The Dantzig-Fulkerson-Johnson TSP formulation is easy to solve for few subtour constraints

Eleonora Vercesi
Istituto Dalle Molle di studi sull'intelligenza artificiale (IDSIA USI-SUPSI), Faculty of Informatics, Università della Svizzera italiana, eleonora.vercesi@usi.ch

Austin Buchanan
Industrial Engineering & Management, Oklahoma State University buchanan@okstate.edu

Abstract. The most successful approaches for the TSP use the integer programming model proposed in 1954 by Dantzig, Fulkerson, and Johnson (DFJ). Although this model has exponentially many subtour elimination constraints (SECs), it has been observed that relatively few of them are needed to prove optimality in practice. This leads us to wonder: What is the complexity of the DFJ model when just a small number of SECs are imposed? Also, for a given instance, what is the minimum number of SECs required to certify the optimality of one of its TSP tours? We offer both positive and negative results. On the positive side, we give an algorithm to solve the DFJ model that runs in polynomial time when only a constant number of SECs are imposed. With an additional condition, we also *find* a minimum number of SECs in polynomial time. This provides an explanation for the apparent easiness of some TSP instances despite the intractability of the problem in the worst case. As part of our experiments, we show that the original 49-city TSP instance of DFJ requires a minimum of four SECs, which we generate in a fraction of a second with a simple Python implementation.

Key words: Traveling Salesman Problem, Integer Programming, Parameterized Complexity

1. Introduction

A classic approach to solve the Traveling Salesman Problem (TSP) uses the integer programming formulation of Dantzig, Fulkerson, and Johnson (DFJ). This formulation was originally introduced in 1954 to solve a 49-city TSP instance (Dantzig et al. 1954) and still forms the basis of the best solvers (Applegate et al. 2003, 2007).

To describe this model, first let us introduce some notation. Consider a TSP instance given by an undirected graph $G = (V, E)$ (usually complete) on n vertices and m edges, with a nonnegative cost c_e for each edge $e \in E$. Given vertex subsets $S, S' \subseteq V$ that are disjoint, denote by $\delta(S, S')$ the subset of edges with one endpoint in S and one endpoint in S'. We use the shorthand $\delta(S) \coloneqq \delta(S, V \setminus S)$ and, for singleton sets $S = \{v\}$, we write $\delta(v)$ rather than the more cumbersome $\delta(\{v\})$. For $S \subseteq V$, denote

by $E(S)$ the subset of edges with both endpoints in S. Last, we define the *restricted power set* of V as

$$\mathcal{P}'(V) \coloneqq \{S \subseteq V \mid 3 \leq |S| \leq |V| - 3\},$$

which is similar to the usual power set $\mathcal{P}(V)$, but without sets of size $0, 1, 2, |V| - 2, |V| - 1$, or $|V|$.

We are now ready to give the DFJ formulation. Introduce a binary variable x_e for each edge $e \in E$ indicating whether it belongs to the tour. For any subcollection $\mathcal{S} \subseteq \mathcal{P}'(V)$ of the restricted power set, denote by $\mathrm{DFJ}(G, \mathcal{S})$ the objective value of the following integer program, which imposes a subtour elimination constraint (SEC) for each $S \in \mathcal{S}$. In particular, the "full" DFJ formulation imposes *all* of the SECs, i.e., using $\mathcal{S} = \mathcal{P}'(V)$, and the cost of an optimal TSP tour is $\mathrm{TSP}(G) = \mathrm{DFJ}(G, \mathcal{P}'(V))$.

$$\begin{aligned}
\mathrm{DFJ}(G, \mathcal{S}) \coloneqq \min \quad & \sum_{e \in E} c_e x_e && && \text{(1a)}\\
\text{s.t.} \quad & \sum_{e \in \delta(v)} x_e = 2 && \forall v \in V && \text{(1b)}\\
& \sum_{e \in E(S)} x_e \leq |S| - 1 && \forall S \in \mathcal{S} && \text{(1c)}\\
& x_e \in \{0, 1\} && \forall e \in E. && \text{(1d)}
\end{aligned}$$

The objective (1a) minimizes the total cost of the selected edges. The degree-two constraints (1b) ensure that each vertex $v \in V$ touches two selected edges. The SECs (1c) ensure the selection of fewer than $|S|$ edges between the vertices of S, ultimately preventing a cycle over each $S \in \mathcal{S}$. The SECs can equivalently be written in a cut-based fashion:

$$\sum_{e \in \delta(S)} x_e \geq 2 \qquad \forall S \in \mathcal{S}. \qquad \text{(2)}$$

A simple approach for solving the TSP is as follows. Solve the integer program (1) for $\mathcal{S} = \emptyset$, obtaining a solution x^*. If x^* represents a tour, then it is optimal. Otherwise, violated SECs are added to $\mathcal{S}$, the integer program $\mathrm{DFJ}(G, \mathcal{S})$ is re-optimized using branch-and-bound, and the process repeats as necessary. Alternatively, one may dynamically grow the set family $\mathcal{S}$ in a *single*

branch-and-bound tree (rather then solving an integer program from scratch each time) by lazily adding the SECs as cutting planes when encountering integer points x^* that do not represent tours.

In practice, these approaches work remarkably well, despite the NP-hardness of TSP. One possible explanation for this phenomenon is that relatively few SECs are needed to prove optimality, as noted by Miliotis (1976), Pferschy and Staněk (2017). In the extreme case where no SECs are imposed, i.e., $\mathcal{S} = \emptyset$, the resulting formulation models the minimum-weight 2-factor problem, which can be solved in strongly polynomial time (Edmonds 1965), see Grotschel and Holland (1987) and (Schrijver 2003, Ch.30); Gabow (2018) shows it is solvable in time $O(n(m + n \log n))$. This leads us to wonder about the complexity of an intermediate regime in which only a small number of SECs are imposed.

Question 1 What is the complexity of solving DFJ$(G, \mathcal{S})$ as a function of $k = |\mathcal{S}|$? Or, as a function of the sizes $|S|$ of the sets $S \in \mathcal{S}$?

We offer both positive and negative results. On the positive side, we show that the problem is polynomial-time solvable if there is only a constant number k of SECs. Our approach is to reduce the problem to a sequence of minimum-weight 2-factor problems, using a new paw-splitting gadget. As a side result, we show that the minimum-weight triangle-free 2-factor problem is fixed-parameter tractable (FPT) with respect to the number t of triangles in the graph; the running time is $2^t \operatorname{poly}(n)$. Whether this problem is polynomial or NP-hard is a longstanding open question, first posed by Vornberger (1980), that remains open to this day (Schrijver 2003, Kobayashi 2022), making the intermediate FPT status interesting.

On the negative side, we observe that solving DFJ$(G, \mathcal{S})$ remains NP-hard when the number of SECs is polynomial in n. By a reduction of Vornberger (1980), hardness persists even when the vertex subsets $S \in \mathcal{S}$ have size three or four. We extend this result to show that, for any $\varepsilon > 0$, solving DFJ$(G, \mathcal{S})$ remains NP-hard when only n^ε SECs are imposed. We also ask:

Question 2: *For a given TSP instance, what is the minimum number of SECs required to certify the optimality of one of its TSP tours, and how can such constraints be identified?*

For the DFJ model to be *correct*, capturing all TSP tours and nothing more, fully half of the SECs are required. This is because, if we omit the SEC for both S and its complement $V \setminus S$ (which are logically equivalent under the degree-two constraints), then the model will allow cycles over S and $V \setminus S$. In contrast, Question 2 regards a particular instance (i.e., choice of edge costs), asking for a

set family $\mathcal{S} \subseteq \mathcal{P}'(V)$ of minimum size $|\mathcal{S}|$ such that $\mathrm{DFJ}(G, \mathcal{S}) = \mathrm{TSP}(G)$, i.e., that solves

$$\begin{aligned} k^* = \min \quad & |\mathcal{S}| \\ \text{s.t.} \quad & \mathrm{DFJ}(G, \mathcal{S}) = \mathrm{TSP}(G) \\ & \mathcal{S} \subseteq \mathcal{P}'(V). \end{aligned} \tag{3}$$

We give an algorithm to calculate this minimum number k^* of SECs. If k^* is a constant and if all 2-factors encountered by our algorithm have at most b components, then it runs in time $2^{bk^*}\,\mathrm{poly}(n)$, which is polynomial if we further have $b = O(\log n)$. As part of our experiments, we apply the algorithm to the famous 49-city TSP instance from the DFJ paper, finding that the minimum number of SECs needed to prove optimality is $k^* = 4$. This is much smaller than the full number $|\mathcal{P}'(V)|/2$ of SECs needed for the DFJ model to be correct. We also experiment with other TSPLIB, HardTSPLIB, and randomly generated instances, finding that k^* and b are often small (e.g., $b \in \{2, 3, 4\}$), enabling us to find the minimum number of SECs for nearly all TSPLIB instances that have fewer than 100 vertices via a simple Python implementation. Our algorithms, along the with the empirical observation that k^* and b are often small, provide a rigorous explanation for the apparent easiness of some TSP instances despite the intractability of the problem in the worst case.

Outline. Section 2 provides background about the problem and surveys the literature. Section 3 considers Question 1, providing both positive results (e.g., a polynomial-time algorithm for constant k) and negative results (e.g., NP-hardness for $k = n^{\varepsilon}$). Section 4 considers Question 2, also providing positive results (e.g., an exact exponential algorithm) and negative results. Computational experiments follow in Section 5. We conclude in Section 6.

2. Background and Literature Review

Here, we briefly review the literature on 2-factors and the TSP.

2.1. 2-Factors

A 2-factor is a subset of edges such that each vertex touches two of them. The associated minimum-weight 2-factor problem is solvable in strongly polynomial time using matching techniques (Edmonds 1965), see (Schrijver 2003, Ch.30). Indeed, Gabow (2018) shows it is solvable in time $O(n(m + n\log n))$. However, this problem becomes hard when we impose length restrictions on the cycles of the 2-factor, as stated in the following theorem.

THEOREM 1 **(Vornberger (1980), Thm 2.3 and Thm 2.4)**. *Given a graph, determining whether it admits a 2-factor without cycles of length three, four, and five is NP-complete, and finding a*

minimum-weight 2-factor without cycles of length three and four is NP-hard. These problems remain hard when the graph's maximum degree is three.

A closely related problem is the minimum-weight triangle-free 2-factor problem. Its complexity has been an open question since Vornberger's 1980 paper, see also Schrijver (2003). Recently, Kobayashi (2022) mentions its complexity as still being open. However, special cases are polynomial-time solvable, such as in subcubic graphs (Bérczi 2012, Hartvigsen and Li 2013, Kobayashi 2010). Kobayashi (2022) shows that it is polynomial when the (forbidden) triangles are edge-disjoint. The approach is to construct an extended formulation for the problem that has exponentially many constraints and show that the associated separation problem is polynomial-time solvable; thus, this special case is polynomial by the ellipsoid method (Grötschel et al. 1993).

