Routing on Sparse Graphs with Non-metric Costs for the Prize-collecting Travelling Salesperson Problem

Patrick O'Hara[1,*], M. S. Ramanujan[1] and Theodoros Damoulas[1,2,3]

[1]Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK
[2]Department of Statistics, University of Warwick, Coventry, CV4 7AL, UK
[3]The Alan Turing Institute, British Library, 96 Euston Road, London, NW1 2DB, UK

Abstract

In many real-world routing problems, decision makers must optimise over sparse graphs such as transportation networks with non-metric costs on the edges that do not obey the triangle inequality. Motivated by finding a sufficiently long running route in a city that minimises the air pollution exposure of the runner, we study the Prize-collecting Travelling Salesperson Problem (Pc-TSP) on sparse graphs with non-metric costs. Given an undirected graph with a cost function on the edges and a prize function on the vertices, the goal of Pc-TSP is to find a tour rooted at the origin that minimises the total cost such that the total prize is at least some quota. First, we introduce heuristics designed for sparse graphs with non-metric cost functions where previous work dealt with either a complete graph or a metric cost function. Next, we develop a branch & cut algorithm that employs a new cut we call the disjoint-paths cost-cover (DPCC) cut. Empirical experiments on two datasets show that our heuristics can produce a feasible solution with less cost than a state-of-the-art heuristic from the literature. On datasets with non-metric cost functions, DPCC is found to solve more instances to optimality than the baseline cutting algorithm we compare against.

Keywords

Heuristics, Branch & cut, Routing, Air pollution, Travelling Salesperson Problem

1. Introduction

Variants of the Travelling Salesperson Problem (TSP) and Vehicle Routing Problem frequently appear in real-world applications [1, 2, 3, 4]. The Prize-collecting Travelling Salesperson Problem (Pc-TSP) is a member of the family of TSPs with Profits [5]. Given a graph with a prize function on the vertices and a cost function on the edges, the objective of Pc-TSP is to minimise the total cost of a tour that starts and ends at a given root vertex such that the total prize is at least a given threshold called the quota. Unlike the traditional TSP, the tour is not required to visit all of the vertices. Applications of Pc-TSP naturally arise in a range of industrial and transportation settings. Balas [6] proposes Pc-TSP as a model for the daily operations of a steel rolling mill. Fischetti and Toth [7] minimise the total distance travelled by a vehicle collecting a quota of a product from suppliers such that the tour starts and finishes at a factory. Awerbuch et al. [8] consider a salesperson visiting cities to sell a quota of brushes in order to win a trip to Hawaii. The above applications and their respective algorithms assume either: (i) every pair of vertices are connected by an edge such that the input graph is complete; (ii) the cost function is a metric function that obeys the triangle inequality; (iii) both of the previous assumptions.

However, in many real-world applications, making assumptions that the graph is complete or the cost function is metric is not desirable. Our motivating example is a runner planning a route through the streets of a city (see Figure 1). The runner requests several attributes of the route. Firstly, the runner is conscious about exposure to air pollution on the route: air pollution has an adverse effect on the cardio-respiratory system, which can be exacerbated by increased inhalation during exercise [9]. In urban environments, air pollution is highly localised because factors such as transportation,

ATT'24: Workshop Agents in Traffic and Transportation, October 19, 2024, Santiago de Compostela, Spain
*Corresponding author.
patrick.h.o-hara@warwick.ac.uk (P. O'Hara); r.maadapuzhi-sridharan@warwick.ac.uk (M. S. Ramanujan); t.damoulas@warwick.ac.uk (T. Damoulas)
0000-0001-9600-7554 (P. O'Hara); 0000-0002-2116-6048 (M. S. Ramanujan); 0000-0002-7172-4829 (T. Damoulas)

