A time- and space-optimal algorithm for the many-visits TSP*

André Berger† László Kozma‡ Matthias Mnich§ Roland Vincze¶

Abstract

The many-visits traveling salesperson problem (MV-TSP) asks for an optimal tour of n cities that visits each city c a prescribed number k_c of times. Travel costs may be asymmetric, and visiting a city twice in a row may incur a non-zero cost. The MV-TSP problem finds applications in scheduling, geometric approximation, and Hamiltonicity of certain graph families.

The fastest known algorithm for MV-TSP is due to Cosmadakis and Papadimitriou (SICOMP, 1984). It runs in time $n^{O(n)} + O(n^3 \log \sum_c k_c)$ and requires $n^{O(n)}$ space. The interesting feature of the Cosmadakis-Papadimitriou algorithm is its *logarithmic* dependence on the total length $\sum_c k_c$ of the tour, allowing the algorithm to handle instances with very long tours, beyond what is tractable in the standard TSP setting. However, its *superexponential* dependence on the number of cities in both its time and space complexity renders the algorithm impractical for all but the narrowest range of this parameter.

In this paper we significantly improve on the Cosmadakis-Papadimitriou algorithm, giving an MV-TSP algorithm that runs in time $2^{O(n)}$, i.e. *single-exponential* in the number of cities, with *polynomial* space. The space requirement of our algorithm is (essentially) the size of the output, and assuming the Exponential-time Hypothesis (ETH), the time requirement is optimal. Our algorithm is deterministic, and arguably both simpler and easier to analyse than the original approach of Cosmadakis and Papadimitriou. It involves an optimization over directed spanning trees and a recursive, centroid-based decomposition of trees.

1 Introduction

The traveling salesperson problem (TSP) is one of the cornerstones of combinatorial optimization, with origins going back (at least) to the 19th century work of Hamilton (for surveys on the rich history, variants, and current status of TSP we refer to the dedicated books [31, 19, 10, 3]). In the standard TSP, given n cities and their pairwise distances, we seek a tour of minimum total distance that visits each city. If the distances obey the triangle inequality, then an optimal tour necessarily visits each city *exactly* once (apart from returning to the starting city in the end). In the general case of the TSP with arbitrary distances, the optimal tour may visit a city multiple times. (Instances with non-metric distances arise from various applications that are modeled by the TSP, e.g. from scheduling problems.)

To date, the fastest known exact algorithms for TSP (both in the metric and non-metric cases) are due to Bellman [6] and Held and Karp [20], running in time $2^n \cdot O(n^2)$ for n-city instances; both algorithms also require space $\Omega(2^n)$.

In this paper we study the more general problem where each city has to be visited *exactly* a given number of times. More precisely, we are given a set V of n vertices, with pairwise distances (or costs) $d_{ij} \in \mathbb{N} \cup \{\infty\}$, for all $i, j \in V$. No further assumptions are made on the values d_{ij}, in particular, they may be asymmetric, i.e. d_{ij} may not equal d_{ji}, and the cost d_{ii} of a self-loop may be non-zero. Also given are integers $k_i \geq 1$ for $i \in V$, which we refer to as *multiplicities*. A valid tour of length k is a sequence $(x_1, \ldots, x_k) \in V^k$, where $k = \sum_{i \in V} k_i$, such that each $i \in V$ appears in the sequence exactly k_i times. The cost of the tour is $\sum_{i=1}^{k-1} d_{x_i, x_{i+1}} + d_{x_k, x_1}$. Our goal is to find a valid tour with minimum cost.

The problem is known as the *many-visits TSP* (MV-TSP). (As an alternative name, *high-multiplicity TSP* also appears in the literature.) It includes the standard metric TSP in the special case when $k = n$ (i.e. if $k_i = 1$ for all $i \in V$) and d_{ij} forms a metric, thus it cannot be solved in polynomial time, unless $\mathsf{P} = \mathsf{NP}$. Nonetheless, the problem is tractable in the regime of small n values, even if the length k of the tour is very large (possibly exponential in n).

As a natural TSP-generalization, MV-TSP is a fundamental problem of independent interest. In addition, MV-TSP proved to be useful for modeling other problems, particularly in scheduling [7, 22, 37, 40]. Suppose there are k jobs of n different types to be executed on a single, universal machine. Processing a job, as well as switching to another type of job come with certain costs, and the goal is to find the sequence of jobs with minimal total cost. Modeling this problem as a MV-TSP instance is straightforward, by letting d_{ij} denote

*Research of L.K. supported by ERC Consolidator Grant No 617951. Research of M.M. supported by DFG Grant MN 59/4-1.

†Maastricht University, Department of Quantitative Economics, a.berger@maastrichtuniversity.nl

‡Eindhoven University of Technology, Department of Mathematics and Computer Science, lkozma@gmail.com

§Universität Bonn, Department of Computer Science *and* Maastricht University, Department of Quantitative Economics, m.mnich@maastrichtuniversity.nl

¶Maastricht University, Department of Quantitative Economics, r.vincze@maastrichtuniversity.nl

the cost of processing a job of type i together with the cost of switching from type i to type j. Emmons and Mathur [14] also describe an application of MV-TSP to the no-wait flow shop problem.

A different kind of application comes from geometric approximation. To solve geometric optimization problems approximately, it is a standard technique to reduce the size of the input by grouping certain input points together. Each group is then replaced by a single representative, and the reduced instance is solved exactly. (For instance, we may snap input points to nearby grid points, if doing so does not significantly affect the objective cost.) Recently, this technique was used by Kozma and Mömke, to give an efficient polynomial-time approximation scheme (EPTAS) for the MAXIMUM SCATTER TSP in doubling metrics [29], addressing an open question of Arkin et al. [4]. In this case, the reduced problem is exactly the MV-TSP. Yet another application of MV-TSP is in settling the parameterized complexity of finding a Hamiltonian cycle in a graph class with restricted neighborhood structure [30].

To the best of our knowledge, MV-TSP was first considered in 1966 by Rothkopf [38]. In 1980, Psaraftis [37] gave a dynamic programming algorithm with run time $O(n^2 \cdot \prod_{i \in V} (k_i + 1))$. Observe that this quantity may be as high as $(k/n + 1)^n$, which is prohibitive even for moderately large values of k. In 1984, Cosmadakis and Papadimitriou [12] observed that MV-TSP can be decomposed into a connectivity subproblem and an assignment subproblem. Taking advantage of this decomposition, they designed a family of algorithms, the best of which has run time $O^*(n^{2n}2^n + \log k)$.[1] The result can be seen as an early example of *fixed-parameter tractability*, where the rapid growth in complexity is restricted to a certain parameter.

The algorithm of Cosmadakis and Papadimitriou is, to date, the fastest solution to MV-TSP.[2] Its analysis is highly non-trivial, combining graph-theoretic insights and involved estimates of various combinatorial quantities. The upper bound is not known to be tight, but the analysis appears difficult to improve, and a lower bound of $\Omega(n^n)$ is known to hold. Similarly, in the space requirement of the algorithm, a term of the form $n^{\Omega(n)}$ appears hard to avoid.

While it extends the tractability of TSP to a new range of parameters, the usefulness of the Cosmadakis-Papadimitriou algorithm is limited by its superexponential[3] dependence on n in the run time. In some sense, the issue of exponential *space* is even more worrisome (the survey of Woeginger [41] goes as far as calling exponential-space algorithms "*absolutely useless*").

There have been further studies of the MV-TSP problem. Van der Veen and Zhang [40] discuss a problem equivalent to MV-TSP, called *K-group TSP*, and describe an algorithm with polylogarithmic dependence on the number k of visits (similarly to Cosmadakis and Papadimitriou). The value n however is assumed constant, and its effect on the run time is not explicitly computed (the dependence can be seen to be superexponential). Finally, Grigoriev and van de Klundert [17] give an ILP formulation for MV-TSP with $O(n^2)$ variables. Applying Kannan's improvement [25] of Lenstra's algorithm [32] for solving fixed-dimensional ILPs to this formulation yields an algorithm with run time $n^{O(n^2)} \cdot \log k$. Further ILP formulations for MV-TSP are due to Sarin et al. [39] and Aguayo et al. [2], both of which again require superexponential time to be solved by standard algorithms.

