S_y) :\ y \in Y\}.$$

By Observation 4.2, this problem in turn is equivalent to solving the following integer program.

$$\textbf{(IP)} \qquad \begin{array}{ll} \min & \sum_{k=1}^{K} f_k(y_k) \\ \text{s.t.} & \begin{cases} 2y_k \leq \sum_{t=1}^{K} y_t & k = 1, \ldots, K, \\ 1 \leq y_k \leq n_k \text{ and } y_k \text{ integers}, & k = 1, \ldots, K. \end{cases} \end{array}$$

In order to solve the integer program **(IP)** efficiently, we parameterize it by introducing a new integer variable $\lambda$ that equals the sum of all values $y_k$. This yields $(n - K + 1)$ integer programs $\mathbf{(P(\lambda))}$ for $\lambda = K, \ldots, n$ that are defined as follows.

$$\mathbf{(P(\lambda))} \qquad \begin{array}{ll} \min & \sum_{k=1}^{K} f_k(y_k) \\ \text{s.t.} & \begin{cases} \sum_{k=1}^{K} y_k = \lambda \\ 1 \leq y_k \leq \min\left\{\frac{\lambda}{2}, n_k\right\} \text{ and } y_k \text{ integers}, & k = 1, \ldots, K \end{cases} \end{array}$$

For $K \leq \lambda \leq n$, we denote by $p^*(\lambda)$ the optimum solution value of $\mathbf{(P(\lambda))}$. In case $\mathbf{(P(\lambda))}$ does not possess feasible solutions, $p^*(\lambda)$ is set to $+\infty$. For finite $p^*(\lambda)$, we denote by $y^*(\lambda)$ the corresponding solution vector. The following lemma shows that from the solutions to the Problems $\mathbf{(P(\lambda))}$ for all $\lambda$, we can find the solution of Problem **(IP)**.

**Lemma 5.1** *The integer program* **(IP)** *possesses at least one feasible solution, and hence its optimum value is finite. This optimum value equals*

$$\min\{\ p^*(\lambda)\ :\ K \leq \lambda \leq n\ \}$$

Proof.
The vector $y$ with $y_k = 1$ for $1 \leq k \leq K$ clearly constitutes a feasible solution for **(IP)** and for $\mathbf{(P(}$ K$\mathbf{))}$. Moreover, since $1 \leq y_k \leq n_k$ must hold for all $y_k$, we derive $K \leq \sum_{k=1}^{K} y_k \leq n$ for their sum. Hence, $K \leq \lambda \leq n$ indeed covers all possible values. □

Observe that $\mathbf{(P(\lambda))}$ does not always allow feasible solutions. Consider e.g. the case where $n = 14$, $K = 3$, $n_1 = 2$, $n_2 = 2$ and $n_3 = 10$ hold. Then the variables $y_1$, $y_2$ and $y_3$ in program $\mathbf{(P(}$ 10$\mathbf{))}$ must fulfill $y_1 \leq 2$, $y_2 \leq 2$ and $y_3 \leq 5$. Hence, their sum is at most 9, which contradicts the restriction $y_1 + y_2 + y_3 = 10$.

In the following three lemmas, we will derive several properties of the programs $\mathbf{(P(\lambda))}$ that will later be used to design an efficient solution algorithm for these integer programs.

**Lemma 5.2** *For $K = 2$, the integer programs* $\mathbf{(P(\lambda))}$ *behave as follows.*

*If $\lambda$ is odd or if $\min\{n_1, n_2\} < \frac{\lambda}{2}$ holds, then* $\mathbf{(P(\lambda))}$ *does not have any feasible solution.*

*If $\lambda$ is even and $\min\{n_1, n_2\} \geq \frac{\lambda}{2}$ holds, then the unique feasible solution of* $\mathbf{(P(\lambda))}$ *is given by $y_1 = y_2 = \frac{\lambda}{2}$.* □

The proof of this lemma is trivial and therefore omitted.

**Lemma 5.3** *Let $K \geq 3$ and $\lambda \in \{K + 1, \ldots, n\}$.*
*If the program* $\mathbf{(P(\lambda))}$ *has feasible solutions then the following statements hold:*

*(i) In any feasible solution $y$ of $(\mathbf{P}(\lambda))$ there is at most one $y_k$ with $y_k > \frac{\lambda-1}{2}$.*

*(ii) The program $(\mathbf{P}(\lambda-1))$ also has feasible solutions.*

PROOF.