The literature is also rich with papers on the related maximum-weight 2-matching problem, which allows each edge to be chosen once, twice, or not at all and for each vertex to be incident to zero, one, or two edges. We refer the reader to (Schrijver 2003, Ch.30) and Kobayashi (2022) for more.

2.2. TSP

The TSP is well-known to be NP-hard as a consequence of the NP-completeness of the Hamiltonian cycle problem (Karp 1972). Consequently, the problem is intractable, motivating the development of many approximation algorithms, see Traub and Vygen (2024) for a recent survey. Another active research area focuses on the behavior of exact methods, such as cutting-plane and branch-and-cut algorithms. Chvátal et al. (1989) prove that cutting plane algorithms for TSP that start with the DFJ model's LP relaxation and generate only Chvátal-Gomory cuts require at least $2^{n/8}/(3n)$ cuts. Cook and Hartmann (1990) prove that branch-and-cut algorithms for TSP require $\Omega(2^{n/16}/n^2)$ operations, even when there is an oracle for optimizing over the DFJ model's LP relaxation; this result assumes "simple" variable branching, i.e., disjunctions of the form $x_e \leq 0$ or $x_e \geq 1$. Dey et al. (2023) prove that there exist TSP instances for which the smallest *general* branch-and-bound tree (i.e., allowing arbitrary disjunctions of the form $\pi^T x \leq \pi_0$ or $\pi^T x \geq \pi_0 + 1$) has size at least $2^{n/16-2}$, even when the root LP relaxation includes all SECs.

Many practical approaches for TSP apply LP-based branch-and-bound, where initially omitted SECs are added to the model as needed. This includes the famous Concorde implementation (Applegate et al. 2007), which cuts off *fractional* points x^* using SECs (and other types of cuts) and has solved instances with 85,900 cities (Applegate et al. 2003, 2009). Much simpler implementations that cut off only *integer* points can perform quite well for smaller instances, see Pferschy and Staněk

(2017). Each iteration of these integer-separation variants amounts to solving the DFJ model over a subset of the SECs. Our first question asks for the complexity of these subproblems. We note that the implementation of Grötschel and Holland (1991) initially cuts off fractional points before resorting to integer separation in its final phase.

The long history of the TSP contains many combinatorial branch-and-bound algorithms, see the surveys by Bellmore and Nemhauser (1968), Balas and Toth (1983), and Laporte (1992). Most of these methods rely on the assignment problem relaxation and are thus designed for directed edge formulations (also known as *asymmetric* TSP). Bellmore and Nemhauser describe an early approach of Eastman (1958) in which an assignment problem is solved at each iteration, and when a subtour on f nodes is encountered, f subproblems are created, with each subproblem picking one subtour edge and effectively setting its variable to zero by giving it a large cost. These subproblems do not partition the solution space, leading to redundant effort. A better way to branch, that partitions the solution space, is to create f branches across the subtour's edges $e_1, e_2, \ldots, e_f$ like:

- Subproblem 1: $x_{e_1} = 0$;
- Subproblem 2: $x_{e_2} = 0$ and $x_{e_1} = 1$;
- Subproblem 3: $x_{e_3} = 0$ and $x_{e_1} = x_{e_2} = 1$;
- ...
- Subproblem f: $x_{e_f} = 0$ and $x_{e_1} = \cdots = x_{e_{f-1}} = 1$.

Balas and Toth (1983) attribute this branching rule to works as early as Murty (1968). A subsequent branching rule, which Balas and Toth attribute to Garfinkel (1973), considers a subtour over vertices $S = \{1, 2, \ldots, f\}$ and essentially constructs subproblems based on how many edges cross the cut from each vertex of S to $V \setminus S$:

- Subproblem 1: $x_{1j} = 0$ for $j \in S$;
- Subproblem 2: $x_{2j} = 0$ for $j \in S$ and $x_{1j} = 0 \; \forall j \in V \setminus S$;
- Subproblem 3: $x_{3j} = 0$ for $j \in S$ and $x_{1j} = x_{2j} = 0 \; \forall j \in V \setminus S$;
- ...
- Subproblem f: $x_{tj} = 0$ for $j \in S$ and $x_{1j} = \cdots = x_{f-1,j} = 0 \; \forall j \in V \setminus S$.

Garfinkel's rule handles *all* edges from $E(S)$, not just the f edges from the subtour. Our proposed disjunction is similar in spirit, but is tailored to *symmetric* or undirected TSP instances. Importantly, by using the 2-factor relaxation, we avoid subtours of size two; these are known to plague approaches that rely on the assignment relaxation (Bellmore and Nemhauser 1968, Balas and Toth 1983).

3. DFJ Complexity for Few SECs

This section provides positive and negative results concerning Question 1.

3.1. Negative Results

We have already seen Theorem 1, which states that finding a minimum-weight 2-factor without cycles of length three and four is NP-hard. This problem is defined on an undirected graph $G = (V, E)$ that may be missing some edges, i.e., with $E \subsetneq \binom{V}{2}$. Meanwhile, the TSP is defined on a complete graph. So, simply by giving each missing edge from $\binom{V}{2} \setminus E$ an exorbitantly large weight, say $\sum_{e \in E} |c_e|$, we have the following observation.

PROPOSITION 1. *Solving* DFJ$(G, \mathcal{S})$ *is NP-hard, even when restricted to the polynomially many SECs given by* $\mathcal{S} = \{S \subset V \mid 3 \leq |S| \leq 4\}$.

In fact, we can make a stronger statement. Hardness persists for n SECs, or even $n^{0.0001}$ of them.

THEOREM 2. *For every constant* $\varepsilon > 0$*, solving* DFJ$(G, \mathcal{S})$ *is NP-hard, even when the number of SECs is only* $|\mathcal{S}| \leq n^{\varepsilon}$.

Proof. We may assume $\varepsilon < 1$. The reduction is from an instance DFJ$(G, \mathcal{S})$ with $\mathcal{S} = \{S \subset V \mid 3 \leq |S| \leq 4\}$, which is NP-hard by Proposition 1. We have $|\mathcal{S}| = \binom{n}{3} + \binom{n}{4} \leq n^4$, where n is the number of vertices of $G = (V, E)$. The idea is to pad this instance with a large cycle. Specifically, let $h = \lceil 1/\varepsilon \rceil$ and add a cycle with zero cost edges over n^{4h} new vertices. (If one desires the new instance to be a complete graph, then again give missing edges exorbitantly large weights.) This new graph $G' = (V', E')$ has a polynomial number of vertices, $n' = n + n^{4h}$, and we impose the previous SECs (over V only) which number at most n^4 and

$$n^4 = (n^{4h})^{1/h} \leq (n^{4h})^{\varepsilon} \leq (n + n^{4h})^{\varepsilon} = (n')^{\varepsilon}.$$

Every optimal solution to DFJ$(G', \mathcal{S})$ will include our zero-cost cycle over $V' \setminus V$. Discarding this cycle yields an optimal solution for DFJ$(G, \mathcal{S})$. □

3.2. Positive Results

Now for the good news. We show that the DFJ formulation with a constant number of SECs is polynomial-time solvable. The main idea is to reduce the problem to a polynomial number of minimum-weight 2-factor problems. This is accomplished via a new branching rule that is analogous to that of Garfinkel (1973)—but carefully crafted for the undirected case—and a new paw-splitting

gadget to enforce this disjunction. In the following, we denote the sum of the x variables over an edge subset $E' \subseteq E$ by the shorthand $x(E') = \sum_{e \in E'} x_e$.

3.2.1. Disjunction and branching rule First, consider the minimum-weight 2-factor problem with one SEC $x(E(S)) \leq |S| - 1$. In light of the degree-two constraints, this is equivalent to $x(\delta(S)) \geq 2$, thus requiring two (or more) edges to cross the cut from S to $V \setminus S$. The question then is: which node(s) from S will neighbor $V \setminus S$ in our 2-factor? This is not known *a priori*, hence the reason to create subproblems.

One may be tempted to use an undirected version of Garfinkel's rule, say, by selecting a vertex $v \in S$ and branching on the disjunction $x(\delta(v, S)) = 0$ or $x(\delta(v, V \setminus S)) = 0$. However, the issue is that this is not exhaustive for the undirected case; we may also have the third case $x(\delta(v, S)) = x(\delta(v, V \setminus S)) = 1$. This would thus create *three* subproblems, leading to an algorithm with a branching factor of three instead of two. Additionally, the third case, where $x(\delta(v, S)) = x(\delta(v, V \setminus S)) = 1$, cannot be enforced simply by fixing select variables to zero (nor by removing edges).

This motivates our proposed disjunction and branching rule. Specifically, the disjunction is

$$x(\delta(v, V \setminus S)) \leq 0 \quad \text{or} \quad x(\delta(v, V \setminus S)) \geq 1, \tag{4}$$

which is indeed exhaustive and irredundant.

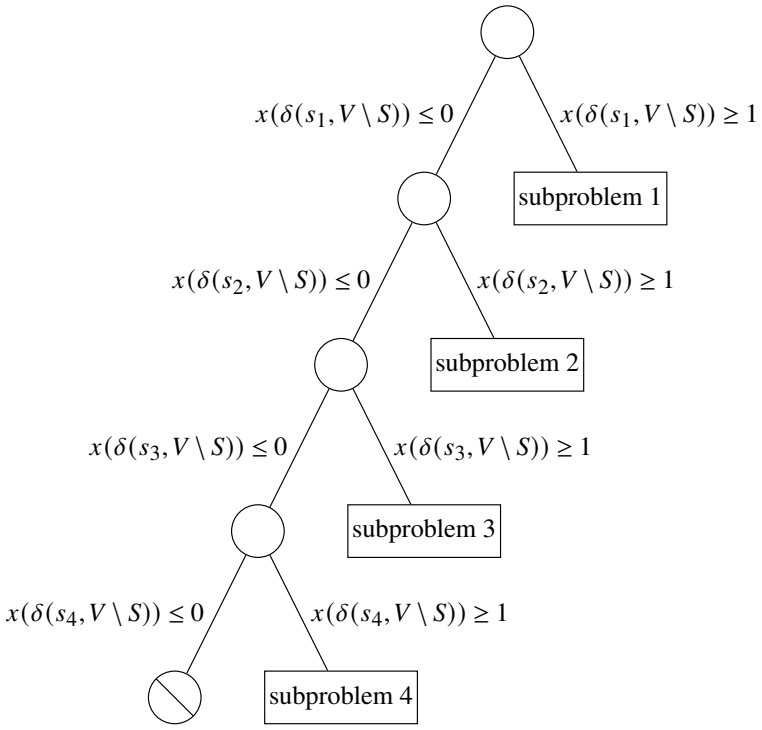
The "atomic" disjunction (4) is applied repeatedly to get the proposed branches for $S = \{s_1, s_2, \ldots, s_f\}$, creating $|S|$ subproblems. Figure 1 illustrates the case $|S| = 4$. Note that the bottom-left branch, where $x(\delta(v, V \setminus S)) \leq 0$ for all $v \in S$, violates the SEC and can be discarded.