For details about the history of the MV-TSP we refer to the TSP textbook of Gutin and Punnen [19, § 11.10].

Our results. Our main result improves both the time and space complexity of the best known algorithm for MV-TSP, the first improvement in over 30 years. Specifically, we show that a *logarithmic* dependence on the number k of visits, a *single-exponential* dependence on the number n of cities, and a *polynomial* space complexity are simultaneously achievable. Moreover, while we build upon ideas from the previous best approach, our algorithm is arguably both easier to describe, easier to implement, and easier to analyse than its predecessor. To introduce the techniques step-by-step, we describe *three* algorithms for solving MV-TSP. These are called ENUM-MV, DP-MV, and DC-MV. We also mention possible practical improvements. All our algorithms are deterministic. Their complexities are summarized in Theorem 1.1, proved in § 2.

THEOREM 1.1.

(i) ENUM-MV *solves MV-TSP using $O(n^2)$ space, in time $O^*(n^n)$.*

(ii) DP-MV *solves MV-TSP using space and time $O^*(5^n)$.*

(iii) DC-MV *solves MV-TSP using $O(n^2)$ space, in time $O((32 + \varepsilon)^n)$ for any $\varepsilon > 0$.*

[1] Here, and in the following, the $O^*(\cdot)$ notation is used to suppress a low-order polynomial factor in n.

[2] It may seem that a linear dependence on the length k of the tour is necessary even to output the result. Observe however, that a tour can be compactly represented by collapsing cycles and storing them together with their multiplicities.

[3] In this paper the term *superexponential* always refers to a quantity of the form $n^{\Omega(n)}$.

The Exponential-Time Hypothesis (ETH) [23] implies that TSP cannot be solved in $2^{o(n)}$, i.e. sub-exponential, time. Under this hypothesis, the run time of our algorithm DC-MV is asymptotically optimal for MV-TSP (up to the base of the exponential). Further note that the space requirement of DC-MV is also (essentially) optimal, as a compact solution encodes for each of the $\Omega(n^2)$ edges the number t of times that this edge is traversed by an optimal tour. (We assume that each multiplicity can be stored in a constant number of machine words; if this is not the case, e.g. if k is exponential in n, a factor $O(\log k)$ should be applied to the given space bounds.)

Our result leads to improvements in applications where MV-TSP is solved as a subroutine. For instance, as a corollary of Theorem 1.1, the approximation scheme for MAXIMUM SCATTER TSP [29] can now be implemented in space *polynomial* in the error parameter ε.

It is interesting to contrast our results for MV-TSP with recent results for the *r-simple path* problem, where a long path is sought that visits each vertex *at most* r times. For that problem, the fastest known algorithms—due to Abasi et al. [1] and Gabizon et al. [15]—have run time *exponential* in the average number of visits, and such exponential dependence is necessary assuming ETH.

Overview of techniques. The Cosmadakis-Papadimitriou algorithm is based on the following high-level insight, common to most work on the TSP problem, whether exact or approximate. The task of finding a valid tour may be split into two separate tasks: (1) finding a structure that connects all vertices, and (2) augmenting the structure found in (1) in order to ensure that each city is visited the required number of times.

Indeed, such an approach is also used, for instance, in the well-known 3/2-approximation algorithm of Christofides for metric TSP [9]. There, the structure that guarantees connectivity is a minimum spanning tree, and "visitability" is enforced by the addition of a perfect matching that connects odd-degree vertices (ensuring that all vertices have even degree, and can thus be entered and exited, as required).

In the case of MV-TSP, Cosmadakis and Papadimitriou ensure connectivity (part (1)) by finding a *minimal connected Eulerian digraph* on the set V. Indeed, a minimal Eulerian digraph must be part of every solution, since a tour must balance every vertex (equal out-degree and in-degree), and all vertices must be mutually reachable. Minimality is meant here in the sense that no proper subgraph is Eulerian, and is required only to reduce the search space.

Assuming that an Eulerian digraph is found that is part of the solution, it needs to be extended to a tour in which all vertices are visited the required number of times (part (2)). If this is done with the cheapest possible set of edges, then the optimum must have been found. This second step amounts to solving a transportation problem, which takes polynomial time.

The first step, however, requires us to consider *all possible* minimal Eulerian digraphs. As it is NP-complete to test the non-minimality of an Eulerian digraph [36], the authors relax minimality and suggest the use of heuristics for pruning out non-minimal instances in practice. On the other hand, they obtain a saving in run time by observing that among all Eulerian digraphs with the same *degree sequence* only the one with smallest cost needs to be considered. (Otherwise, in the final tour, the Eulerian subdigraph could be swapped with a cheaper one, while maintaining the validity of the tour.)

Cosmadakis and Papadimitriou iterate thus over feasible degree sequences of Eulerian digraphs; for each such degree sequence they construct the cheapest Eulerian digraph (which may not be minimal) by dynamic programming; finally, for each such Eulerian digraph construct the cheapest extension to a valid tour (by solving a transportation problem). The returned solution is the cheapest tour found over all iterations.

Iterating and optimizing over these structures is no easy task, and Cosmadakis and Papadimitriou invoke a number of graph-theoretic and combinatorial insights. For estimating the total cost of their procedure a sophisticated global counting argument is developed.

The key insight of our approach is that the machinery involving Eulerian digraphs is *not necessary* for solving MV-TSP. To ensure connectivity (i.e. task (1) above), a *directed spanning tree* is sufficient. This may seem surprising, as a directed tree fails to satisfy the main property of Eulerian digraphs, *strong connectivity*. Observe however, that a collection of directed edges with the same out-degrees and in-degrees as a valid MV-TSP tour is itself a valid tour, unless it consists of *disjoint* cycles. Requiring the solution to contain a tree is sufficient to avoid the case of disjoint cycles. The fact that the tree can be assumed to be *rooted* (i.e. all of its edges are directed away from some vertex) follows from the strong connectedness of the tour.[4]

Directed spanning trees are easier to enumerate and optimize over than minimal Eulerian digraphs; this fact alone explains the reduced complexity of our approach. However, to obtain our main result, further ideas are needed. In particular, we find the cheapest directed spanning tree that is feasible for a given

[4]We thank Andreas Björklund for the latter observation which led to an improved run time and a simpler correctness argument.

degree sequence, first by dynamic programming, then by a recursive partitioning of trees, based on centroid-decompositions.

Given the fundamental nature of the TSP-family of problems and the remaining open questions they pose, we hope that our techniques may find further applications.

2 Improved algorithms for the Many-Visits TSP

In this section we describe and analyse our three algorithms. The first, ENUM-MV is based on exact enumeration of trees (§ 2.2), the second, DP-MV uses a dynamic programming approach to find an optimal tree (§ 2.3), and the third, DC-MV is based on divide and conquer (§ 2.4). Before presenting the algorithms, we introduce some notation and structural observations that are subsequently used (§ 2.1).

2.1 Trees, tours, and degree sequences. Let V be a set of vertices. We view a *directed multigraph* G with vertex set V as a *multiset* of edges (i.e. elements of $V \times V$). Accordingly, self-loops and multiple copies of the same edge are allowed. The *multiplicity* of an edge (i, j) in a directed multigraph G is denoted $m_G(i, j)$. The *out-degree* of a vertex $i \in V$ is $\delta^{\text{out}}_G(i) = \sum_{j \in V} m_G(i, j)$, the *in-degree* of a vertex $i \in V$ is $\delta^{\text{in}}_G(i) = \sum_{j \in V} m_G(j, i)$. Given edge costs $d : V \times V \to \mathbb{N} \cup \{\infty\}$, the *cost* of G is simply the sum of its edge costs, i.e. $\text{cost}(G) = \sum_{i,j \in V} m_G(i, j) \cdot d(i, j)$.