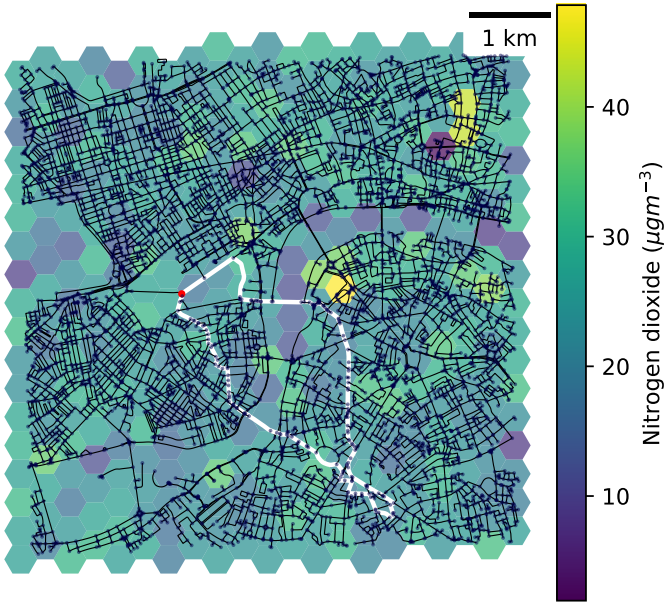


Figure 1: Example of a 10km tour that minimises the air pollution exposure of a runner in London, UK. The route starts and ends at the red vertex. Yellow cells indicate high air pollution. Blue cells indicates low air pollution.

industry and construction largely contribute to poor air quality [10]. Fortunately, recent advances in air quality modelling [11] means live, localised air quality forecasts can be used to predict the air pollution exposure on individual streets (see Section 5). Secondly, the runner requests the route starts and ends at the same location and is sufficiently long. For example, the runner may ask for a route that starts and ends at their home and is at least 5km. Finally, the runner asks that the route is not repetitive: whilst running up and down a single street with good air quality many times is an effective way to minimise air pollution exposure, it is also a very boring route for the runner. We add the constraint that no section of the route is repeated, leaving other constraints on route repetition to future work.

Our motivating example can be formulated as an instance of Pc-TSP. The road network is represented by an undirected graph G where the edges are roads and vertices are intersections. The road network is a sparse graph: the number of edges m is at most κn where κ is a constant and n is the number of vertices. The start and end location of the route is represented by a root vertex. The air pollution exposure on a road can be modelled by a non-metric cost function on the edges. The length of the runner's route is analogous to prize collected on vertices by the salesperson: the prize is generated by splitting each edge (i, j) in the graph into two edges $(i, k), (k, j)$ and assigning the length of edge (i, j) to the prize of a newly created vertex k, whilst the prize of i and j is zero. To avoid route repetition, we ask that the route is a simple cycle. The objective is to minimise the air pollution exposure of a simple cycle starting and ending at the root vertex such that the length of the route is at least the given quota.

From the polyhedral results derived by Balas [6, 12, 13], three papers propose exact algorithms for Pc-TSP [7, 14, 15]. Fischetti and Toth [7] derive lower bounds, however, experimental results on randomly-generated, directed, complete graphs show instances with a *symmetric* cost function (which is equivalent to a cost function over undirected edges) could not be solved on graphs with greater than 40 vertices due to excessive computing time. Both [14] and [15] use cutting planes to strengthen the lower bound via valid inequalities (constraints that do not eliminate any *feasible* integer solutions) and conditional inequalities [16] (constraints that are valid for every *optimal* solution but not for every feasible solution). Given an upper bound on the optimal solution, cost-cover inequalities are a type of conditional inequality for Pc-TSP that constrain vertex variables using the cost of connecting a set of vertices. The cost-cover inequalities of [14] and [15] both assume the cost function is metric. Moreover, the heuristics used in [14] and [15] to obtain the upper bound assumes the graph is complete.