Also, in any 2-factor, the number of edges crossing the cut from S to $V \setminus S$ will be even. So, if any of the ≥ 1 inequalities are satisfied, then in fact we will have $x(\delta(S)) \geq 2$. As a consequence, the final subproblem (for s_f) can actually impose $x(\delta(s_f, V \setminus S)) \geq 2$ or equivalently $x(\delta(s_f, S)) = 0$.

For this reason, the final subproblem can sometimes be discarded. For example, if S has just three vertices, i.e., $S = \{s_1, s_2, s_3\}$, then the subproblem for s_3 will necessarily be infeasible. Similarly, if we are interested only in TSP tours, then the final subproblem can also be discarded as any feasible solution to this subproblem will necessarily have one or more subtours over $\{s_1, s_2, \ldots, s_{f-1}\}$.

We remark that any 2-factor satisfying the SEC $x(\delta(S)) \geq 2$ is feasible for precisely one subproblem. This is a consequence of the fact that the disjunction is exhaustive and irredundant. Specifically, suppose that s_j has the smallest index j among the vertices from $S = \{s_1, s_2, \ldots, s_f\}$ that neighbor

Figure 1 **Expanded view of branching rule for** $S = \{s_1, s_2, s_3, s_4\}$

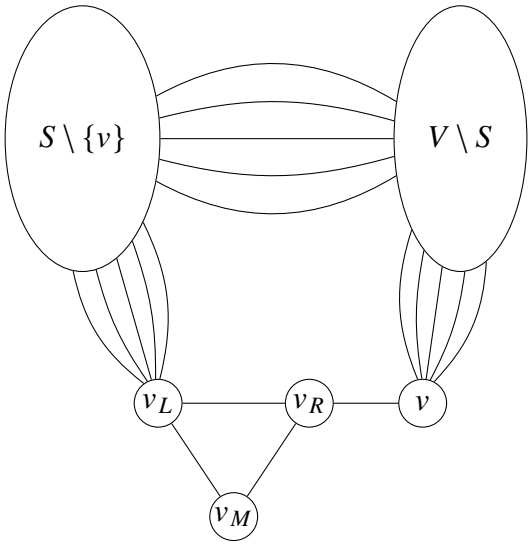


$V \setminus S$ in the 2-factor. Then, this 2-factor is feasible for subproblem j. Meanwhile, it is infeasible for subsequent subproblems, because they are defined in part by the constraint $x(\delta(s_j, V \setminus S)) \leq 0$, which our 2-factor violates. Finally, our 2-factor is infeasible for earlier subproblems, because they are defined in part by constraints of the form $x(\delta(s_i, V \setminus S)) \geq 1$ for $i < j$, but j is the smallest index for which the 2-factor can satisfy such a constraint.

3.2.2. Gadget As we have seen, the subproblems are defined by disjunctions of the form: $x(\delta(v, V \setminus S)) \leq 0$ or $x(\delta(v, V \setminus S)) \geq 1$. We propose graph operations whereby each subproblem can be solved as a minimum-weight 2-factor problem. The $x(\delta(v, V \setminus S)) \leq 0$ branch can be obtained by removing the edges $\delta(v, V \setminus S)$ in an operation that we call *edge deletion*. For the $x(\delta(v, V \setminus S)) \geq 1$ branch, we propose *paw-splitting*, which is similar to node-splitting, but stitches the two sides together using a paw graph, as in Figure 2.

In paw-splitting, we introduce a triangle on three new vertices: v_L, v_M, v_R. Any edges originally extending from $S \setminus \{v\}$ to v are rerouted to v_L instead. Last, we add the edge $\{v_R, v\}$. In total, this adds three new vertices and four zero-cost edges. The reason why this gadget works is because any 2-factor in the new graph is forced to select the two edges incident to v_M. If the 2-factor also picks the edge $\{v_L, v_R\}$ this makes a triangle on the new vertices, forcing v to pick its two neighbors from $V \setminus S$. Otherwise, if edge $\{v_L, v_R\}$ is not selected, then the 2-factor selects edges along the path

Figure 2 **A *paw-splitting* gadget to impose** $x(\delta(v, V \setminus S)) \geq 1$.



v_L-v_M-v_R-v, forcing v to pick its other neighbor from $V \setminus S$ and forcing v_L to pick its other neighbor from $S \setminus \{v\}$. Hence, the gadget edge $\{v_L, v_R\}$ essentially captures the dichotomy $x(\delta(v, V \setminus S)) = 1$ or $x(\delta(v, V \setminus S)) = 2$.

Last, to recover a 2-factor for the original graph, simply take shortcuts through the new vertices and discard any cycles that consist solely of new vertices, a process that we call *short-cutting*. Observe that short-cutting preserves the cost of the 2-factor since only zero-cost edges are removed.

The reverse process also works. That is, given a 2-factor in the original graph, we can easily generate a 2-factor in the new graph with the same cost by adding the appropriate (zero-cost) edges from the paw-splitting gadget. That is, if the original 2-factor has v neighboring two vertices of $V \setminus S$, then simply add the triangle edges from the paw. Otherwise, replace the original edge $\{v, s\}$ between v and a vertex s from $S \setminus \{v\}$ by the four edges $\{s, v_L\}$ and $\{v_L, v_M\}$ and $\{v_M, v_R\}$ and $\{v_R, v\}$.

3.2.3. Algorithm for given subtour constraints In what follows, we propose an algorithm to solve the DFJ formulation for a small number of given SECs. In it, we (repeatedly) apply the edge deletion and paw-splitting operations to create the subproblems. These subproblems are all minimum-weight 2-factor problems, which are polynomial-time solvable. Moreover, the operations either produce smaller instances (for edge deletion) or instances that are larger by only three vertices and four edges (for paw-splitting).

More formally, consider the problem of finding a minimum-weight 2-factor of $G = (V, E)$, with respect to costs c, that satisfies SECs for given vertex subsets $S_1, S_2, \ldots, S_k$. We propose the following recursive algorithm `ralg`. It should be initiated by calling it with the input graph G and the original collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$.

`ralg`$(G, \mathcal{S})$:

- if $\mathcal{S} = \emptyset$, return the cost of a minimum-weight 2-factor of G (or ∞ if infeasible)
- select a vertex subset $S = \{s_1, s_2, \ldots, s_f\}$ from the collection $\mathcal{S}$
- from G, create the edge-weighted graphs $G_1, G_2, \ldots, G_f$ using the paw-splitting and edge deletion operations (Section 3.2.2) for the minimum-weight 2-factor subproblems from Section 3.2.1
- return $\min\{$`ralg`$(G_j, \mathcal{S} \setminus \{S\}) \mid 1 \leq j \leq f\}$

As previously remarked, the last subproblem G_f does not need the paw-splitting gadget, as we can simply remove the edges between s_f and the rest of S.

THEOREM 3. *The algorithm* `ralg` *is correct and solves* $|S_1||S_2|\cdots|S_k|$ *minimum-weight 2-factor problems, each having at most* $n + 3k$ *vertices and* $m + 4k$ *edges.*

Proof. First, we consider the number of minimum-weight 2-factor problems. By the recursive nature of the algorithm, the bottom of the search tree has $|S_1||S_2|\cdots|S_k|$ leaf nodes in which a minimum-weight 2-factor problem is solved (when $\mathcal{S} = \emptyset$). The depth is k, and each level corresponds to another edge deletion or paw-splitting operation, adding at most 3 vertices and 4 edges, so the leaf nodes of the tree have subproblems with at most $3k$ additional vertices and $4k$ additional edges.

For the correctness, consider the optimal objective value z^* and the objective value returned by the algorithm z_{alg}. Observe that the algorithm returns the "best" weight of a minimum-weight 2-factor across all the $|S_1||S_2|\cdots|S_k|$ subproblems. With the short-cutting process, we can find a 2-factor for the original graph G of the same cost. This 2-factor for graph G also satisfies all SECs for the sets $S_1, S_2, \ldots, S_k$ by the subproblem definitions, which require at least one edge to cross the cut from S_j to its complement (and an even number of edges must cross in any 2-factor). This shows that $z^* \leq z_{\text{alg}}$. For the reverse inequality, consider an optimal solution to the original problem. By the subproblem definition and the fact that the disjunction is exhaustive, we can perform the reverse process of short-cutting to obtain a solution for one of the subproblems that has the same objective value, and thus the algorithm can return no worse, i.e., $z_{\text{alg}} \leq z^*$. □

We have a polynomial number $O(n^k)$ of minimum-weight 2-factor problems, and each of them is defined on a graph with $n + O(1)$ vertices and $m + O(1)$ edges. This gives the following corollary.

COROLLARY 1. *We can solve* DFJ$(G, \mathcal{S})$ *in polynomial time when the number of SECs is constant,* $|\mathcal{S}| = O(1)$.

We remark that the number of minimum-weight 2-factor problems in Theorem 3 can be reduced from $\prod_{j=1}^{k} |S_j|$ to $\prod_{j=1}^{k} (|S_j| - 1)$ when the subsets $S_1, S_2, \ldots, S_k$ have size three. The reason, as noted in Section 3.2.1, is that the third subproblem is infeasible. This gives the following corollary.

COROLLARY 2. *The minimum-weight triangle-free 2-factor problem can be solved in time* $2^t \operatorname{poly}(n)$ *where* t *is the number of triangles in the graph.*

As previously mentioned, it is a longstanding open question to determine whether the minimum-weight triangle-free 2-factor problem is polynomial or NP-hard (Vornberger 1980). Our result shows polynomiality when the number of triangles is $O(\log n)$, but the general case remains open.

4. Minimum Number of SECs

This section provides positive and negative results concerning Question 2.

4.1. Negative Results

We show that a very simple class of instances requires more than just a handful of SECs.

PROPOSITION 2. *Some classes of instances require* $\Omega(n)$ *SECs to prove the optimality of a TSP tour. For example, Euclidean instances with* $n \geq 6$ *vertices on a line at* x, y*-coordinates* $(1,0)$, $(2,0)$, $\ldots$, $(n,0)$ *require* $n-5$ *SECs.*

Proof. For each $i \in [n] := \{1, 2, \ldots, n\}$, create a vertex i located at $(i, 0)$. An optimal tour is 1-2-3-$\cdots$-n-1 of cost $2(n-1)$. First, we show that $n-5$ SECs suffice. Specifically, we use the sets $S_1, S_2, \ldots, S_{n-5}$, where $S_j = \{1, 2, \ldots, j+2\}$, i.e., $S_j = [j+2]$. We argue that any 2-factor satisfying these SECs has cost at least $2(n-1)$. For each cycle C of the 2-factor, denote by

$$l(C) = \min\{i \in [n] : i \in V(C)\} \quad \text{and}$$
$$u(C) = \max\{i \in [n] : i \in V(C)\}$$

its minimum and maximum element, respectively. Observe that the weight of cycle C is at least $2(u(C) - l(C))$. We number (a subset of) the 2-factor cycles as follows.