Consider a feasible solution vector $y$ for $(\mathbf{P}(\lambda))$, i.e. the $y_k$ are integers with $1 \leq y_k \leq \min\{\frac{\lambda}{2}, n_k\}$ and $\sum_{k=1}^{K} y_k = \lambda$.

(i) An integer greater than $\frac{\lambda-1}{2}$ implies that it is at least $\frac{\lambda}{2}$. If there are two components $y_k$ larger or equal to $\frac{\lambda}{2}$ then, together with a third component (that exists as $K \geq 3$), the sum of these three components would be at least $\lambda + 1$, which gives a contradiction.

(ii) Statement (i) yields that if we decrement the largest $y_k$ by one and leave all other entries of $y$ unchanged, then we arrive at a feasible solution for $(\mathbf{P}(\lambda-1))$. □

**Lemma 5.4** *Assume $K \geq 3$, and let $\lambda \in \{K+1, \ldots, \leq n\}$ be such that $(\mathbf{P}(\lambda))$ possesses a feasible solution.*
*Then for every optimum solution vector $z = [z_1, \ldots, z_K]$ for the program $(\mathbf{P}(\lambda-1))$, there exists an optimum solution vector $y = [y_1, \ldots, y_K]$ for the program $(\mathbf{P}(\lambda))$ such that*

$$y_k = \begin{cases} z_k & \text{if } k \neq j \\ z_j + 1 & \text{if } k = j \end{cases}$$

*for some index $j \in \{1, ..., K\}$.*

PROOF.

Let $y$ be an optimum solution vector for $(\mathbf{P}(\lambda))$ that is "closest" to $z$ in the sense that $y$ minimizes $\sum_{i=1}^{K} |z_i - y_i|$ (i.e. the Manhattan distance from $y$ to vector $z$) among all optimum solution vectors $y$ for $(\mathbf{P}(\lambda))$. Note that in order to prove the lemma, it suffices to show that $\sum_{i=1}^{K} |z_i - y_i| = 1$.

Suppose otherwise. By a simple parity argument, $\sum_{i=1}^{K} |z_i - y_i|$ is odd and hence has value at least three. This immediately yields the existence of an index $t \in \{1, \ldots K\}$ with $z_t > y_t$. Moreover, we will define another index $s \in \{1, \ldots K\}$ with $z_s < y_s$ in the following way. If there exists an index $s$ with $z_s < y_s \leq \frac{\lambda-1}{2}$, we choose this index $s$. Otherwise, there is a unique index $s$ with $z_s < y_s$ and $y_s > \frac{\lambda-1}{2}$. Since the Manhattan distance between $z$ and $y$ is at least three, it follows that in this case $z_s \leq y_s - 2$. Summarizing, in both cases we have

$$z_s < y_s \quad \text{and} \quad z_s + 1 \leq \frac{\lambda - 1}{2} \quad \text{and} \quad z_t > y_t.$$

Now construct two new vectors $y'$ and $z'$ that are based on $y$ and $z$ as follows:

$$y'_k = \begin{cases} y_k & \text{for } k \in \{1, \ldots, K\} \setminus \{s, t\} \\ y_s - 1 & \text{for } k = s \\ y_t + 1 & \text{for } k = t. \end{cases}$$

and

$$z'_k = \begin{cases} z_k & \text{for } k \in \{1, \ldots, K\} \setminus \{s, t\} \\ z_s + 1 & \text{for } k = s \\ z_t - 1 & \text{for } k = t. \end{cases}$$

Clearly by the definition of $z_s$, $z_t$, $y_s$ and $y_t$, vector $y'$ is feasible for $\mathbf{(P(\lambda))}$ and vector $z'$ is feasible for $\mathbf{(P(\lambda-1))}$. When going from $y$ to $y'$, the value of the objective function in $\mathbf{(P(\lambda))}$ cannot decrease since it is assumed that $y$ is an optimum solution, i.e.

$$f_s(y_s-1) - f_s(y_s) \;+\; f_t(y_t+1) - f_t(y_t) \;\geq\; 0. \tag{4}$$

By analogous considerations for $\mathbf{(P(\lambda-1))}$, we derive

$$f_s(z_s+1) - f_s(z_s) \;+\; f_t(z_t-1) - f_t(z_t) \;\geq\; 0. \tag{5}$$

Since by Lemma 4.3 the functions $f_s$ and $f_t$ are convex, it follows from $z_s+1 \leq y_s$ and $y_t+1 \leq z_t$ that

$$f_s(z_s+1) - f_s(z_s) \;\leq\; f_s(y_s) - f_s(y_s-1) \tag{6}$$
$$f_t(y_t+1) - f_t(y_t) \;\leq\; f_t(z_t) - f_t(z_t-1). \tag{7}$$