For two directed multigraphs G and H over the same vertex set V, let $G + H$ denote the directed multigraph obtained by adding the corresponding edge multiplicities of G and H. Observe that as an effect, out-degrees and in-degrees are also added pointwise. Formally, $m_{G+H}(i, j) = m_G(i, j) + m_H(i, j)$, and $\delta^{\text{out}}_{G+H}(i) = \delta^{\text{out}}_G(i) + \delta^{\text{out}}_H(i)$, and $\delta^{\text{in}}_{G+H}(i) = \delta^{\text{in}}_G(i) + \delta^{\text{in}}_H(i)$, for all $i, j \in V$. The relation $\text{cost}(G + H) = \text{cost}(G) + \text{cost}(H)$ clearly holds.

In the following, for a directed multigraph G we refer to its *underlying graph*, i.e. to the undirected graph consisting of those edges $\{i, j\}$ for which $m_G(i, j) + m_G(j, i) \geq 1$.

Consider a tour $C = (x_1, \ldots, x_k) \in V^k$. We refer to the unique directed multigraph G consisting of the edges $(x_1, x_2), \ldots, (x_{k-1}, x_k), (x_k, x_1)$ as the *edge set* of C. We state a simple but crucial observation.

LEMMA 2.1. *Let G be a directed multigraph over V with out-degrees $\delta^{\text{out}}_G(\cdot)$ and in-degrees $\delta^{\text{in}}_G(\cdot)$. Then G is the edge set of a tour that visits each vertex $i \in V$ exactly k_i times if and only if both of the following conditions hold:*

(i) the underlying graph of G is connected, and

(ii) for all $i \in V$, we have $\delta^{\text{out}}_G(i) = \delta^{\text{in}}_G(i) = k_i$.

Proof. The fact that connectedness of G and $\delta^{\text{out}}_G(i) = \delta^{\text{in}}_G(i)$ is equivalent with the existence of a tour that uses each edge of G exactly once is the well-known "Euler's theorem". (See [5, Thm 1.6.3] for a short proof.) Clearly, visiting each vertex i exactly k_i times is equivalent with the condition that the tour contains k_i edges of the form $(\cdot, i)$ and k_i edges of the form $(i, \cdot)$. □

Moreover, given the edge set G of a tour C, a tour C' with edge set G can easily be recovered. This amounts to finding an Eulerian tour of G, which can be done in time linear in the length k of the tour. To avoid a linear dependence on k, we can apply the algorithm of Grigoriev and van de Klundert [17] that constructs a compact representation of C' in time $O(n^4 \log k)$. As the edge sets of C and C' are equal, C' also visits each $i \in V$ exactly k_i times and $\text{cost}(C') = \text{cost}(C) = \text{cost}(G)$.

Thus, in solving MV-TSP we only focus on finding a directed multigraph whose underlying undirected graph is connected, and whose degrees match the multiplicities required by the problem.

A *directed spanning tree* of V is a tree with vertex set V whose edges are directed *away* from some vertex $r \in V$; in other words, the tree contains a directed path from r to every other vertex in V. (Directed spanning trees are alternatively called *branchings*, *arborescences*, or *out-trees*.) We refer to the vertex r as the *root* of the tree. We observe that every valid tour contains a directed spanning tree.

LEMMA 2.2. *Let G be the edge set of a tour of V (with arbitrary non-zero multiplicities), and let $r \in V$ be an arbitrary vertex. Then there is a directed spanning tree T of G rooted at r, and a directed multigraph X, such that $G = T + X$.*

Proof. We choose T to be the single-source *shortest path tree* in G with source r. More precisely, let P_x be the edge set of the shortest path from r to x in G for all $x \in V \setminus \{r\}$. (In a valid tour all vertices are mutually reachable.) In case of ties, we choose the path that is alphabetically smaller. Let $T = \bigcup_{x \in V \setminus \{r\}} P_x$, i.e. the union of all shortest paths. The fact that T is a directed tree with root r is a direct consequence of the definition of shortest paths (see e.g. [11, § 24]). □

We can thus split the MV-TSP problem into finding a directed spanning tree T with an arbitrary root r and an extension X, such that $T + X$ is a valid tour. We claim that in the decomposition $G = T + X$ of an optimal tour G, both T and X are optimal with respect to their degree sequences.

LEMMA 2.3. *Let G be the edge set of an optimal tour for MV-TSP, let T be a directed spanning tree, and let X be a directed multigraph such that $G = T + X$. Then, T has the smallest cost among all directed spanning trees with degrees $\delta_T^{\text{out}}(\cdot)$ and $\delta_T^{\text{in}}(\cdot)$, and X has the smallest cost among all directed multigraphs with degrees $\delta_X^{\text{out}}(\cdot)$ and $\delta_X^{\text{in}}(\cdot)$.*

Proof. Suppose there is a directed spanning tree T' such that $\text{cost}(T') < \text{cost}(T)$, and $\delta_{T'}^{\text{out}}(i) = \delta_T^{\text{out}}(i)$, and $\delta_{T'}^{\text{in}}(i) = \delta_T^{\text{in}}(i)$ for all $i \in V$. But then $T' + X$ is connected, has the same degree sequence as G, while $\text{cost}(T' + X) < \text{cost}(G)$, contradicting the optimality of G.

Similarly, suppose there is a directed multigraph X' such that $\text{cost}(X') < \text{cost}(X)$, and $\delta_{X'}^{\text{out}}(i) = \delta_X^{\text{out}}(i)$, and $\delta_{X'}^{\text{in}}(i) = \delta_X^{\text{in}}(i)$ for all $i \in V$. But then $T + X'$ is connected, has the same degree sequence as G, while $\text{cost}(T + X') < \text{cost}(G)$, contradicting the optimality of G. □

Next, we characterize the feasible degree sequences of directed spanning trees.

LEMMA 2.4. *Let $V = \{x_1, \ldots, x_n\}$ be a set of vertices. There is a directed spanning tree of V with root x_1 whose out-degrees and in-degrees are respectively $\delta^{\text{out}}(\cdot)$ and $\delta^{\text{in}}(\cdot)$, if and only if*

(i) $\delta^{\text{in}}(x_1) = 0$,

(ii) $\delta^{\text{in}}(x_i) = 1$ *for every* $1 < i \leq n$,

(iii) $\delta^{\text{out}}(x_1) > 0$, *and*

(iv) $\sum_i \delta^{\text{out}}(x_i) = n - 1$.

Proof. In the forward direction, in a directed spanning tree all non-root vertices have exactly one parent, proving (i) and (ii). The root must have at least one child (iii), and the total number of edges is $n-1$, proving (iv).

In the backward direction, we argue by induction on n. In the case $n = 2$, we have $\delta^{\text{in}}(x_1) = \delta^{\text{out}}(x_2) = 0$, and $\delta^{\text{in}}(x_2) = \delta^{\text{out}}(x_1) = 1$, hence an edge (x_1, x_2) satisfies the degree requirements.

Consider now the case of $n > 2$ vertices. From *(ii)*–*(iv)* it follows that for some k $(1 < k \leq n)$, we have $\delta^{\text{in}}(x_k) = 1$ and $\delta^{\text{out}}(x_k) = 0$, i.e. x_k is a leaf.