1. Let C_1 be the cycle containing vertex 1.
2. for $t = 1, 2, \ldots$ do
   - if $u(C_t) = n$, then return the cycles $C_1, C_2, \ldots, C_t$
   - Among cycles that have a vertex in the set $\{1, 2, \ldots, u(C_t)\}$, let C_{t+1} be the one with largest maximum element

In the following, we prove that the procedure successfully terminates and that the (disjoint) cycles $C_1, C_2, \ldots, C_t$ from the 2-factor that it finds have a combined weight at least $2(n-1)$, in which case the 2-factor's weight can be no less.

First, suppose that the first cycle C_1 has $u(C_1) = n$. Then, the procedure returns just this cycle, and its weight is at least $2(u(C_1) - l(C_1) = 2(n-1)$, as desired.

Second, suppose that $u(C_1) \in \{n-2, n-1\}$. Then, the weight of C_1 is at least $2(u(C_1) - l(C_1)) \geq 2((n-2)-1)$. Meanwhile, the cycle that contains vertex n must have at least two other vertices, and one or both of them must belong to the set $\{1, 2, \ldots, u(C_1)\}$, in which case the procedure calls this cycle C_2. As cycle C_2 cannot touch the vertex $u(C_1)$ from the first cycle, C_2 in fact contains a vertex from $\{1, 2, \ldots, u(C_1) - 1\}$, in which C_2's weight is at least 6. So, the combined weight of cycles C_1 and C_2 is at least $2((n-2)-1)+6 = 2n$, in fact *greater* than the weight of our optimal tour.

In the third and final case, suppose that $u(C_1) \leq n-3$. In this case, we claim that the procedure successfully returns cycles $C_1, C_2, \ldots, C_t$ and that the inequality $l(C_{j+1}) < u(C_j)$ holds for each $j = 1, 2, \ldots, t-1$. The key idea lies in the procedure's last line. By the SECs for $S_1, S_2, \ldots, S_{n-5}$, there must be at least two edges crossing the cut from $S = \{1, 2, \ldots, u(C_j)\}$ to its complement, meaning that the 2-factor must have a cycle with one vertex in S and another in $V \setminus S$. This shows $l(C_{j+1}) < u(C_j) < u(C_{j+1})$, implying that such a cycle exists and that each iteration makes progress towards the termination criterion $u(C_t) = n$, so the procedure is finite. We also have $l(C_1) = 1$ by line 1. Thus, the cycles $C_1, C_2, \ldots, C_t$ have weight

$$\begin{aligned}
&\sum_{j=1}^{t} w(C_j) \\
&\geq \sum_{j=1}^{t} 2(u(C_j) - l(C_j)) \\
&= 2\left(u(C_t) - l(C_1) + \sum_{j=1}^{t-1} (u(C_j) - l(C_{j+1}))\right) \\
&\geq 2(u(C_t) - l(C_1)) = 2(n-1).
\end{aligned}$$

Last, we show that at least $n-5$ constraints are necessary. Consider the sets $S_1, S_2, \ldots, S_{n-5}$ from before. We claim that for $j \in [n-5]$ the SEC for either S_j or its complement $\overline{S}_j = V \setminus S_j$ is necessary. Indeed, if neither is imposed, then the 2-factor consisting of two cycles 1-2-$\cdots$-$(j+2)$-1 and $(j+3)$-$(j+4)$-$\cdots$-n-$(j+3)$ is feasible with cost $2(n-2)$, less than the TSP cost $2(n-1)$. □

4.2. Positive Results

In the following, we provide an algorithm to determine the minimum number of SECs needed to prove the optimality of a TSP tour. In every iteration, we solve the DFJ formulation for a subset $\mathcal{S}$ of

SECs. If the returned objective value is less than the cost of our optimal TSP tour, then we add a cut of the form $x(E(S)) \leq |S| - 1$. There are many such cuts. Which should we choose? The following lemma tells us precisely which ones will cut off the present 2-factor.

LEMMA 1 **(SECs cutting off a 2-factor)**. *Suppose that the 2-factor defined by x^* is made up of cycles over vertex sets $V_1, V_2, \ldots, V_t$. Then, x^* violates the SEC $x(E(S)) \leq |S| - 1$ if and only if S can be written as the union $S = \cup_{j \in J} V_j$ for some index set $J \subseteq [t]$ with $0 < |J| < t$.*

Proof. ($\Leftarrow$) Suppose $S = \cup_{j \in J} V_j$ for some $J \subseteq [t]$ with $0 < |J| < t$. Then,

$$x^*(E(S)) \geq \sum_{j \in J} x^*(E(V_j)) = \sum_{j \in J} |V_j| = |S|.$$

($\Rightarrow$) Suppose that x^* violates the SEC $x(E(S)) \leq |S| - 1$, implying under the degree-two constraints that $x^*(E(S)) = |S|$ or equivalently $x^*(\delta(S)) = 0$. Consider a vertex set V_j with $j \in [t]$. By assumption, the 2-factor selects a cycle v_1-v_2-v_3-$\cdots$-v_1 in V_j, and none of the selected edges $\{v_i, v_{i+1}\}$ can cross the cut $\delta(S)$ by $x^*(\delta(S)) = 0$. So, if v_1 belongs to S, then so do the other vertices from V_j, and we put j in J. Likewise, if v_1 does not belong to S, then neither do the other vertices from V_j, and we do not put j in J. This gives an index set J for which $S = \cup_{j \in J} V_j$. Moreover, we cannot have $|J| = 0$ or $|J| = t$, because this would imply $S = \emptyset$ or $S = V$, which do not give valid SECs, so $0 < |J| < t$. □

As an easy consequence of Lemma 1, we have the following theorem. Note that by convention $\mathrm{TSP}(G)$ is defined to be infinite when it is infeasible, e.g., when G has fewer than three vertices.

THEOREM 4 **(Characterizing sufficient set families)**. *Let $\mathcal{S} \subseteq \mathcal{P}'(V)$ be a set family. Then, $\mathrm{DFJ}(G, \mathcal{S}) = \mathrm{TSP}(G)$ if and only if for every partition $V_1, V_2, \ldots, V_k$ of V with $\sum_{j=1}^{k} \mathrm{TSP}(G[V_j]) < \mathrm{TSP}(G)$ there is an index set $J \subset [k]$ with $0 < |J| < k$ such that $S := \cup_{j \in J} V_j$ belongs to $\mathcal{S}$.*

By Lemma 1, if the current 2-factor has t cycles, then there are $2^t - 2$ SECs to choose from. We can halve this quantity by observing that, for 2-factors, the constraint $x(E(S)) \leq |S| - 1$ for S is equivalent to that for its complement $\overline{S} = V \setminus S$, so it suffices to consider just one of them. Accordingly, the number of possible constraints can be reduced to $2^{t-1} - 1$. We accomplish this by restricting ourselves to index sets J for which $S = \cup_{j \in J} V_j$ has size $0 < |S| < |V|/2$ (or $|S| = |V|/2$ and S contains an arbitrary vertex, say, vertex 1).

This leads to our second algorithm, called `ialg`. As written, it returns the minimum number $k^* = |\mathcal{S}|$ of necessary SECs, but it could just as easily return the set family $\mathcal{S}$ itself. It starts with

zero SECs, i.e., $\mathcal{S} = \{\}$, and branches over the disjunction from Lemma 1. The search proceeds in a breadth-first manner, which we accomplish by storing the subproblems in a queue Q.

`ialg`():

- create an empty queue Q and then Q.`enqueue`($\{\}$)
- while Q is not empty
  1. $\mathcal{S} \leftarrow Q$.`dequeue`()
  2. compute a solution to DFJ$(G, \mathcal{S})$
  3. if DFJ$(G, \mathcal{S})$ = TSP(G), then return $|\mathcal{S}|$
  4. find the vertex sets $V_1, V_2, \ldots, V_t$ of the DFJ solution over $\mathcal{S}$
  5. for $J \subseteq [t]$ do
     — $S \leftarrow \cup_{j \in J} V_j$
     — if $0 < |S| < |V|/2$ or ($|S| = |V|/2$ and $1 \in S$) then Q.`enqueue`($\mathcal{S} \cup \{S\}$)

Note that the algorithm creates a branch for essentially every subset of subtours; one may question whether such substantial effort is truly necessary. Appendix A motivates this choice and explains why simpler alternatives do not work. We remark that the objective DFJ$(G, \mathcal{S})$ in line 3 is *at most* TSP(G) because the SECs added in line 1 are all valid for TSP. If DFJ$(G, \mathcal{S})$ is *strictly less* than TSP(G), then the 2-factor cannot be a TSP tour, and hence consists of cycles over multiple vertex sets $V_1, V_2, \ldots, V_t$. This 2-factor must be excluded, and the only SECs that would do the trick take the form $x(E(S)) \leq |S| - 1$ with $S = \cup_{j \in J} V_j$, by Lemma 1, hence the branching under line 5. This disjunction is key to the correctness. Otherwise, if DFJ$(G, \mathcal{S})$ *equals* TSP(G), then the selected SECs suffice to prove the optimality of our TSP tour. Moreover, the number $|\mathcal{S}|$ is minimum by the algorithm's queue implementation, recognizing that nodes at depth d involve d SECs.

THEOREM 5. *The algorithm* `ialg` *correctly determines the minimum number k^* of SECs required to prove optimality of a given TSP tour. If k^* is a constant, then it can be implemented to run in time 2^{bk^*} poly(n) where b is the most components encountered in line 4.*

Proof. The algorithm is correct by the disjunction proven in Lemma 1. Next, suppose that k^* is a constant. The branching factor is less than $2^t \leq 2^b$. So, at depth k^*, we have at most 2^{bk^*} DFJ subproblems. Then, by Theorem 3, these DFJ problems can be solved as at most n^{k^*} minimum-weight 2-factor problems, each having at most $n + 3k^*$ vertices and $m + 4k^*$ edges. Thus, each DFJ subproblem can be solved in time polynomial in n. □

COROLLARY 3. *If k^* is a constant and $b = O(\log n)$, then the running time of* `ialg` *is polynomial.*

It may be that the solution to line 2 is not a TSP tour even when the objective equals $\text{TSP}(G)$. In this case, we have alternative optima, with one of them being our TSP tour. This is acceptable because our stated task was to prove the optimality of a given tour, and not to produce it. Appendix B considers the alternative problem setup, where the SECs must produce a tour.