By adding (4) to (5) and comparing the resulting inequality to the sum of (6) and (7), one sees that in all four inequalities (4) through (7) actually *equality* must hold. This implies that $y'$ is also optimal for $\mathbf{(P(\lambda))}$ and that $z'$ is also optimal for $\mathbf{(P(\lambda-1))}$. Since $y'$ has smaller Manhattan distance to $z$ than $y$, we arrive at a contradiction to the definition of $y$ and the proof is complete. □

The above lemma suggests the following procedure for sequentially solving all problems $\mathbf{(P(\lambda))}$ for $\lambda = K, \ldots, n$.

**Step 0** In case $K = 2$ holds, compute the minimum of $f_1(\alpha) + f_2(\alpha)$ for $1 \leq \alpha \leq \min\{n_1, n_2\}$ and let $\alpha_0$ denote the corresponding $\alpha$. The optimum solution value of **(IP)** is given by $y^* = [\alpha_0, \alpha_0]$ with value $f_1(\alpha_0) + f_2(\alpha_0)$. Stop.

**Step 1** Otherwise, $K \geq 3$ holds.

Initialize $y_k := 1$ and define $f_k(n_k+1) := +\infty$ for $k = 1, \ldots K$.

Compute the values $\delta_k := f_k(2) - f_k(1)$ for $k = 1, \ldots K$.

Set $p^*(K) := \sum_{k=1}^{K} f_k(1)$.

**Step 2** Perform the following procedure for $\lambda = K+1, \ldots, n$ (Note that every time the procedure is executed, the sum of the $y_k$ increases by one):

Determine the index $t$ for which $\delta_t$ currently is minimum. If $y_t + 1 > \frac{\lambda}{2}$ holds, let $t$ be the index of the second-smallest $\delta_t$.

Set $p^*(\lambda) = p^*(\lambda-1) + \delta_k$.

Update $\delta_t := f_t(y_t+1) - f_t(y_t)$.

Finally, increment $y_t$ by one.

We argue that the in algorithm described above, $p^*(\lambda)$ equals the optimum solution value of $\mathbf{(P(\lambda))}$ for all $K \leq \lambda \leq n$.

Step 0 correctly solves the case $K = 2$ according to Lemma 5.2. Step 1 initializes all involved numbers for $\lambda = K$. Defining $f_k(n_k+1) = +\infty$ causes that in case the $y_k$ try to

pass the borderline $n_k$, then the value of the objective function jumps to $+\infty$. The variables $\delta_t$ always store the change in the objective function in case $y_t$ is increased by one. Step 2 essentially applies Lemma 5.4. Every solution for $\mathbf{(P(\lambda-1))}$ can be extended to a solution for $\mathbf{(P(\lambda))}$ by increasing one of the $y_t$ (taking for granted that $\mathbf{(P(\lambda))}$ has a finite solution). The cheapest way to extend a solution is incrementing the $y_t$ with smallest $\delta_t$ value. In case this increment is infeasible (since it would make $y_k$ exceed the bound $\frac{\lambda}{2}$), the second-cheapest increment is chosen. Note that by Lemma 5.3(i), the second-cheapest increment can never violate the bound $\frac{\lambda}{2}$. In case the chosen increment violates the condition $y_t \leq n_t$, program $\mathbf{(P(\lambda))}$ does not possess a feasible solution and by Lemma 5.3(ii), the same holds for all larger values of $\lambda$.

We now investigate the time complexity of this algorithm. The most "expensive" part is the handling of the $\delta_k$ values. If they are stored in a balanced search tree, then finding the minimum element, finding the second-smallest element, deleting and reinserting the updated $\delta_k$ values, all take $O(\log K)$ time per operation. Since the procedure in Step 2 is performed $O(n)$ times, this yields an overall time complexity of $O(n \log K)$.

Note that we cannot store or output with every value $p^*(\lambda)$ the corresponding solution vector $y^*(\lambda)$, since this could cost up to $O(nK) = O(n^2)$ operations. However, we are mainly interested in the optimal solution value and in the corresponding solution vector of problem **(IP)**, which by Observation 5.1 coincides with the optimal solution value and corresponding solution vector of the $\mathbf{(P(\lambda))}$ with minimum $p^*(\lambda)$. Hence, we can simply perform the whole algorithm *twice*. The first time in order to determine the smallest $p^*(\lambda)$, and the second time in order to output the solution vector $y^*(\lambda)$.

Summarizing, we have proven the following lemma.