Let x_j be a vertex $(1 \leq j \leq n$, and $j \neq k)$, such that $\delta^{\text{out}}(x_j) \geq 1$, and $\delta^{\text{in}}(x_j) + \delta^{\text{out}}(x_j) \geq 2$; by *(ii)*–*(iv)* there must be such a vertex. We decrease $\delta^{\text{out}}(x_j)$ by one. Conditions *(i)*–*(iv)* clearly hold for $V \setminus \{x_k\}$. By induction, we can build a tree on $V \setminus \{x_k\}$, and attach x_k to this tree as a leaf, with the edge (x_j, x_k). □

Let $\text{DS}(n)$ denote the number of different pairs of sequences $(\delta'_1, \ldots, \delta'_n), (\delta''_1, \ldots, \delta''_n)$, that are feasible for a directed spanning tree, i.e. for vertex set $V = \{x_1, \ldots, x_n\}$, for some directed spanning tree T with root x_1, we have $\delta_T^{\text{out}}(x_i) = \delta'_i$, and $\delta_T^{\text{in}}(x_i) = \delta''_i$, for all $i \in \{1, \ldots, n\}$. By Lemma 2.4, $\text{DS}(n)$ equals the number of ways to distribute $n-1$ out-degrees to n vertices, such that a designated vertex has non-zero out-degree. This task is the same as distributing $n - 2$ balls arbitrarily into n bins, of which there are $\binom{2n-3}{n-1} = O(4^n)$ ways.

2.2 ENUM-MV: polynomial space and superexponential time. Given a vertex set V with $|V| = n$, multiplicities $k_i \in \mathbb{N}$ and cost function d, we wish to find a tour of minimum cost that visits each $i \in V$ exactly k_i times.

From Lemma 2.2, our first algorithm presents itself. It simply iterates over all directed spanning trees T with vertex set V, and extends each of them optimally to a valid tour C_T. Among all valid tours constructed, the one with smallest cost is returned (Algorithm 1).

This simple algorithm already improves on the previous best run time (although it is still superexponential), and reduces the space requirement from superexponential to polynomial.

The correctness of the algorithm is immediate: from Lemma 2.2 it follows that all C_T's considered are valid (connected, and with degrees matching the required multiplicities), and by Lemma 2.3, the optimal tour C must be considered during the execution.

The iteration of Line 1 requires us to enumerate all labeled trees with vertex set V. There are n^{n-2} such trees [8], and standard techniques can be used to enumerate them with a constant overhead per item (see e.g. Kapoor and Ramesh [26]). For each considered tree we orient the edges in a unique way (away from r).

Let T be the current tree. In Line 2 we need to find a minimum cost directed multigraph X, with given out-degree and in-degree sequence (such as to extend T into a valid tour). If, for some vertex i, it holds that $\delta_T^{\text{in}}(i) > k_i$ or $\delta_T^{\text{out}}(i) > k_i$, we proceed to the next spanning tree, since the current tree cannot be extended to a valid tour. Observe that this may happen only if $k_i < n - 1$, for some $i \in V$.

Otherwise, we find the optimal X by solving a transportation problem in polynomial time. During the algorithm we maintain the current best tour, which we output in the end. We describe next the transportation subroutine, which is common to all our algorithms, and is essentially the same as in the Cosmadakis-Papadimitriou algorithm.

Algorithm 1 ENUM-MV for solving MV-TSP using enumeration

Input: Vertex set V, cost function d, multiplicities k_i.

Output: A tour of minimum cost that visits each $i \in V$ exactly k_i times.

1: **for** each directed spanning tree T with root $r \in V$ **do**

2: Find minimum cost directed multigraph X such that for all $i \in V$:
$\delta^{\text{out}}_X(i) := k_i - \delta^{\text{out}}_T(i)$,
$\delta^{\text{in}}_X(i) := k_i - \delta^{\text{in}}_T(i)$.
Denote $C_T := T + X$.

3: **return** C_T with smallest cost.

The transportation problem. The subproblem we need to solve is finding a minimum cost directed multigraph X over vertex set V, with given out-degree and in-degree requirements.

We can map this problem to an instance of the Hitchcock transportation problem [21], where the goal is to transport a given amount of goods from N warehouses to M outlets with given pairwise shipping costs. (This is a special case of the more general min-cost max-flow problem.)

More precisely, let us define a digraph with vertices $\{s,t\} \cup \{s_i, t_i \mid i \in V\}$. Edges are $\{(s,s_i),(t_i,t) \mid i \in V\}$, and $\{(s_i,t_j) \mid i,j \in V\}$. We set cost 0 to edges (s,s_i) and (t_i,t) and cost d_{ij} (i.e. the costs given in the MV-TSP instance) to (s_i,t_j). We set capacity ∞ to edges (s_i,t_j) and capacities $k_i - \delta^{\text{out}}_T(i)$ to (s,s_i), and $k_i - \delta^{\text{in}}_T(i)$ to (t_i,t). The construction is identical to the one used by Cosmadakis and Papadimitriou, apart from the fact that in our case the capacity of (s,s_i) may be different from the capacity of (t_i,t). Observe that the sum of capacities of (s,s_i)-edges equals the sum of capacities of (t_i,t)-edges over all $i \in V$. Thus, a maximal $s-t$ flow saturates all these edges.

The amount of flow transmitted on the edge (s_i,t_j) gives the multiplicity of edge (i,j) in the sought after multigraph, for all $i,j \in V$. A minimum cost maximum flow clearly maps to a minimum cost edge set with the given degree constraints.

In the Cosmadakis-Papadimitriou algorithm, the transportation subproblems are solved via the scaling method of Edmonds and Karp [13]. This algorithm proceeds by solving $O(\log k)$ approximate versions of the problem, where the costs are the same as in the original problem, but the capacities are scaled by a factor 2^p for $p = \lceil \log k \rceil, \ldots, 0$. Each approximate problem is solved in $O(n^3)$ time, by performing flow augmentations on the optimal flow found in the previous approximation (multiplied by two). The overall run time for solving the described transportation problem is therefore $O(n^3 \cdot \log k)$.

Cosmadakis and Papadimitriou describe an improvement which also applies for our case. Namely, they show that the total run time for solving several instances with the same costs can be reduced, if the capacities on corresponding edges in two different instances may differ by at most n.

The strategy is to solve all but the last $\log_2 n$ approximate problems only once, as these are (essentially) the same for all instances. For different instances we only need to solve the last $\log_2 n$ approximate problems (i.e. at the finest levels of approximation). This gives a run time of $O(n^3 \cdot \log k)$ for solving the "master problem", and $O(n^3 \cdot \log n)$ for solving each individual instance. (We refer to Cosmadakis and Papadimitriou [12] as well as Edmonds and Karp [13] for details.)

In our case, the different instances of the transportation problem are for finding the directed multigraphs X for different trees T. Each of these instances agree in the underlying graph and cost function, and may differ only in the capacities. As the maximum degree of each tree T is at most $n-1$, the differences are bounded, as required.

As an alternative to the Edmonds-Karp algorithm, we may solve the arising transportation problems with a *strongly polynomial* algorithm, e.g. the one by Orlin [35] or its extension due to Kleinschmidt and Schannath [27]. (These algorithms were not yet available when Cosmadakis and Papadimitriou obtained their result.) The run time of these algorithms for the transportation subproblem is $O(n^3 \log n)$, i.e. independent of k. Such an improvement is likely of theoretical interest only; furthermore, it assumes that operations on the multiplicities k_i take constant time. If this assumption is unrealistic (e.g. if k is exponential in n), we may fall back to the Edmonds-Karp algorithm, and the term $O(n^3 \log k)$ should be *added* to our stated running times.

Analysis of Algorithm 1. We iterate over all $O(n^{n-2})$ directed spanning trees and solve a transportation problem for each, with run time $O(n^3 \cdot \log n)$. The total run time $O(n^{n+1} \cdot \log n)$ follows. The space requirement of the algorithm is dominated by that of solving a (single) transportation problem, and of storing the edge set of a single tour (apart from minor bookkeeping).

2.2.1 Improved enumeration algorithm. We can improve the run time of ENUM-MV by observing that the solution of the transportation problem depends only on the degree sequence of the current tree T, and not the actual edges of T. Therefore, different trees with the same degree sequence can be extended in the same way. (Several trees may have the same degree sequence; in an extreme case, all $(n-2)!$ simple Hamiltonian paths with the same endpoints have the same degree sequence.)