4.2.1. Implementation tweaks When implementing `ialg`, we make a few tweaks. First, we observe that the MIP solver Gurobi (2023) can solve the $\text{DFJ}(G, \mathcal{S})$ subproblems quite quickly in practice. So, for implementation simplicity, we use it rather than the `ralg`() algorithm (which would also require a minimum-weight 2-factor subroutine). Second, we observe that the number of subproblems solved by `ialg` is 2^{bk^*}, which grows exponentially in b and k^*. While the value k^* depends on the TSP instance and cannot be changed, the value b is somewhat malleable.

Specifically, consider a vertex partition $V_1, V_2, \ldots, V_t$ in line 4. The reason we branch in line 5 is because the associated 2-factor has cost less than $\text{TSP}(G)$, and so we must cut it off. What would be great is an alternative 2-factor over vertex sets $V'_1, V'_2, \ldots, V'_{t'}$ that *also* has cost less than $\text{TSP}(G)$ and that has *fewer* components, i.e., with $t' < t$. If we could find such a 2-factor, we could branch on it instead and create fewer subproblems. Moreover, if the vertex sets $V'_1, V'_2, \ldots, V'_{t'}$ were somehow obtained by merging parts of $V_1, V_2, \ldots, V_t$ together, then branching on the former would also cut off the latter (see Lemma 1). In this sense, it would be a stronger disjunction. Ideally, we reduce $V_1, V_2, \ldots, V_t$ down to just two parts V'_1, V'_2, in which case line 5 creates only one subproblem(!).

The question then becomes, how should we go about merging parts together? In our implementation, we simply try all possibilities, starting with all ways to throw the vertex subsets $V_1, V_2, \ldots, V_t$ into two parts V'_1, V'_2, then into three parts V'_1, V'_2, V'_3, etc, until we find a satisfactory alternative partition (possibly resorting to the original one). By satisfactory, we mean that it satisfies the condition from Theorem 4, i.e., that the partition $V'_1, V'_2, \ldots, V'_{t'}$ has

$$\sum_{j=1}^{t'} \text{TSP}(G[V'_j]) < \text{TSP}(G). \tag{5}$$

Ultimately, trying all possibilities like this is not too costly because the 2-factors that solve $\text{DFJ}(G, \mathcal{S})$ already tend to have few components.

For speed reasons, we do not compute the values $\text{TSP}(G[V'_j])$ exactly; a heuristic will do. Initial experiments with polynomial-time heuristics such as that of Christofides (1976) did not perform well enough in terms of solution quality for our use case, so we use the well-known LKH heuristic (Helsgaun 2000, 2009, Tinós et al. 2018, Taillard and Helsgaun 2019) via the `elkai` Python wrapper (Dimitrovski 2023). This and other local search heuristics generally do not run in polynomial time (Papadimitriou and Steiglitz 1977, Papadimitriou 1992), but can be very fast in practice (Bentley 1992, Johnson and McGeoch 2007). For each $J \subseteq [t]$ with $0 < |J| < t$, we cache the heuristic estimate of $\text{TSP}(G[S])$ for $S = \cup_{j \in J} V_j$, allowing us to instantly check condition (5).

Additionally, we try to initialize the approach in a smart way. In the pseudocode, we simply initialize the queue Q with a single subproblem for the empty set family $\mathcal{S} = \{\}$. Meanwhile, in our implementation, we (try to) initialize it with a non-empty set family of "required" SECs. Namely, we take our optimal TSP tour and remove two of its edges, thus creating two paths. We stitch each path's endpoints u and v together with the edge $\{u, v\}$ to create a 2-factor consisting of two cycles C_1 and C_2. If this 2-factor has cost less than $\text{TSP}(G)$, then the SEC for either $S = V(C_1)$ or $S = V(C_2)$ is required, and these SECs are equivalent, and so we can add the smaller one to the root subproblem's set family $\mathcal{S}$. We repeat this procedure for all pairs of edges from the TSP tour whose removal gives two paths with at least two edges (since a path with fewer than two edges cannot be stitched back together to make a 2-factor). This smart initialization takes polynomial time for a given TSP tour.

The pseudocode uses a queue so that subproblems are processed in a breadth-first manner. In our implementation, we instead use a priority queue where a parent node with set family $\mathcal{S}$ gives a priority score of $M|\mathcal{S}| + \text{DFJ}(G, \mathcal{S})$ to each of its children, where M is a sufficiently large value such as $\sum_{e \in E} c_e$. This ensures that the subproblems are still processed in a breadth-first manner but prioritizes subproblems whose parent had largest cost, i.e., closest to the target cost $\text{TSP}(G)$.

Finally, we store every vertex partition that `ialg` branches on. This is for two reasons. First, we may be able to re-use the same vertex partition in different parts of the search tree. So, rather than computing one from scratch, we first check if any of the vertex partitions from the "partition pool" will suffice, analogous to how MIP solvers store a cut pool. Second, the partition pool can be used to certify the value k^* at the algorithm's termination, as we will see next.

4.2.2. Certifying optimality Here, we provide an independent way to certify that the output of `ialg` is optimal. Recall that problem (3) seeks a set family $\mathcal{S} \subseteq \mathcal{P}'(V)$ of minimum cardinality k^* such that $\text{DFJ}(G, \mathcal{S}) = \text{TSP}(G)$. When `ialg` terminates, we should have such a set family $\mathcal{S}$. To check $k^* \leq |\mathcal{S}|$, we can confirm that $\mathcal{S}$ is feasible for problem (3), i.e., compute $\text{DFJ}(G, \mathcal{S})$ and

check that it equals $\mathrm{TSP}(G)$. But how can we check that the reverse inequality $k^* \geq |\mathcal{S}|$? For this, recall Theorem 4, which says that $\mathrm{DFJ}(G,\mathcal{S}) = \mathrm{TSP}(G)$ if and only if for every vertex partition $V_1, V_2, \ldots, V_k$ with $\sum_{j=1}^{k} \mathrm{TSP}(G[V_j]) < \mathrm{TSP}(G)$ there is an index set $J \subset [k]$ with $0 < |J| < k$ such that $S := \cup_{j\in J} V_j$ belongs to $\mathcal{S}$. This theorem allows us to rewrite the k^* problem as an integer program with exponentially many variables and exponentially many constraints. Namely, for every set $S \in \mathcal{P}'(V)$, we introduce a binary variable y_S indicating whether S is included in set family $\mathcal{S}$.

$$\min \sum_{S\in\mathcal{P}'(V)} y_S \tag{6a}$$

$$\sum_{S\in U(V_1,V_2,\ldots,V_k)} y_S \geq 1 \qquad \forall (V_1, V_2, \ldots, V_k) \in \Pi_<(V) \tag{6b}$$

$$y_S \in \{0,1\} \qquad \forall S \in \mathcal{P}'(V). \tag{6c}$$

The objective (6a) minimizes the number of sets S in set family $\mathcal{S}$. Denote by $\Pi_<(V)$ the collection of all partitions $(V_1, V_2, \ldots, V_k)$ of V that have $\sum_{j=1}^{k} \mathrm{TSP}(G[V_j]) < \mathrm{TSP}(G)$. (This implies that each part V_j has at least three nodes, since otherwise $\mathrm{TSP}(G[V_j])$ is defined to be infinite.) For each such partition, we write a cover constraint (6b). Here the sum is taken over all sets $S \in U(V_1, V_2, \ldots, V_k)$ whose SEC would cut off the partition, where $U(V_1, V_2, \ldots, V_k)$ is the collection $\{\cup_{j\in J} V_j \mid J \subset [k],\ 0 < |J| < k\}$ of sets obtained by unioning parts of the partition (avoiding $\emptyset$ and V).

Clearly, if we optimize model (6) over only a subset of the constraints (6b), then this is a relaxation and its objective can be no larger than k^*. In particular, if we impose constraints (6b) only for the the partitions that `ialg` branched on, then the associated objective value k' of the relaxed model (6) satisfies $k' \leq k^*$. And, indeed the set family $\mathcal{S}$ from `ialg` has $|\mathcal{S}| = k'$, certifying that $|\mathcal{S}| \leq k^*$.

This provides an alternative way to view `ialg`. Essentially, it is solving the exponentially large integer program (6) with a lazy version of branch-and-price. The `ialg` algorithm repeatedly identifies a violated constraint (6b) for some vertex partition $(V_1, V_2, \ldots, V_k) \in \Pi_<(V)$, introduces the associated y_S variables, and then creates a separate branch for each variable, fixing it to one. This disjunction is different than those used by most branch-and-price algorithms, but is useful for our purposes as the child nodes are each closer (by one unit) to achieving the optimal objective value k^*.

5. Computational Experiments

In our experiments, we seek to answer several questions: (i) How many SECs are needed for famous instances from TSPLIB, like `dantzig42`? (ii) Is it true that very challenging instances, such as those

Table 1 Results for TSPLIB instances (< 100 vertices) in 3600s time limit (TL).

			Minimalization			No Minimalization						Minimalization			No Minimalization		
Instance	n	#SECs	b'	time(s)	#nodes	b	time(s)	#nodes	Instance	n	#SECs	b'	time(s)	#nodes	b	time(s)	#nodes
burma14	14	2	2	0.00	1	2	0.00	1	att48	48	10	3	1.72	6	6	20.38	2034
ulysses16	16	4	2	0.00	1	2	0.00	1	gr48	48	11	3	3.12	11	5	732.42	39790
gr17	17	5	2	0.10	5	4	0.25	187	hk48	48	8	2	1.44	8	7	269.27	46644
gr21	21	0	0	0.01	1	0	0.06	1	eil51	51	2	2	0.95	3	4	0.16	7
ulysses22	22	5	2	0.04	2	2	0.01	2	berlin52	52	2	2	0.43	2	6	0.16	18
gr24	24	1	2	0.05	2	4	0.02	5	brazil58	58	11	6	985.41	131	TL	TL	TL
fri26	26	4	4	1.23	6	5	0.08	18	st70	70	12	6	81.89	8	TL	TL	TL
bayg29	29	4	2	0.05	4	3	0.06	10	eil76	76	2	2	1.05	3	3	0.28	7
bays29	29	5	3	0.13	5	3	0.03	5	pr76	76	TL	TL	TL	TL	TL	TL	TL
dantzig42	42	4	2	0.08	3	2	0.09	3	gr96	96	20	5	299.24	74	TL	TL	TL
swiss42	42	3	3	1.35	4	10	28.75	2802	rat99	99	7	4	61.59	65	7	655.88	18704

from HardTSPLIB, require more SECs? (iii) How well does the `ialg` algorithm perform in practice, and do the implementation tweaks from Section 4.2.1 have a significant impact? Appendix C provides experiments with random instances. Appendix D considers the Rectilinear 3-Dimensional instances of Zhong (2021), which are hard for Concorde; empirically, they seem to require $\Theta(n^2)$ SECs.