**Lemma 5.5** *If all values $f_k(y_k)$ with $1 \leq k \leq K$ and $1 \leq y_k \leq n_k$ are given, the optimum solution value and the corresponding solution vector $y^*$ for* **(IP)** *can be computed in $O(n \log K)$ time.* □

**Example** (continued). Applying the above algorithm yields the following values.

| $\lambda$ | $p^*(\lambda)$ | $y^*(\lambda)$ |
|---|---|---|
| 3 | 494 | [1,1,1] |
| 4 | 451 | [1,1,2] |
| 5 | 457 | [2,1,2] |
| 6 | 436 | [2,1,3] |
| 7 | 456 | [3,1,3] |
| 8 | 445 | [3,1,4] |
| 9 | 471 | [3,2,4] |

Note that for the sake of demonstration we invested $O(nK)$ time by computing *all* solution vectors $y^*(\lambda)$). □

# 6 Proof of the Main Result and an Application

We combine the results that have been derived in the preceding three sections in our main theorem.

**Theorem 6.1** *The $K$–template TSP on $n$ cities ($K \leq n$) is solvable in $O(n \log n)$ time.*

Proof.

By the Lemmas 3.2, 3.3, and 4.1 and by Observation 4.2, the length of an optimal tour is equal to

$$\min \left\{ \sum_{k=1}^{K} f_k \left( y_k^*(\lambda) \right) : \lambda = K, \ldots, n \right\}.$$

First, we compute all values $f_k(y_k)$ with $1 \leq k \leq K$ and $1 \leq y_k \leq n_k$ according to Lemma 4.4 in $O(n \log n)$ overall time. Then we apply Lemma 5.5 and find the above minimum and the corresponding solution vector $y^*(\lambda)$ in $O(n \log K)$ time. As shown in the proof of Lemma 3.3, constructing an optimal tour from the optimal tour-set that corresponds to $y^*(\lambda)$ also can be done in time $O(n)$. Since $K \leq n$ holds, this results in an overall time complexity of $O(n \log n)$. □

**Example** (continued). An optimal vector is $y^* = [2, 1, 3]$. The corresponding optimal tour-set (as determined by the algorithm in Lemma 3.3) is $S_{y^*} = \{a_1, a_4, a_7\} \cup \{b_1, b_2, b_4, b_6, b_7, b_9\}$ with $d(S_{y^*}) = 436$. An optimal tour is $\tau_{S_{y^*}} = (1, 3, 6, 2, 7, 8, 4, 5, 9)$ with $d(\tau_{S_{y^*}}) = 436$. □

We conclude this section with an application / generalization of the $K$–template TSP. As before it is assumed that part of the input is given by $n$, $K$ and the partition of the jobs $N_1, \ldots, N_K$. Consider a one machine scheduling problem where the change-over time between two jobs is determined by the removal time of the processed job plus the set-up time of the next job and that the objective is to minimize makespan. Clearly, if all jobs belong to one group and the removal time and set-up time are job-dependent, any sequence of the jobs is optimal (no matter what the sequence is, each job is to be set-up and removed). The corresponding TSP is called the *constant TSP* (see Gilmore *et al.* [8]) because each tour has the same length. Now assume that the change-over time is not only determined by the set-up time plus the removal time but also by the fact whether or not the two jobs belong to the same job-class. More precisely, assume that, if job $J_j$ is scheduled directly after job $J_i$, the change-over time is $p_i + q_j$ if both jobs belong to the same class and $r_i + s_j$ otherwise, where $p, q, r$ and $s$ are given $n$-dimensional vectors. Clearly, the distance matrix $D = (d[i, j])$ of the corresponding TSP is given by

$$d\,[i, j] = \begin{cases} p_i + q_j & \text{if } i \text{ and } j \text{ belong to the same group} \\ r_i + s_j & \text{if } i \text{ and } j \text{ belong to different groups.} \end{cases}$$

It will be shown that this TSP can be reformulated as a $K$–template TSP. Define $a_i := p_i - r_i$ and $b_i := s_i - q_i$ for $i = 1, \ldots, n$. Furthermore, define

$$d'[i, j] := d[i, j] - r_i - q_j$$

for $i, j = 1, \ldots, n$. Since subtracting constants from rows and columns does not change the ordering of the tours according to length, solving the TSP with distance matrix $D = (d[i, j])$

is equivalent to solving the TSP with distance matrix $D' = (d'[i,j])$. It is easy to see that the TSP with distance matrix $D' = (d'[i,j])$ is exactly the $K$–template TSP. Therefore, we conclude that the TSP with the distance matrix given by $D = (d[i,j])$ is solvable in $O(n \log n)$ time.