Algorithm 2 implements this idea, iterating over all trees, grouped by their degree sequences. It solves the transportation problem only once for each degree sequence (there are $\mathrm{DS}(n)$ of them).

The correctness is immediate, as all directed spanning trees T are still considered, as before. Assume that we can iterate over all feasible degree sequences, and all corresponding trees with $O(n)$ overhead per item. By Lemma 2.4, the first task only requires us to consider all ways of distributing $n-2$ out-degrees among n vertices. For completeness, we describe a procedure for this task in Appendix A. We give the subroutine for the second task in § 2.2.2. We thus obtain the run time $O(n^{n-1} + \mathrm{DS}(n) \cdot n^4 \cdot \log n) = O(n^{n-1})$. The space requirement is asymptotically unchanged.

2.2.2 Generating trees by degree sequence. In the proof of Lemma 2.4, we generate *one* directed tree from its degree sequence. In this subsection we show how to generate *all* trees for a given degree sequence (Algorithm 3).

The initial call to the BUILDTREE() procedure is with a feasible input degree sequence $(\delta^{\mathrm{out}}, \delta^{\mathrm{in}})$ and an empty "stub" ($\mathbf{x}^{\mathrm{stub}} \equiv 0$) as arguments. The algorithm finds the first *unattached* vertex that is either a leaf of the final tree or whose subtree is already complete (that is, $\delta^{\mathrm{out}}(x_i) = 0$), then it finds all possibilities for attaching x_i to the rest of the tree. (The fact that x_i has no more capacity for outgoing edges prevents cycles.) The procedure is then called recursively with modified arguments: edge (x_j, x_i) is added to the stub and the out-degree of x_j and the in-degree of x_i are decremented.

At each recursive level there are as many new calls as possible candidates for the next edge, with both degree demands decreased by one, and with the stub gaining one additional edge. (During the intermediate calls the stub may be disconnected.) At the $(n-1)$-th level exactly two one-degree vertices remain, say, x_i with in-degree 1 and the root x_1 with out-degree 1. Adding the edge (x_1, x_i) finishes the construction.

Observe that there are no "dead ends" during this process, i.e. every call of BUILDTREE eventually results in a valid directed tree in the last level of the recursion, and there are no discarded graphs during the process. Furthermore, each tree is constructed exactly once.

2.3 DP-MV: exponential space and single-exponential time. Next, we improve the run time to single-exponential, by making use of Lemma 2.3. Specifically, we observe that for every feasible degree sequence only the tree with minimum cost needs to be considered. The enumeration in Algorithm 3 can easily be modified to return, instead of all trees, just the one with smallest cost, this, however, would not improve the asymptotic run time. Instead, in this section we give a dynamic programming algorithm resembling the algorithms by Bellman, and Held and Karp, for directly computing the best directed tree for a given sequence.

The outline of the algorithm is shown in Algorithm 4, and it is identical for DP-MV and DC-MV, described in § 2.4. The algorithms DP-MV and DC-MV differ in the way they find the minimum cost directed tree, i.e. Line 2 of the generic Algorithm 4.

The dynamic programming approach (DP-MV) resembles Algorithm 3 for iterating over all directed trees with a given degree sequence. Specifically, let $(\delta^{\mathrm{out}}, \delta^{\mathrm{in}})$ be the degree sequence for which we wish to find the minimum-cost tree. The dynamic programming table $\mathcal{T}$ holds the optimum tree (and its cost) for *every* feasible degree sequence. The solution can thus be read from $\mathcal{T}[\delta^{\mathrm{out}}, \delta^{\mathrm{in}}]$.

Observe that specifying a degree sequence allows us to restrict the problem to arbitrary subsets of V (we can simply set the degrees of non-participating vertices to zero).

To compute $\mathcal{T}[\delta^{\mathrm{out}}, \delta^{\mathrm{in}}]$, we find the leaf with smallest index x_i and all non-leaves x_j that may be connected to x_i by an edge in the optimal tree (similarly to Algorithm 3). For each choice of x_j we compute the optimal tree by adding the connecting edge (x_j, x_i) to the optimal tree over $V \setminus \{x_i\}$, with the degree sequence suitably updated.

The correctness of the dynamic programming algorithm follows from an observation similar to Lemma 2.3; every subtree of the optimal tree must be optimal for its degree sequence, as otherwise it could be swapped for a cheaper subtree. The details are shown in Algorithm 5.

Analysis of Algorithm 5. Observe that the number of possible entries $\mathcal{T}[\cdot,\cdot]$ is $\sum_{k=1}^{n-1} \binom{n-1}{k} \mathrm{DS}(k)$, i.e. the number of feasible degree sequences for trees on subsets of V. Using our previous estimate $\mathrm{DS}(n) = O(4^n)$, this sum evaluates to $O(5^n)$ (by the binomial theorem), yielding the required time complexity. The overall run time of Algorithm 4 (excluding the transportation subproblem) with Algorithm 5 as a subroutine is still

Algorithm 2 ENUM-MV for solving MV-TSP using enumeration (improved)

Input: Vertex set V, cost function d, multiplicities k_i.
Output: A tour of minimum cost that visits each $i \in V$ exactly k_i times.

for each feasible degree sequence $\delta^{\text{out}}(\cdot), \delta^{\text{in}}(\cdot)$ of a directed spanning tree with root r **do**
  Find minimum cost directed multigraph X such that for all $i \in V$:
    $\delta^{\text{out}}_X(i) := k_i - \delta^{\text{out}}(i)$,
    $\delta^{\text{in}}_X(i) := k_i - \delta^{\text{in}}(i)$.
  for each directed spanning tree T with $\delta^{\text{out}}_T = \delta^{\text{out}}$ and $\delta^{\text{in}}_T = \delta^{\text{in}}$ **do**
    Denote $C_T := T + X$.
return C_T with smallest cost.

Algorithm 3 BUILDTREE for generating all directed trees for a given degree sequence

Input: degree sequence $(\delta^{\text{out}}, \delta^{\text{in}})$ and partial tree $\mathbf{x}^{\text{stub}}$
procedure BUILDTREE(δ^{out}, δ^{in}, $\mathbf{x}^{\text{stub}}$)
  if $\sum \delta^{\text{in}}(\cdot) = 1$ **then**
    $i \leftarrow$ index where $\delta^{\text{in}}(x_i) = 1$
    yield $x^{\text{stub}} \cup \{(x_1, x_i)\}$
  else
    $i \leftarrow$ first index where $\delta^{\text{out}}(x_i) = 0$ and $\delta^{\text{in}}(x_i) = 1$ ▷ attach x_i to parent
    for every index $j \neq i$, s.t. $\delta^{\text{out}}(x_j) \geq 1$ **do**
      $\delta^{\text{out}\prime} := \delta^{\text{out}}$ and $\delta^{\text{in}\prime} := \delta^{\text{in}}$
      $\delta^{\text{out}\prime}(x_j) := \delta^{\text{out}\prime}(x_j) - 1$ and $\delta^{\text{in}\prime}(x_i) := \delta^{\text{in}\prime}(x_i) - 1$
      BUILDTREE($\delta^{\text{out}\prime}$, $\delta^{\text{in}\prime}$, $\mathbf{x}^{\text{stub}} \cup \{(x_j, x_i)\}$)

Algorithm 4 Solving MV-TSP by optimizing over trees (common for DP-MV and DC-MV).

Input: Vertex set V, cost function d, multiplicities k_i.
Output: A tour of minimum cost that visits each $i \in V$ exactly k_i times.

for each feasible degree sequence $\delta^{\text{out}}(\cdot), \delta^{\text{in}}(\cdot)$ of a directed spanning tree **do**
  Find a minimum-cost directed tree T with $\delta^{\text{in}}_T = \delta^{\text{in}}$ and $\delta^{\text{out}}_T = \delta^{\text{out}}$.
  Find a minimum-cost directed multigraph X such that for all $i \in V$:
    $\delta^{\text{in}}_X(i) := k_i - \delta^{\text{in}}_T(i)$,
    $\delta^{\text{out}}_X(i) := k_i - \delta^{\text{out}}_T(i)$.
  Denote $C_T := T + X$.
return C_T with smallest cost.