All experiments are conducted on a Linux computer with twenty 13th Gen Intel i5 CPUs (5GHz) and 30GB of RAM. Our code is written in Python 3.10.4. MIPs are solved with Gurobi (2023). Like all commercial MIP solvers, Gurobi works in floating-point arithmetic, making it conceivable that the reported solutions and objective values could be incorrect; compare this to Concorde's ability to use the rational arithmetic LP solver `QSopt_ex`. However, we do not expect such issues to arise in our experiments given that we solve relatively small instances ($n < 100$ cities) that are numerically well-behaved (e.g., small integer cost coefficients, 0-1 constraint matrices, 0-1 variables). We also check the integrity of our solutions via the set cover MIP (6), as in Section 4.2.2.

5.1. TSPLIB

TSPLIB is a well-known library of TSP instances commonly used for benchmarking (Reinelt 1991). We tested the `ialg` algorithm on instances from TSPLIB, motivated both by curiosity and by a desire to evaluate the algorithm's performance.

Table 1 gives results for TSPLIB instances with fewer than 100 vertices. (Beyond this size, our machine often fails to solve in one hour.) We find that many well-known instances require only a handful of SECs. This includes the original 49-city instance from Dantzig, Fulkerson, and Johnson, which requires four SECs, as shown in Figure 3. (Note that the DFJ instance originally had 49 cities but was simplified by DFJ to 42.). Instances of this size or smaller require at most five SECs, and `gr21` requires none. Larger instances typically require more, including `gr96` which requires 20 SECs. However, `eil51`, `berlin52`, and `eil76` only need two SECs.

Figure 3 **A minimum collection of four SECs for the 49-city instance from Dantzig, Fulkerson, and Johnson. The two sets provided by the smart initialization are represented with dashed lines.**

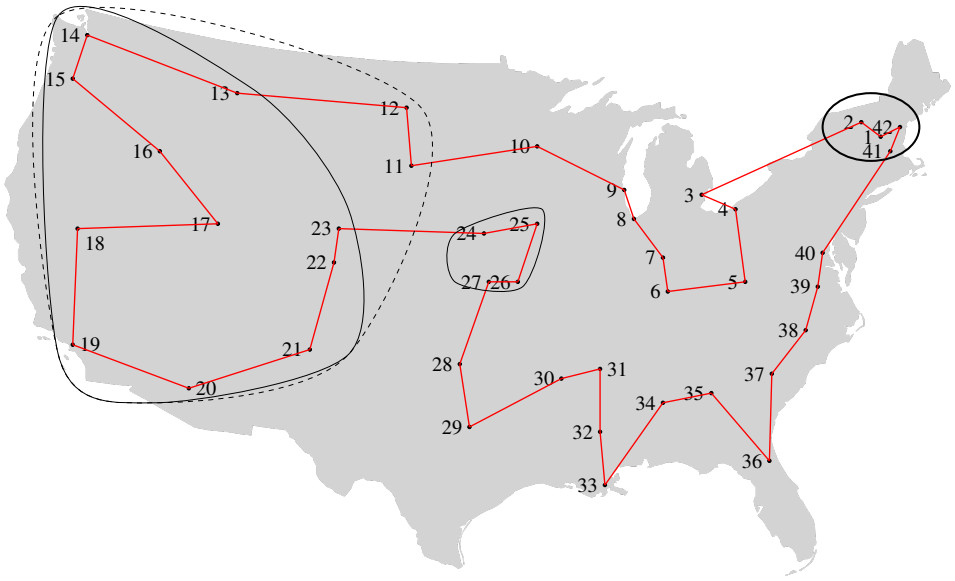


Table 1 also summarizes the performance of the `ialg` algorithm, with and without the implementation tweak "minimalization" that strengthens the disjunction by reducing the partition size (i.e., number of components of each 2-factor). Without this tweak, the algorithm encounters 2-factors with up to b components, with b typically being in the single digits, although growing to $b = 10$ for `swiss42`. Meanwhile, with this tweak, the number of components is at most b', e.g., reducing to $b' = 3$ for `swiss42`. For more than half of the instances (12/22), b' is two or smaller. In this case, each subproblem creates at most one child node, causing the search tree to have linear size (in k^*). Further, 82% of the instances (18/22) have $b' \leq 4$, and 77% (17/22) are solved in ten nodes or less.

In total, 95% of the instances (21/22) are solved with the minimalization tweak, while 82% (18/22) are solved without it. The running times also favor minimalization, sometimes by two orders of magnitude. For example, see `gr48` (3.12s vs 732.42s) and `hk48` (1.44s vs 269.27s). We conclude that the minimalization tweak is quite helpful in practice.

Next we evaluate smart initialization, which is where we take an optimal TSP tour, drop two edges, and add back two others in an attempt to find a 2-factor (with two components) whose cost is less than TSP(G). This allows to quickly identify a collection of required SECs, and is analogous to adding cutting planes at the root node in a branch-and-cut algorithm.

To illustrate, we consider `dantzig42` for historical reasons and `brazil58` because it is "hardest" among the solved instances. For `dantzig42`, we find two initial SECs, shown with dashed lines

Table 2 TSPLIB vs HardTSPLIB

Instance	TSPLIB				HardTSPLIB			
	#SECs	b'	runtime	B&B nodes	#SECs	b'	runtime	B&B nodes
gr24	1	2	0.06	2	63	2	6.56	55
bayg29	4	2	0.05	4	134	2	119.70	133
bays29	5	3	0.15	5	125	2	54.73	113

in Figure 3. After adding these two SECs, the next two 2-factors that the `ialg` algorithm encounters have just two components, in which case minimalization is not needed.

For `brazil58`, we find four initial SECs. After adding them, the next 2-factor that the `ialg` algorithm encounters has $t = 14$ components. Without minimalization, this creates $2^{t-1} - 1 = 8191$ child nodes. With minimalization, these 14 components can be merged into $t' = 2$ components, leading to just $2^{t'-1} - 1 = 1$ child node. (This is one reason why `brazil58` is unsolved in Table 1 if minimalization is not used.) Now, suppose that minimalization is used, but that the smart initialization is either turned on or off. In the former case, `ialg` solves in 131 nodes after 1310.93s. In the latter case, it solves in 500 nodes after 1347.53s. This shows that the smart initialization can have a positive impact on the size of the search tree and the total running time, although less than minimalization.

5.2. HardTSPLIB

Recently, Vercesi et al. (2023) introduced HardTSPLIB, featuring 41 small metric TSP instances that pose significant runtime challenges for the state-of-the-art solver Concorde. HardTSPLIB includes two types of instances: (i) those derived by "increasing the difficulty" of TSPLIB instances, and (ii) those created by "increasing the difficulty" of randomly generated instances. We want to investigate if there is a substantial difference between the "usual" and "hard" versions of these instances in terms of the number of SECs required. For simplicity, we consider only the `ialg` algorithm with all proposed implementation tweaks, including minimalization and smart initialization.

Table 2 provides results for the first type of HardTSPLIB instances. We see that they require many more SECs than their TSPLIB counterparts, sometimes by two orders of magnitude. Further, `ialg` is considerably slower on them, which is why we give results only for the three smallest instances. This suggests that these instances are "hard" in more than one way; they are hard not *just* for Concorde.

Next, we turn to HardTSPLIB instances of the second type, which were randomly generated and then adjusted to have a large integrality gap. Table 3 gives results. To our astonishment, all instances have $b' = 2$, meaning that every 2-factor encountered by the algorithm could be reduced to just two components. Again, this implies that the number of search nodes of the `ialg` algorithm will be linear in k^*. Without this property, it is unlikely that `ialg` would terminate successfully

Table 3 **Results for the HardTSPLIB instances derived from randomly generated instances.**

Instance	#SECs	b'	time(s)	#nodes	Instance	#SECs	b'	time(s)	#nodes
Random 10 nodes	7	2	0.02	6	Random 20 nodes	50	2	3.36	39
Random 10 nodes	6	2	0.01	5	Random 20 nodes	42	2	2.29	37
Random 10 nodes	7	2	0.01	5	Random 25 nodes	84	2	19.30	74
Random 10 nodes	7	2	0.02	5	Random 25 nodes	84	2	24.25	74
Random 11 nodes	8	2	0.04	8	Random 25 nodes	76	2	16.43	64
Random 12 nodes	13	2	0.13	12	Random 30 nodes	135	2	100.36	126
Random 14 nodes	19	2	0.25	19	Random 30 nodes	235	2	1625.91	221
Random 15 nodes	22	2	0.38	18	Random 30 nodes	160	2	280.27	139
Random 15 nodes	21	2	0.38	18	Random 33 nodes	171	2	263.79	157
Random 15 nodes	27	2	0.64	22	Random 35 nodes	167	2	827.20	151
Random 16 nodes	26	2	0.46	22	Random 35 nodes	197	2	2807.76	188
Random 20 nodes	56	2	4.82	47	Random 35 nodes	TL	TL	TL	TL
Random 20 nodes	54	2	4.41	51					

on the largest instances (with just 35 vertices) that require nearly 200 SECs. Compare this to the larger real-life instance `dantzig42`, which required only 4 SECs.

6. Conclusion

This paper is motivated by the empirical observation that few subtour elimination constraints (SECs) are needed to solve the DFJ model in practice. In the extreme case where no SECs are necessary, TSP is solvable in time $O(n^3)$ as a minimum-weight 2-factor problem. This prompted two questions.

First, we ask: What is the complexity of optimizing over the DFJ model when just k SECs are imposed? We find that if $k = O(1)$ is a constant, then the problem is polynomial-time solvable, which we prove with help from a new paw-splitting gadget that effectively imposes the constraint $x(\delta(v, V \setminus S)) \geq 1$. However, for any fixed constant $\varepsilon > 0$, the problem is NP-hard for $k = n^{\varepsilon}$ SECs, even when they are written for vertex subsets of size three and four.

Second, we ask: What is the minimum number of SECs needed to prove the optimality of a TSP tour, and how should they be found? We show that straightforward approaches, such as repeatedly adding a violated SEC for a single (smallest) subtour, do not work. For some classes of instances, this simple approach adds essentially *all* SECs even though one SEC would suffice. Our proposed algorithm, which is exact, instead branches over *subsets of subtours* and runs in time $2^{bk^*}\,\mathrm{poly}(n)$ if the minimum number k^* of SECs is a constant, where b is the (maximum) number of components or subtours in the 2-factors encountered by the algorithm. Consequently, if $b = O(\log n)$, then the algorithm is polynomial. To achieve this worst-case running time, we use as a subroutine the proposed algorithm for Question 1. However, for better practical performance, this subroutine is replaced by a MIP solver. Further, a "subtour merging" procedure is used to generate stronger disjunctions, ultimately reducing the algorithm's branching factor from b to b'.