# 7 Conclusions

In this paper we have shown that the *$K$–template TSP* is solvable in polynomial time. The input of the $K$–template TSP consists of $n$, $K$, two $n$-dimensional vectors $a$ and $b$ and a partition of the jobs $\{1, \ldots, n\}$ into $K$ groups $N_1, \ldots, N_K$. Every entry of the distance matrix can be computed from this data in constant time.

Our solution algorithm consisted of three parts. First, we established a connection between the $K$–template TSP and certain subsets (the so-called tour-sets) of the entries in vectors $a$ and $b$. Secondly, we defined and investigated certain convex functions for the tour-sets. Thirdly, we formulated the $K$–template TSP as an appropriate integer program involving these related convex functions. The convexity of the functions allowed us to design a fast $O(n \log n)$ solution procedure for the integer program.

**Acknowledgments.** Three anonymous referees are kindly acknowledged for their helpful comments leading to an improvement of the paper. We would also like to thank Bettina Klinz for bringing the authors together, for pointing out an error in an earlier version of the paper and for several helpful comments.

This research has been partially supported by the Spezialforschungsbereich F 003 "Optimierung und Kontrolle", Projektbereich Diskrete Optimierung.

# References

[1] B.H. Ahn and J.H. Hyun: Single Facility Multi-Class Job Scheduling. *Computers and Operations Research* 17 (1990) 265–272.

[2] M.O. Ball and M.J. Magazine: Sequencing of Insertions in Printed Circuit Board Assembly. *Operations Research* 36 (1988) 192–201.

[3] L. Bianco, G. Rinaldi and A. Sassano: A Combinatorial Optimization Approach to Aircraft Sequencing Problem. In: A.R. Odini et al. (eds.): Flow Control of Congested Networks. *NATO-ASI Series* 38 (1987) 324–339.

[4] L. Bianco, S. Ricciardelli, G. Rinaldi and A. Sassano: Scheduling Tasks with Sequence Dependent Processing Times. *Naval Research Logistics* 35 (1988) 177–184.

[5] R.E. Burkard, V.G. Deineko, R. van Dal, J.A.A. van der Veen and G.J. Woeginger: Solvable Cases of the Traveling Salesman Problem: A Survey. *In preparation.*

[6] S.S. Cosmadakis and C.H. Papadimitriou: The Traveling Salesman Problem with Many Visits to Few Cities. *SIAM Journal on Computing* 13 (1984) 99–108.

[7] P.C. Gilmore and R.E. Gomory: Sequencing a One-State-Variable Machine: A Solvable Case of the Traveling Salesman Problem. *Operations Research* 12 (1964) 655–679.

[8] P.C. Gilmore, E.L. Lawler and D.B. Shmoys: Well-Solved Special Cases. In: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys (eds.): *The Traveling Salesman Problem.* Wiley-Interscience (1985) 87–143.

[9] J.N.D. Gupta: Single Facility Scheduling with Multiple Job Classes. *European Journal of Operational Research* 8 (1988) 42–45.

[10] J.N.D. Gupta, J.C. Ho and J.A.A. van der Veen: Single Machine Bi-criteria Scheduling with Customer Orders and Multiple Job Classes. To appear in: *Annals of Operations Research.*

[11] H.W. Hamacher: Combinatorial Optimization Problems Motivated by Robotic Assembly Problems. In: M. Akgul (Ed.): Combinatorial Optimization. *NATO ASI Series* F82 (1992) 187–198.

[12] C.L. Monma and C.N. Potts: On the Complexity of Scheduling with Batch Setup Times. *Operations Research* 37 (1989) 798–804.

[13] C.N. Potts: Scheduling Two Job Classes on a Single Machine. *Computers and Operations Research* 18 (1991) 411–415.

[14] H.N. Psaraftis: A Dynamic Programming Approach for Sequencing Groups of Identical Jobs. *Operations Research* 28 (1980) 1347–1359.

[15] R. van Dal: *Special Cases of the Traveling Salesman Problem.* Ph.D.-thesis University of Groningen (1992). Wolters-Noordhoff Groningen.

[16] J.A.A. van der Veen: *Solvable Cases of the Traveling Salesman Problem with Various Objective Functions.* Ph.D.-thesis University of Groningen (1992).

[17] J.A.A. van der Veen and S. Zhang: Low-Complexity Algorithms for Sequencing Jobs with a Fixed Number of Job-Classes. *Nijenrode Working Paper Series* No. 1(1995).