$O^*(5^n)$, since the entry for a given degree sequence is computed at most once over all iterations, and the values are stored through the entire iteration. In practice, storing an entire tree in each $\mathcal{T}[\cdot,\cdot]$ is wasteful; for the optimum tree to be constructible from the table, it is sufficient to store the node to which the lowest-index leaf is connected. The claimed time and space complexity follows.

2.4 DC-MV: polynomial space and single-exponential time. Finally, we show how to reduce the space complexity to polynomial, while maintaining a single-exponential run time.

The outer loop (Algorithm 4) remains the same, but we replace the subroutine for finding an optimal directed spanning tree (Algorithm 5) with an approach based on divide and conquer (Algorithm 6). The approach is inspired by the algorithm of Gurevich and Shelah [18] for finding an optimal TSP tour.

The algorithm relies on the following observation about tree-separators. Let (V_1, V_2) be a partition of V, i.e. $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \emptyset$, and let $|V| = n$. We say that (V_1, V_2) is *balanced* if $\lfloor n/3 \rfloor \leq |V_1|, |V_2| \leq \lceil 2n/3 \rceil$.

LEMMA 2.5. *For every tree T with edge set V, there is a balanced partition (V_1, V_2) of V such that all edges of T between V_1 and V_2 are incident to a vertex $v \in V_1$.*

Proof. A very old result of Jordan [24] states that every tree has a *centroid* vertex, i.e. a vertex whose removal splits the tree into subtrees not larger than half the

Algorithm 5 DP-MV for generating the optimal directed tree for a given degree sequence.

Input: degree sequence $(\delta^{\text{out}}, \delta^{\text{in}})$.
Output: (T, c), where T is optimum tree with root x_1, degrees $\delta^{\text{out}}, \delta^{\text{in}}$ and cost c.

1: **procedure** $\mathcal{T}\left[\delta^{\text{out}}, \delta^{\text{in}}\right]$
2: **if** $\sum \delta^{\text{in}}(\cdot) = 1$ **then**
3: $i \leftarrow$ index where $\delta^{\text{in}}(x_i) = 1$
4: **return** $(\{(x_1, x_i)\}, d_{1i})$
5: **else** $(\sum \delta^{\text{in}}(\cdot) > 1)$
6: $i \leftarrow$ first index where $\delta^{\text{out}}(x_i) = 0$ and $\delta^{\text{in}}(x_i) = 1$ ▷ attach x_i to parent
7: **for** every index $j \neq i$, s.t. $\delta^{\text{out}}(x_j) \geq 1$ **do**
8: $\delta^{\text{out}\prime} := \delta^{\text{out}}$ and $\delta^{\text{in}\prime} := \delta^{\text{in}}$
9: $\delta^{\text{out}\prime}(x_j) := \delta^{\text{out}\prime}(x_j) - 1$ and $\delta^{\text{in}\prime}(x_i) := \delta^{\text{in}\prime}(x_i) - 1$
10: $(T_j, c_j) := \mathcal{T}[\delta^{\text{out}\prime}, \delta^{\text{in}\prime}]$
11: $T_j := T_j \cup \{(x_j, x_i)\}$
12: $c_j := c_j + d_{ji}$
13: **return** (T_j, c_j) with minimum c_j

original tree. (To find such a centroid, move from an arbitrary vertex, one edge at a time, towards the largest subtree).

Let $T_1, \ldots, T_m$ be the vertex sets of the trees, in decreasing order of size, resulting from deleting the centroid of T. Let $e_1, \ldots, e_m$ denote the edges that connect the respective trees to the centroid in T. If $|T_1| \geq \lfloor n/3 \rfloor$, then we have the balanced partition $(V \setminus T_1, T_1)$, with a single crossing edge.

Otherwise, let m' be the smallest index such that $\lfloor n/3 \rfloor \leq |T_1| + \cdots + |T_{m'}| \leq \lceil 2n/3 \rceil$. (As $|T_i| < \lfloor n/3 \rfloor$ for all i, such an index m' must exist.) Now the balanced partition is $(V \setminus (T_1 \cup \cdots \cup T_{m'}), T_1 \cup \cdots \cup T_{m'})$. The partition fulfils the stated condition as all crossing edges are incident to the centroid, which is on the left side. □

At a high-level, DC-MV works as follows. It "guesses" the partition (V_1, V_2) of vertex set V according to a balanced separator of the (unknown) optimal rooted tree T, satisfying the conditions of Lemma 2.5. It also "guesses" the distinguished vertex $v \in V_1$ to which all edges that cross the partition are incident.

The balanced separator splits T into a tree with vertex set V_1 and a forest with vertex set V_2. There are two cases to consider, depending on whether the root $r = x_1$ of T falls in V_1 or V_2.

In the first case, V_1 induces a directed subtree of T rooted at r, in the second case, V_1 induces a directed subtree of T rooted at v.

Observe that the out-degrees and in-degrees of vertices in V_1 are feasible for a directed tree, except for vertex v which has additional degrees due to the edges crossing the partition (we refer to these out-degrees and in-degrees as the *excess* of v). The excess of v can be computed from Lemma 2.4. This excess determines the number and orientation of edges across the cut (even if the endpoints, other than v, of the edges are unknown).

By the same argument as in Lemma 2.3, the subtree of T induced by V_1 is optimal for its corresponding degree sequence (after subtracting the excess of v), therefore we can find it by a recursive call to the procedure.

On the other side of the partition we have a *collection* of trees. To obtain an instance of our original problem, we add a virtual vertex w to V_2 (that plays the role of v). We set the out-degree and in-degree of w to the *excess* of v, to allow it to connect to all vertices of V_2 that v connects to in T. Now we can find the optimal tree on this side too, by a recursive call to the procedure.

On both sides, the roots of the directed trees are uniquely determined by the remaining degrees. Observe that if the original root coincides with the centroid v, then v and w will be the roots of the trees in the recursive calls.

After obtaining the optimal trees on the two sides (assuming the guesses were correct), we reconstruct T by gluing the two trees together, identifying the vertices v and w. As this operation adds all degrees, we obtain a valid tree for the original degree sequence, furthermore, the tree must be optimal. We illustrate the two cases of this process in Fig. 1 and describe the algorithm in Algorithm 6.

Finally, we remark that the "guessing" should be understood as iterating through all possible choices.

In Line 7 of Algorithm 6, the excess out-degree and in-degree of v is calculated. Both types of degrees need to sum to $|V_1| - 1$, from which the expression follows. The condition in Line 12 ensures that we only