There are some key differences when designing polynomial-time heuristics for Pc-TSP on complete

graphs with metric costs versus sparse graphs with non-metric costs. First, given an instance of Pc-TSP on a complete graph, one can construct a feasible solution [14, 15, 17] in polynomial time; but when the graph is incomplete, we prove in Proposition 2 that no polynomial-time algorithm is guaranteed to recover a feasible solution to every instance of Pc-TSP, unless $\mathcal{P} = \mathcal{NP}$. Second, given a tour $\mathcal{T}$ whose prize is less than the quota, heuristics [18, 19] for complete graphs increase the prize by adding a vertex j between two adjacent vertices u_h, u_{h+1} in $\mathcal{T}$ until the tour has sufficient prize; however, on sparse graphs, one cannot assume edges (u_h, j) and (j, u_{h+1}) always exist. Lastly, given tour $\mathcal{T}$ whose prize is at least the quota, heuristics from the literature decrease the cost whilst maintaining feasibility by applying operations such as deleting a vertex [18], replacing a vertex [20, 17], or re-sequencing the tour [19]; similarly to above, these operators assume the existence of edges that may not exist in a sparse graph. Moreover, [19] explicitly assumes the cost function is metric in their re-sequencing operator. We conclude our literature review by emphasising that constructing a complete graph G' from our sparse input graph G then naively running a heuristic from the literature on G' does not guarantee the solution returned by the heuristic will be feasible for G. Furthermore, such a construction increases the number of edges to $\mathcal{O}(n^2)$, slowing down the running time. We give two constructions in Appendix A of the supplementary material.

Contributions With the motivation of finding routes minimising air pollution exposure in mind, the focus of this paper is to develop algorithms to solve Pc-TSP on sparse, undirected graphs with non-metric cost functions. Our contributions are:

1. Proposing a *three-stage heuristic* called Suurballe's path extension & collapse (SBL-PEC) which (i) generates an initial low-cost tour; (ii) increases the prize of the tour by adding paths with the smallest cost-to-prize ratio; and (iii) decreases the cost of the tour whilst maintaining feasibility by swapping a sub-path of the tour for an alternative path with less cost. Stages (ii) and (iii) contain [19] as a special case.

2. Deriving a new *disjoint-paths cost-cover inequality* and proving it is stronger than a shortest-path cost-cover inequality (Proposition 4). Our disjoint-paths cost-cover inequality is integrated into our branch & cut algorithm for solving instances to optimality.

3. Empirically evaluating our method on real-world instances from the London air quality dataset [11] and on synthetic instances from TSPLIB [21]. We compare our heuristics against [19] and compare our branch & cut algorithm against an adaptation of [14].

2. Problem definition

An instance of Pc-TSP is given by the tuple $\mathcal{I} = (G, c, p, Q, v_1)$. The graph G is undirected with vertices $V(G)$ and edges $E(G)$. The graph is assumed to be connected and simple with no self-loops. The number of vertices and the number of edges are denoted $n = |V(G)|$ and $m = |E(G)|$ respectively. The set of neighbours of vertex i is denoted $N(i) = \{j \in V(G) : (i, j) \in E(G)\}$. The cost function $c : E(G) \to \mathbb{N}_0$ defined on the edges is non-negative where $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. The prize function $p : V(G) \to \mathbb{N}_0$ defined on the vertices is also non-negative. For convenience, we denote the total prize of a set of vertices $S \subseteq V(G)$ as $p(S) = \sum_{i \in S} p(i)$. Similarly, the total cost of a set of edges $F \subseteq E(G)$ is $c(F) = \sum_{(i,j) \in F} c(i, j)$. We are given a non-negative quota $Q \in \mathbb{N}_0$. The root vertex is $v_1 \in V(G)$.

Definition 1. *A tour $\mathcal{T} = (u_1, \ldots, u_k, u_1)$ of an undirected graph G is a sequence of adjacent vertices such that each vertex in the tour is visited exactly once. A vertex u is visited exactly once if the tour enters u exactly once and leaves u exactly once. The set of vertices in the tour is defined by $V(\mathcal{T}) = \{u_1, \ldots, u_k\}$ and the set of edges of the tour is defined by $E(\mathcal{T}) = \{(u_1, u_2), \ldots, (u_{k-1}, u_k), (u_k, u_1)\}$. We say a tour is prize-feasible if $p(V(\mathcal{T})) \geq Q$.*

A *feasible solution* to a given instance $\mathcal{I}$ of Pc-TSP is a tour $\mathcal{T}$ starting and ending at the given root vertex v_1 such that $\mathcal{T}$ is prize-feasible. The objective of Pc-TSP is to minimise the total cost $c(E(\mathcal{T}))$ of

the edges in a tour $\mathcal{T}$ such that the tour is a feasible solution. A tour $\mathcal{T}^\star$ is an *optimal solution* to $\mathcal{I}$ if for all feasible tours $\mathcal{T}$ we have $c(E(\mathcal{T}^\star)) \leq c(E(\mathcal{T}))$.