We test this approach on familiar (classes of) TSP instances, such as the original instance considered by Dantzig, Fulkerson, and Johnson, finding the minimum number of SECs that would suffice for them. We find that more than half of TSPLIB instances have $b' \leq 2$, meaning that our `ialg` algorithm visits a linear number ($\leq k^* + 1$) of search nodes. We observe that randomly generated Euclidean instances require more SECs than those in which costs are drawn randomly from either an exponential or uniform distribution. Additionally, instances from HardTSPLIB require *substantially* more SECs than those from TSPLIB, analogous to how solvers like Concorde that are fast on TSPLIB are challenged by HardTSPLIB.

Future work may consider the following additional questions. Is the problem of optimizing over the DFJ model with k SECs fixed-parameter tractable (FPT) with respect to k? For example, can it be solved in time $2^k \operatorname{poly}(n)$? How many SECs are necessary in the worst-case to prove the optimality of a tour? Can one construct a class of instances that requires $k^* = 2^{\Omega(n)}$ many SECs? (As an aside, Kaibel and Weltge (2015) show that any TSP formulation in the original space of variables requires $2^{\Omega(n)}$ constraints to get a correct formulation, although this does not answer our question.) Do the rectilinear 3-dimensional instances of Zhong (2021) indeed require $k^* = \Omega(n^2)$ SECs? Is computing k^* complete for the second level of the polynomial hierarchy? For randomly generated Euclidean instances in the unit square, how is k^* expected to grow in terms of n?

References

Applegate D, Bixby R, Chvátal V, Cook W (2003) Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Mathematical Programming* 97(1):91–153.

Applegate DL, Bixby RE, Chvátal V, Cook W, Espinoza DG, Goycoolea M, Helsgaun K (2009) Certification of an optimal TSP tour through 85,900 cities. *Operations Research Letters* 37(1):11–15.

Applegate DL, Bixby RE, Chvátal V, Cook WJ (2007) *The Traveling Salesman Problem: A Computational Study* (Princeton University Press).

Balas E, Toth P (1983) Branch and bound methods for the traveling salesman problem. Technical report, CMU.

Bellmore M, Nemhauser GL (1968) The traveling salesman problem: a survey. *Operations Research* 16(3):538–558.

Bentley JL (1992) Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing* 4(4):387–411.

Bérczi K (2012) The triangle-free 2-matching polytope of subcubic graphs. Technical Report TR-2012-02, Egerváry Research Group on Combinatorial Optimization, Budapest, Hungary.

Christofides N (1976) Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, CMU.

Chvátal V, Cook W, Hartmann M (1989) On cutting-plane proofs in combinatorial optimization. *Linear Algebra and its Applications* 114:455–499.

Cook WJ, Hartmann M (1990) On the complexity of branch and cut methods for the traveling salesman problem. Cook WJ, Seymour P, eds., *Polyhedral Combinatorics*, volume 1, 75–82 (American Mathematical Society).

Dantzig G, Fulkerson R, Johnson S, et al. (1954) Solution of a large-scale traveling-salesman problem. *Operations Research* 2(4):393–410.

Dey SS, Dubey Y, Molinaro M (2023) Lower bounds on the size of general branch-and-bound trees. *Mathematical Programming* 198(1):539–559.

Dimitrovski F (2023) elkai - a Python library for solving TSP problems. URL `https://pypi.org/project/elkai/`.

Eastman WL (1958) *Linear programming with pattern constraints: a thesis*. Ph.D. thesis, Harvard University.

Edmonds J (1965) Maximum matching and a polyhedron with 0, 1-vertices. *Journal of Research of the National Bureau of Standards B* 69:125–130.

Gabow HN (2018) Data structures for weighted matching and extensions to b-matching and f-factors. *ACM Transactions on Algorithms (TALG)* 14(3):1–80.

Garfinkel RS (1973) On partitioning the feasible set in a branch-and-bound algorithm for the asymmetric traveling-salesman problem. *Operations Research* 21(1):340–343.

Grotschel M, Holland O (1987) A cutting plane algorithm for minimum perfect 2-matchings. *Computing* 39(4):327–344.

Grötschel M, Holland O (1991) Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming* 51(1):141–202.

Grötschel M, Lovász L, Schrijver A (1993) *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics* (Berlin: Springer-Verlag), second edition.

Gurobi (2023) Gurobi Optimizer Reference Manual. URL `https://www.gurobi.com`.

Hartvigsen D, Li Y (2013) Polyhedron of triangle-free simple 2-matchings in subcubic graphs. *Mathematical Programming* 138:43–82.

Helsgaun K (2000) An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research* 126(1):106–130.

Helsgaun K (2009) General k-opt submoves for the Lin–Kernighan TSP heuristic. *Mathematical Programming Computation* 1:119–163.

Johnson DS, McGeoch LA (2007) Experimental analysis of heuristics for the STSP. *The traveling salesman problem and its variations*, 369–443 (Springer).

Kaibel V, Weltge S (2015) Lower bounds on the sizes of integer programs without additional variables. *Mathematical Programming* 154(1):407–425.

Karp RM (1972) Reducibility among combinatorial problems. *Complexity of Computer Computations*, 85–103 (Springer).

Kobayashi Y (2010) A simple algorithm for finding a maximum triangle-free 2-matching in subcubic graphs. *Discrete Optimization* 7(4):197–202.

Kobayashi Y (2022) Weighted triangle-free 2-matching problem with edge-disjoint forbidden triangles. *Mathematical Programming* 192(1):675–702.

Laporte G (1992) The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research* 59(2):231–247.

Miliotis P (1976) Integer programming approaches to the travelling salesman problem. *Mathematical Programming* 10(1):367–378.

Murty KG (1968) An algorithm for ranking all the assignments in order of increasing cost. *Operations Research* 16(3):682–687.

Osorio MA, Pinto D (2003) Hard and easy to solve TSP instances. *XXX Aniversario del Programa Educativo de Computación, BUAP* 26–30.

Papadimitriou CH (1992) The complexity of the Lin–Kernighan heuristic for the traveling salesman problem. *SIAM Journal on Computing* 21(3):450–465.

Papadimitriou CH, Steiglitz K (1977) On the complexity of local search for the traveling salesman problem. *SIAM Journal on Computing* 6(1):76–83.

Pferschy U, Staněk R (2017) Generating subtour elimination constraints for the TSP from pure integer solutions. *Central European Journal of Operations Research* 25(1):231–260.

Reinelt G (1991) TSPLIB—a traveling salesman problem library. *ORSA Journal on Computing* 3(4):376–384.

Schrijver A (2003) *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics* (Springer).

Taillard ÉD, Helsgaun K (2019) POPMUSIC for the travelling salesman problem. *European Journal of Operational Research* 272(2):420–429.

Tinós R, Helsgaun K, Whitley D (2018) Efficient recombination in the Lin-Kernighan-Helsgaun traveling salesman heuristic. *PPSN XV: Coimbra, Portugal, September 8–12, 2018, Proceedings, Part I 15*, 95–107 (Springer).

Traub V, Vygen J (2024) *Approximation algorithms for traveling salesman problems* (Cambridge University Press).

Vercesi E, Gualandi S, Mastrolilli M, Gambardella LM (2023) On the generation of metric TSP instances with a large integrality gap by branch-and-cut. *Mathematical Programming Computation* 15(2):389–416.

Vornberger O (1980) Easy and hard cycle covers. Technical report, Universität/Gesamthochschule Paderborn.

Zhong X (2021) Lower Bounds on the Integraliy Ratio of the Subtour LP for the Traveling Salesman Problem. URL `http://dx.doi.org/10.48550/arXiv.2102.04765`, arXiv:2102.04765 [cs, math].

Appendix

A. Simpler Approaches That Do Not Work

The algorithm `ialg` is slow, in part, because it creates a branch for every subset of subtours (or half of them, anyway). Is this really necessary? Would it suffice to consider just one subtour constraint? Perhaps for a smallest subtour? Here, we show that simple heuristics like this can fail miserably. For concreteness, we propose the following approach that we call the `single_SEC_heuristic`().

`single_SEC_heuristic`():

- $\mathcal{S} \leftarrow \{\}$
- while True
    1. compute a solution to DFJ$(G, \mathcal{S})$
    2. if DFJ$(G, \mathcal{S})$ = TSP(G), then return $|\mathcal{S}|$
    3. find the vertex sets $V_1, V_2, \ldots, V_t$ of the DFJ solution over $\mathcal{S}$
    4. $\mathcal{S} \leftarrow \mathcal{S} \cup \{S\}$ where S is one of $V_1, V_2, \ldots, V_t$

PROPOSITION 3. *For some classes of instances, just one SEC is required to prove the optimality of a TSP tour, but the* `single_SEC_heuristic`() *may add* $\Omega(2^n)$ *SECs. For example, this is true for Euclidean instances with* 3 *points located at the* x, y*-coordinate* $(0,0)$ *and* $n-3$ *points at* $(1,0)$.

Proof. For each integer n for which $q := (n-3)/2 \geq 7$ is odd, we construct such an instance. Let the vertex set be $V = V_0 \cup V_1$ where $V_0 = \{1,2,3\}$ are the vertices at $(0,0)$ and $V_1 = \{4,5,\ldots,n\}$ are the vertices at $(1,0)$. An optimal tour is 1-2-3-$\cdots$-n-1 of weight 2, and the single SEC $x(E(S)) \leq |S| - 1$ for $S = V_0$ suffices. Now, consider a cycle C_0 over V_0. For every $S \subseteq V_1$ with $2 < |S| < |V_1|/2 = (n-3)/2 = q$, consider a cycle C_1 over S and a cycle C_2 over $V_1 \setminus S$. Any 2-factor (C_0, C_1, C_2) constructed in this way has zero weight. If `single_SEC_heuristic` encountered them (in any arbitrary sequence), it would add a SEC for each S. The number of them is $(2^{2q} - 2 - 2(2q) - 2\binom{2q}{2})/2 = \Omega(2^{2q}) = \Omega(2^{2(n-3)/2}) = \Omega(2^n)$. □

Next, we consider a variant of the `single_SEC_heuristic` in which the vertex subset S in line 4 is always taken as a smallest vertex set among $V_1, V_2, \ldots, V_t$. We call this variant the `smallest_SEC_heuristic`().

PROPOSITION 4. *For some classes of instances, just one SEC is required to prove optimality of a TSP tour, but the* `smallest_SEC_heuristic`() *may add* $\Omega(2^{2n/3}) = \Omega(1.5874^n)$ *SECs. For example, this is true for Euclidean instances with* q *points located at the* x, y*-coordinate* $(0,0)$ *and* $2q$ *points at* $(1,0)$.