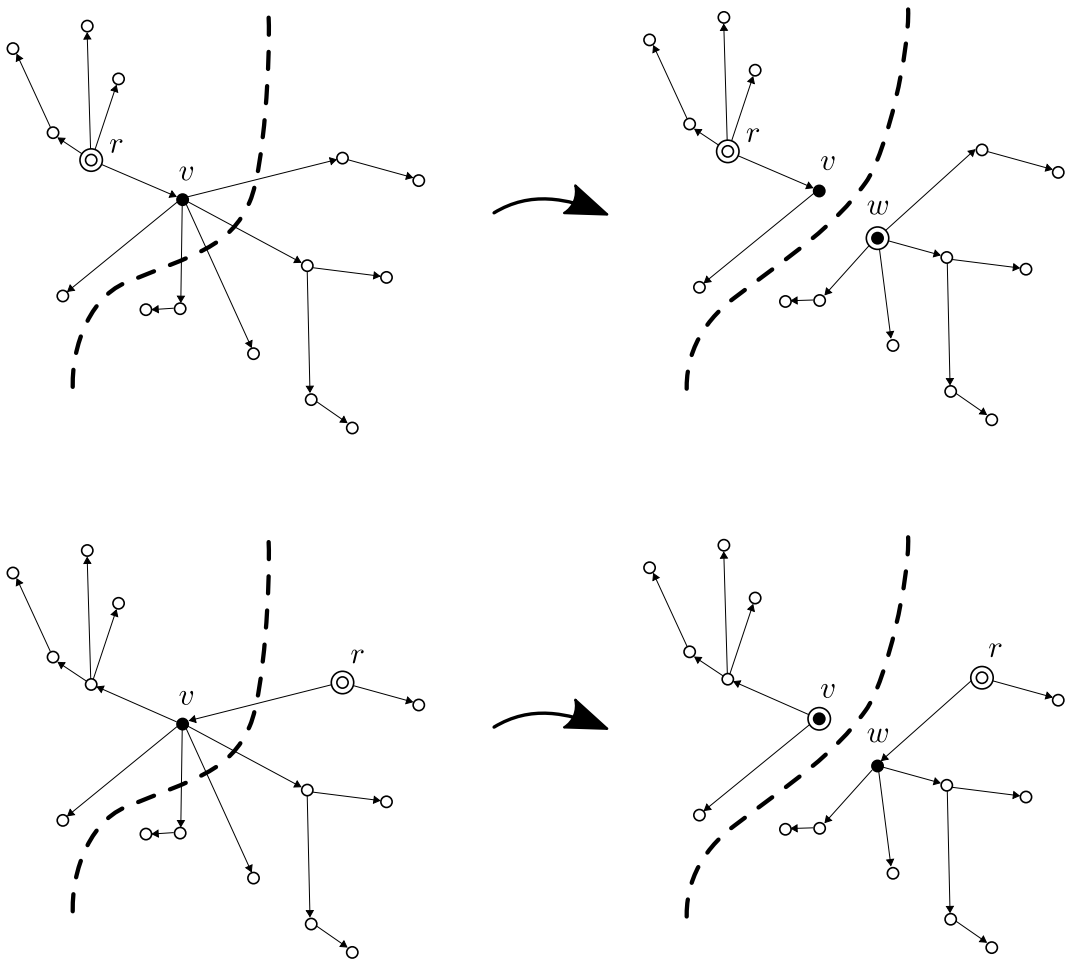

Figure 1: Illustration of DC-MV. (left) optimal tree and balanced centroid-partitioning, centroid vertex shown as filled circle; (right) optimal trees on the two sides of the partitioning, centroid vertex v and its virtual pair w shown as filled circles; (above) root falls to the left side of the partition, shown as double circle; (below) root falls to the right side of the partition, shown as double circle.

solve feasible problems. The notation $\delta|_X$ indicates a restriction of a degree sequence to a set X of vertices.

For a set V of size n we consider at most 2^n partitions and at most $\lceil 2n/3 \rceil$ choices of v. We recur on subsets of size at most $\lceil 2n/3 \rceil$. All remaining operations take $O(n)$ time. The run time $t(n)$ of DC-MV (excluding the transportation subproblem) thus satisfies:

$$\begin{aligned} t(n) &\leq 2^n \cdot n \cdot 2 \cdot \tau(2n/3) \\ &= n^{O(\log n)} \cdot 2^{n \cdot \left(\sum_k (2/3)^k\right)} \\ &= n^{O(\log n)} \cdot O(2^{3n}) \ . \end{aligned}$$

The overall run time is therefore $\mathrm{DS}(n) \cdot 8^n \cdot n^{O(\log n)} = O((32+\varepsilon)^n)$, for any $\varepsilon > 0$. For the space complexity, observe that as we do not precompute $\mathcal{T}[\cdot,\cdot,\cdot]$, only a single tour is stored at each time, spread over $O(\log n)$ recursive levels. The claimed bounds follow.

3 Discussion

We described three new algorithms for the many-visits TSP problem. In particular, we showed how to solve the problem in time single-exponential in the number n of cities, while using space only polynomial in n. This yields the first improvement over the Cosmadakis-Papadimitriou algorithm in more than 30 years.

It remains an interesting open question to improve the bases of the exponentials in our run times; recent algebraic techniques [33, 16, 28] may be of help. An algorithm solving MV-TSP in time $O^*(2^n)$, i.e. matching the best known upper bounds for solving the standard TSP, would be particularly interesting, even in the special case when all edge costs are equal to 1 or 2. (Such instances of MV-TSP arise e.g. in the Maximum Scatter TSP application [29].)

Algorithm 6 DC-MV for generating the optimal directed tree for a given degree sequence.

Input: degree sequence $\left(\delta^{\text{out}}, \delta^{\text{in}}\right)$, vertex set V
Output: (T, c), where T is optimum tree for $\delta^{\text{out}}, \delta^{\text{in}}$ and c is its cost

procedure $\mathcal{T}\left[V, \delta^{\text{out}}, \delta^{\text{in}}\right]$
  if $|V| \leq 3$ **then**
    directly find optimum tree T with cost c
    return (T, c)
  else
    for each partition (V_1, V_2) of V, such that $|V_1|, |V_2| \leq \lceil 2n/3 \rceil$ **do**
      for each $v \in V_1$ **do**
        $\mathsf{excess}_v^{\text{out}} := |V_1| - 1 - \sum_{\substack{p \in V_1 \\ p \neq v}} \delta^{\text{out}}(p)$ and $\mathsf{excess}_v^{\text{in}} := |V_1| - 1 - \sum_{\substack{p \in V_1 \\ p \neq v}} \delta^{\text{in}}(p)$
        $\delta^{\text{out}'} := \delta^{\text{out}}$ and $\delta^{\text{in}'} := \delta^{\text{in}}$
        $\delta^{\text{out}'}(v) := \delta^{\text{out}'}(v) - \mathsf{excess}_v^{\text{out}}$ and $\delta^{\text{in}'}(v) := \delta^{\text{in}'}(v) - \mathsf{excess}_v^{\text{in}}$
        $V_2' := V_2 \cup \{w\}$
        $\delta^{\text{out}'}(w) := \mathsf{excess}_v^{\text{out}}$ and $\delta^{\text{in}'}(w) := \mathsf{excess}_v^{\text{in}}$
        if $\delta^{\text{out}'}(\cdot), \delta^{\text{in}'}(\cdot) \geq 0$ and $\delta^{\text{out}'}(v) + \delta^{\text{in}'}(v) \geq 1$ and $\delta^{\text{out}'}(w) + \delta^{\text{in}'}(w) \geq 1$ **then**
          $c_1 := \mathcal{T}\left[V_1, \ \delta^{\text{out}'}|_{V_1}, \ \delta^{\text{in}'}|_{V_1}\right]$
          $c_2 := \mathcal{T}\left[V_2', \ \delta^{\text{out}'}|_{V_2'}, \ \delta^{\text{in}'}|_{V_2'}\right]$
          $c := c_1 + c_2$
          $T \leftarrow$ merge T_1 and T_2 by identifying v and w
    return (T, c) with minimum c

In practice, one may reduce the search space of our algorithms via heuristics, for instance by forcing certain (directed) edges to be part of the solution. An edge (i, j) is part of the solution if the optimal tour visits it *at least once*. This may be reasonable if the edges in question are very cheap.

References

[1] H. Abasi, N. H. Bshouty, A. Gabizon, and E. Haramaty. On r-simple k-path. In *Proc. MFCS 2014*, pages 1–12, 2014.

[2] M. M. Aguayo, S. C. Sarin, and H. D. Sherali. Single-commodity flow-based formulations and accelerated Benders algorithms for the high-multiplicity asymmetric traveling salesman problem and its extensions. *J. Oper. Res. Soc.*, 69(5):734–746, 2018.

[3] D. L. Applegate, R. E. Bixby, V. Chvatál, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[4] E. M. Arkin, Y. Chiang, J. S. B. Mitchell, S. Skiena, and T. Yang. On the maximum scatter traveling salesperson problem. *SIAM J. Comput.*, 29(2):515–544, 1999.