Definition 2. *A graph is sparse if the number of edges m is at most κn, where κ is a positive constant.*

Definition 3. *Let $\mathcal{P}_{uv}$ be the least-cost path from vertex u to vertex v in G and let $E(\mathcal{P}_{uv})$ be the set of path edges. An edge (u, v) is metric if $\forall w \in V(G)$: (i) $c(u, v) = 0 \Leftrightarrow u = v$, (ii) $c(u, v) = c(v, u)$, (iii) $c(u, v) \leq c(E(\mathcal{P}_{uw})) + c(E(\mathcal{P}_{wv}))$. A cost function is metric if edge (u, v) is metric for $\forall (u, v) \in E(G)$, and the cost function is non-metric otherwise.*

Lemma 1. *Let G be a connected, undirected graph and $c : E(G) \to \mathbb{N}_0$ be a cost function on the edges. The number of metric edges in $E(G)$ is at least $n - 1$.*

Definition 4. *Given graph G and cost function $c : E(G) \to \mathbb{N}_0$, the metric surplus $\zeta(G, c)$ is:*

$$\zeta(G, c) = \frac{\left(\sum_{(u,v)\in E(G)} M(u, v)\right) - (n - 1)}{m - (n - 1)}$$

where $M(u, v) = 1$ if edge (u, v) is metric, or 0 otherwise.

In this paper, we assume the input graph is sparse and we do not assume the triangle inequality (iii) in Definition 3 always holds for the cost function. The metric surplus from Definition 4 gives us a measure of "how metric" a cost function is. Notice that if all edges in G are metric, then $\zeta(G, c) = 1$. But if the number of metric edges is equal to the lower bound $n - 1$ from Lemma 1, then $\zeta(G, c) = 0$. Finally, given an instance $\mathcal{I}$, we apply pre-processing to reduce the size of a sparse input graph G by removing vertices that are not in the same biconnected component as the root vertex v_1 (see Appendix B.2).

3. Heuristics

Heuristics for Pc-TSP provide an upper bound on the optimal solution. A heuristic that runs efficiently in worst-case polynomial-time complexity is desirable. However, on a general graph which is not guaranteed to be complete, the design of polynomial-time heuristics that always return a prize-feasible tour (if one exists) is not possible:

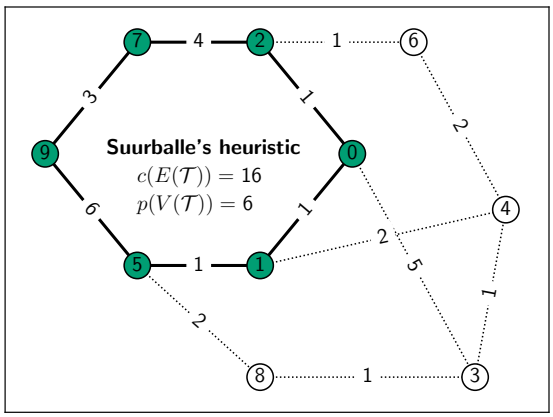
Proposition 2. *Assume $\mathcal{P} \neq \mathcal{NP}$. Let G be any undirected graph. No polynomial-time algorithm A is guaranteed to find a feasible solution (if one exists) for every instance of Pc-TSP.*