Proof. For each odd $q \geq 7$, we construct such an instance. Let the vertex set be $V = V_0 \cup V_1$ where $V_0 = \{1,2,\ldots,q\}$ are the vertices at $(0,0)$ and $V_1 = \{q+1, q+2, \ldots, 3q\}$ are the vertices at $(0,1)$. An optimal tour is 1-2-3-$\cdots$-n-1 of weight 2, and the single SEC $x(E(S)) \leq |S| - 1$ for $S = V_0$ suffices. Now, consider a cycle C_0 over V_0. For every $S \subseteq V_1$ with $2 < |S| < |V_1|/2 = q$, consider a cycle C_1 over S and a cycle C_2 over $V_1 \setminus S$. Any 2-factor (C_0, C_1, C_2) constructed in this way has zero weight. If `smallest_SEC_heuristic` encountered them (in any arbitrary sequence), it would add a SEC for each S. The number of them is $(2^{2q} - 2 - 2(2q) - 2\binom{2q}{2})/2 = \Omega(2^{2n/3})$. □

B. On a Different Question

What would happen if we posed the question differently? What if we sought the minimum number of SECs that necessarily *produces* a tour? Although this could be an interesting task for future work, one should not ignore the fact that essentially *all* SECs are necessary when the edges have the same cost. Indeed, this observation is what prompted us to phrase the question in terms of proving the optimality of a tour rather than producing it.

REMARK 1. Consider a TSP instance with $n \geq 5$ vertices in which all edges have the same cost. The number of SECs needed to necessarily *produce* a TSP tour is $2^{n-1} - \binom{n}{2} - n - 1 = \Theta(2^n)$, but zero suffice to prove optimality.

Proof. For every $S \subseteq V$ with $2 < |S| < n-2$, we must impose the SEC for either S or its complement $\overline{S} = V \setminus S$, because a cycle over S and a cycle over $\overline{S}$ will satisfy all other SECs and have minimum weight. The number of vertex subsets $S \subseteq V$ that satisfy $2 < |S| < n-2$ is $2^n - 2 - 2n - 2\binom{n}{2}$. Halving this gives the result. □

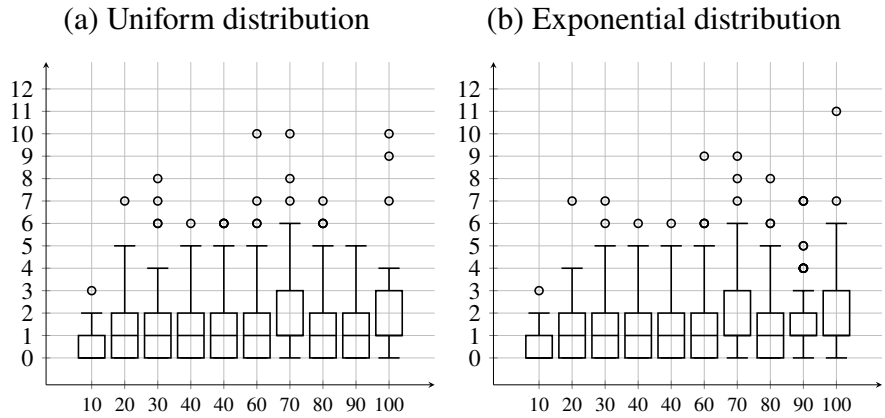
C. Randomly Generated Instances

We consider three classes of randomly generated instances. The first two have edge costs drawn at random, either from a uniform or exponential distribution. The third randomly places points in a square and uses Euclidean distances. For numerical reasons, we round costs to the nearest integer.

The first two classes are motivated by the work of Osorio and Pinto (2003) who claim that costs drawn from the uniform distribution lead to easier instances than from the exponential distribution. The way they measure difficulty is by the number of branch-and-bound nodes needed to prove optimality. The greater such number, the harder the instance. We sampled costs for instances with $n \in \{10, 20, \ldots, 100\}$ vertices. For the uniform distribution, we sample costs between 0 and 1000 and round to the nearest integer. For the exponential distribution, we set $\lambda = \frac{1}{999}$, as suggested by Osorio and Pinto to allow a higher probability of sampling numbers between 0 and 1000, and then round to the nearest integer. For each n, we sample 100 instances.

Results for the first two classes are given in Figure 4. For each each n, we give a boxplot showing the number of SECs required across the 100 samples. We observe that roughly 50% of the instances require 0 or 1 SEC, roughly 75% require at most two SECs, and no cases require more than 11 SECs. All in all, it seems that both of these classes are quite easy.

Figure 4 Results for randomly generated instances whose edge costs are drawn from the uniform and exponential distributions.



Finally, in the third class, we randomly place n points in the unit square and use Euclidean distances. To get a diverse set of edge costs, we multiply the distances by 1000 and round to the nearest integer. We find that computing the minimum number of SECs for these instances is much harder than before, so we sample 100 instances for each $n \in \{10, 15, \ldots, 50\}$.

Figure 5 Results for randomly generated Euclidean instances in the unit square.

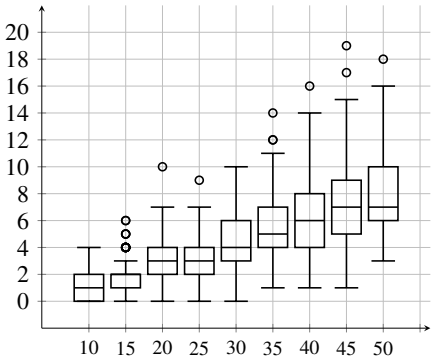
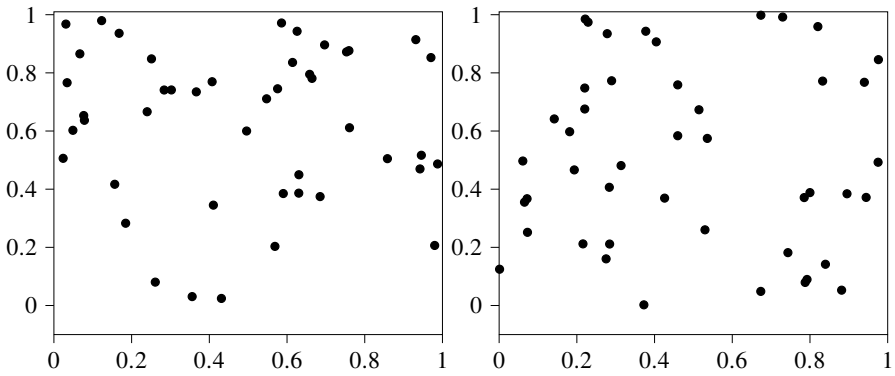


Figure 5 gives results. We observe that the number of SECs grows more quickly than before, already reaching a median of 7 SECs when $n = 45$, whereas only the outliers for the other two classes required this many—even when $n = 100$. We also see some $n = 45$ instances that require up to 19 SECs, while others require only one. Figure 6 shows one instance at each of these extremes.

Figure 6 Two instances sampled from the unit square. The left instance needs just one SEC, while the right needs 19 SECs.

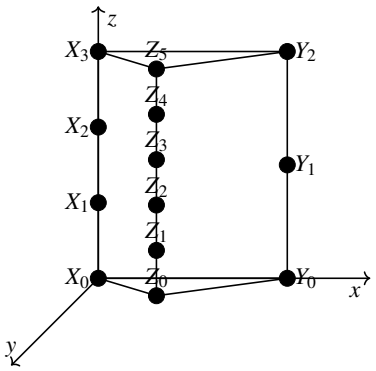


D. Rectilinear 3-Dimensional Instances

Previously, we have seen a class of TSP instances on n vertices that require a linear number $\Theta(n)$ of SECs (see Proposition 2) and several instaces requiring many SECs (see Tables 2 and 3). Here, we consider a class of instances that seem to require a quadratic number $\Theta(n^2)$ of SECs.

Specifically, Zhong (2021) proposes the *Rectilinear 3-Dimensional instances*, positioning n points on 3 parallel lines in $\mathbb{R}^3$, forming a prism in three-dimensional space. Following their notation, the coordinates of the vertices are given by $X_s = \left(0, 0, \frac{s}{i+1}\right),\ s \in \{0, \ldots, i+1\}, Y_s = \left(\frac{1}{i+1} + \frac{1}{j+1}, 0, \frac{s}{j+1}\right), s \in \{0, \ldots, j+1\}$ and $Z_s = \left(\frac{1}{i+1}, \frac{1}{k+1}, \frac{s}{k+1}\right), s \in \{0, \ldots, k+1\}$,

Figure 7 **An example of rectilinear 3-dimensional instance, with $n = 13$ and hence $i = 2, j = 1, k = 4$**



where X_0, Y_0, and Z_0 form the base of the prism. The positions of the n points depend on three parameters: i, j, k, such that $n = i + j + k + 6$, and the rectilinear distance between points is used. Figure 7 shows an example for $n = 13$.

Zhong shows that certain choices of i, j, k make these instances hard for Concorde. Here, we tune these parameters in an attempt to find instances that require $\Theta(n^2)$ SECs. In the case where $n \equiv 1 \pmod 3$, we set the parameters as follows $i = \frac{n-1}{3} - 2,\ j = i - 1,\ k = i + 2$. For convenience, we relabel the vertices into three groups, one for each parallel line:

$$V_1 = \{0, 1, \ldots, i+1\}, \qquad V_2 = \{i+2, \ldots, 2i+2\}, \qquad V_3 = \{2i+3, \ldots, n-1\}.$$

For such instances, we have empirically found that the smallest collection of SECs needed to prove optimality often has the following *chain structure*: For each group of nodes V_j, we have all the chains of length k, for $k \in \{3, \ldots, |V_j|\}$. Empirically, this holds for $n \in \{13, 16, 19, 22, 25, 28, 31, 34\}$. For $n \geq 37$, the `ialg` computation is too demanding; this is not surprising given that the Rectilinear 3-dimensional instances are arguably the hardest metric TSP instances in the literature.

Example. For $n = 13$, we have $i = 2$, and the three groups are:

$$V_1 = \{0, 1, 2, 3\} \qquad V_2 = \{4, 5, 6\} \qquad V_3 = \{7, 8, 9, 10, 11, 12\}.$$

In this case, the SECs we add are $x(E(S)) \leq |S| - 1$ for S being:

$$\{0,1,2\}, \{1,2,3\}, \{0,1,2,3\}, \{4,5,6\}, \{7,8,9\}, \{8,9,10\}, \{9,10,11\}, \{10,11,12\},$$
$$\{7,8,9,10\}, \{8,9,10,11\}, \{9,10,11,12\}, \{7,8,9,10,11\}, \{8,9,10,11,12\},$$
$$\{7,8,9,10,11,12\}.$$

Generally, the number of chain SECs can be computed as:

$$\sum_{t=1}^{i} t + \sum_{t=1}^{i-1} t + \sum_{t=1}^{i+2} t = \frac{3i^2 + 5i + 6}{2},$$

which grows quadratically in n. Proving whether these instances truly require a quadratic number of SECs is an interesting question for future work.