[5] J. Bang-Jensen and G. Z. Gutin. *Digraphs - theory, algorithms and applications*. Springer, 2002.

[6] R. Bellman. Dynamic programming treatment of the travelling salesman problem. *J. Assoc. Comput. Mach.*, 9:61–63, 1962.

[7] N. Brauner, Y. Crama, A. Grigoriev, and J. van de Klundert. A framework for the complexity of high-multiplicity scheduling problems. *J. Combinatorial Optim.*, 9(3):313–323, 2005.

[8] A. Cayley. A theorem on trees. *Quart. J. Pure Appl. Math.*, 23:376–378, 1889.

[9] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Technical Report 388, Carnegie Mellon University, 1976.

[10] W. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2011.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[12] S. S. Cosmadakis and C. H. Papadimitriou. The traveling salesman problem with many visits to few cities. *SIAM J. Comput.*, 13(1):99–108, 1984.

[13] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. In *Combinatorial Structures and their Applications (Proc. Calgary Internat. Conf., Calgary, Alta., 1969)*, pages 93–96. Gordon and Breach, New York, 1970.

[14] H. Emmons and K. Mathur. Lot sizing in a no-wait flow shop. *Oper. Res. Lett.*, 17(4):159–164, 1995.

[15] A. Gabizon, D. Lokshtanov, and M. Pilipczuk. Fast algorithms for parameterized problems with relaxed

disjointness constraints. In *Proc. ESA 2015*, pages 545–556, 2015.

[16] A. Golovnev. Approximating asymmetric TSP in exponential time. *Int. J. Found. Comput. Sci.*, 25(1): 89–100, 2014.

[17] A. Grigoriev and J. van de Klundert. On the high multiplicity traveling salesman problem. *Discrete Optim.*, 3(1):50–62, 2006.

[18] Y. Gurevich and S. Shelah. Expected computation time for Hamiltonian path problem. *SIAM J. Comput.*, 16(3):486–502, 1987.

[19] G. Gutin and A. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, 2002.

[20] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *J. Soc. Indust. Appl. Math.*, 10:196–210, 1962.

[21] F. L. Hitchcock. The distribution of a product from several sources to numerous localities. *J. Math. Phys. Mass. Inst. Tech.*, 20:224–230, 1941.

[22] D. S. Hochbaum and R. Shamir. Strongly polynomial algorithms for the high multiplicity scheduling problem. *Oper. Res.*, 39(4):648–653, 1991.

[23] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.

[24] C. Jordan. Sur les assemblages de lignes. *Journal für die reine und angewandte Mathematik*, 70:185–190, 1869.

[25] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proc. STOC 1983*, pages 193–206, 1983.

[26] S. Kapoor and H. Ramesh. Algorithms for enumerating all spanning trees of undirected and weighted graphs. *SIAM J. Comput.*, 24(2):247–265, 1995.

[27] P. Kleinschmidt and H. Schannath. A strongly polynomial algorithm for the transportation problem. *Math. Program.*, 68:1–13, 1995.

[28] M. Koivisto and P. Parviainen. A space-time tradeoff for permutation problems. In *Proc. SODA 2010*, pages 484–492, 2010.

[29] L. Kozma and T. Mömke. Maximum scatter TSP in doubling metrics. In *Proc. SODA 2017*, pages 143–153, 2017.

[30] M. Lampis. Algorithmic meta-theorems for restrictions of treewidth. *Algorithmica*, 64(1):19–37, 2012.

[31] E. Lawler, D. Shmoys, A. Kan, and J. Lenstra. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[32] H. W. Lenstra, Jr. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8(4):538–548, 1983.

[33] D. Lokshtanov and J. Nederlof. Saving space by algebraization. In *Proc. STOC 2010*, pages 321–330, 2010.

[34] A. Nijenhuis and H. S. Wilf. *Combinatorial algorithms*. Academic Press, Inc. [Harcourt Brace Jovanovich, Publishers], New York-London, second edition, 1978.

[35] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2):338–350, 1993.

[36] C. H. Papadimitriou and M. Yannakakis. On minimal Eulerian graphs. *Inf. Proc. Lett.*, 12(4):203–205, 1981.

[37] H. N. Psaraftis. A dynamic programming approach for sequencing groups of identical jobs. *Oper. Res.*, 28(6): 1347–1359, 1980.

[38] M. Rothkopf. Letter to the editor–the traveling salesman problem: On the reduction of certain large problems to smaller ones. *Oper. Res.*, 14(3):532–533, 1966.

[39] S. C. Sarin, H. D. Sherali, and L. Yao. New formulation for the high multiplicity asymmetric traveling salesman problem with application to the Chesapeake problem. *Optim. Lett.*, 5(2):259–272, 2011.

[40] J. A. A. van der Veen and S. Zhang. Low-complexity algorithms for sequencing jobs with a fixed number of job-classes. *Comput. Oper. Res.*, 23(11):1059–1067, 1996.

[41] G. J. Woeginger. Open problems around exact algorithms. *Discrete Appl. Math.*, 156(3):397–405, 2008.

A Deferred subroutines

Algorithm 7 Generating all possible r-subsets of $\{1, \ldots, n\}$.

```
1: Input: Positive integers n and r.
2: Output: All possible r-subsets of {1, ..., n}.
3: procedure COMBINATIONS(n, r)
4:     comb := [1, ..., r]
5:     yield comb
6:     while true do
7:         i ← last index such that comb_i ≠ n − r + i,
   if no such index exists, break
8:         comb_i := comb_i + 1
9:         for every index j := i+1, ..., r do
10:            comb_j := comb_{j−1} + 1
11:        yield comb
```

The algorithm COMBINATIONS is an implementation of the algorithm NEXKSB by Nijenhuis and Wilf [34, page 26]. It takes two integers as input, n and r, and generates all (ordered) subsets of size r, of the base set $\{1, \ldots, n\}$. It starts with the set $[1, 2, \ldots, r]$ and generates all r-subsets in lexicographical order, up to $[n-r+1, n-r+2, \ldots, n-r+r] = [n-r+1, n-r+2, \ldots, n]$. In every iteration, it increases the rightmost number comb_i not being equal to $n-r+i$ by one, and then makes comb_j equal to $\mathsf{comb}_{j-1}+1$ for all j indices between $[i+1, r]$. The algorithm stops when there are no such indices i, that happens when reaching $[n-r+1, \ldots, n]$.

An example is shown in Fig. 2. The corresponding integer sequence would be $[1, 1, 2, 0, 0]$, however COMBINATIONS returned the sequence $[2, 4, 7, 8]$, that is, a sequence of the positions (separating bars) of the integer sequence above. In order to obtain the actual degree sequence, we use a short script COMBINATIONTOSEQUENCE in Algorithm 8, that converts the sequence with the positions of the bars to the degree sequence.

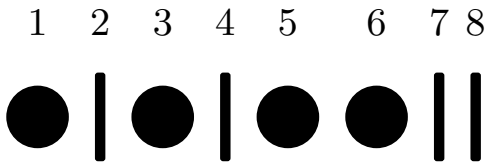

Figure 2: Sequence $[2, 4, 7, 8]$ representing the degree sequence $[1, 1, 2, 0, 0]$

Finally, the procedure DISTRIBUTE(n, k) simply calls COMBINATIONTOSEQUENCE(a, n), for each output a of COMBINATIONS$(n+k, n)$.

Algorithm 8 Converting $[a_1, \ldots, a_m]$ into $[a_1, a_2 - a_1 - 1, \ldots, r + m - a_m]$.

```
1: Input: list of positions a = [a_1, ..., a_m], integer r
2: Output: A sequence of m+1 integers that sum up
   to r.
3: procedure COMBINATIONTOSEQUENCE(a, r)
4:     seq_1 := a_1 − 1
5:     for every index i := 2, ..., m do
6:         seq_i := a_i − a_{i−1} − 1
7:     seq_{m+1} := r + m − a_m
8:     return seq
```