Proof. Reduction from Hamiltonian Cycle: see Appendix C. □

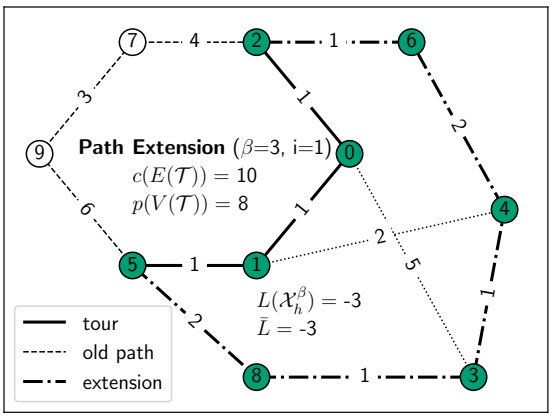
Whilst the polynomial-time heuristics we design in this section are not guaranteed to always find a feasible solution on *all* general input graphs due to Proposition 2, our aim is to design heuristics that find a low-cost feasible solution on *most* sparse, non-metric instances in practice. This section presents a combined heuristic (Section 3.4) comprising of three distinct components: generating a starting solution (Section 3.1); increasing the prize of the tour (Section 3.2); and reducing the cost of the tour (Section 3.3).

3.1. Suurballe's Heuristic

To find an initial low-cost tour, we propose Suurballe's heuristic (SBL). The central idea is to find $n - 1$ tours $\mathcal{T}_1, \ldots, \mathcal{T}_{n-1}$ from the least-cost pair of vertex-disjoint paths from the root vertex v_1 to every other vertex $t \in V(G)\backslash\{v_1\}$. Two simple paths $\mathcal{D}_1 = (u_1, \ldots, u_k)$, $\mathcal{D}_2 = (w_1, \ldots, w_l)$ from $v_1 = u_1 = w_1$ to $t = u_k = w_l$ are *vertex disjoint* if $V(\mathcal{D}_1) \cap V(\mathcal{D}_2) = \{v_1, t\}$. In an undirected graph, $\mathcal{D}_1$ and $\mathcal{D}_2$ form a tour $\mathcal{T} = (u_1, \ldots, u_{k-1}, u_k, w_{l-1}, \ldots, w_1)$ by removing w_l from $\mathcal{D}_2$, reversing the order of $\mathcal{D}_2$, then appending the modified $\mathcal{D}_2$ to the end of $\mathcal{D}_1$.



(a) Suurballe's heuristic generates the initial tour (0,1,5,9,7,2,0) from a pair of vertex-disjoint paths (0,1,5,9) and (0,2,7,9). Vertices in green are in the tour. The total cost of $\mathcal{T}$ is 16 and the total prize is 6. We set $\mathcal{T}^\star \leftarrow \mathcal{T}$.



(b) Path extension of Fig. 2a with $\beta = 3$ finds a tour with cost 10 and prize 8. The only feasible extension on iteration $i = 1$ is (2,6,4,3,8,5). For $i = 2$, path (2,7,9,5) is not considered because its prize is not greater than the internal path (2,0,1,5).

Figure 2: Demonstration of the SBL and path extension stages of PEC-SBL heuristic (Algorithm 1). Vertex 0 is the root. The prize of every vertex is 1. Edges are labelled with their cost. The quota is $Q = 6$.

SBL returns the least-cost prize-feasible tour from $\mathcal{T}_1, \ldots, \mathcal{T}_{n-1}$ if one exists; else if no tours are prize-feasible, it returns the tour that maximises the prize. Suurballe's algorithm[1] [22] for finding the least-cost pair of vertex-disjoint paths from v_1 to every other vertex t takes $\mathcal{O}(m \log n)$. Checking a single tour is prize-feasible takes $\mathcal{O}(n)$ and the heuristic checks $n - 1$ tours for prize-feasibility. The overall running time of our algorithm is $\mathcal{O}(m \log n + n^2)$. On a sparse graph, the worst-case is dominated by $\mathcal{O}(n^2)$. On a dense graph, the worst-case is dominated by $\mathcal{O}(m \log n)$. The weakness of SBL is it fails to find a prize-feasible tour if there does not exist a least-cost pair of vertex-disjoint paths $\mathcal{D}_1, \mathcal{D}_2$ from v_1 to some $t \in V(G)$ such that $p(V(\mathcal{D}_1)) + p(V(\mathcal{D}_2)) - p(t) \geq Q$.

3.2. Path Extension

Given an initial tour $\mathcal{T} = (u_1, \ldots, u_k, u_1)$ with $u_1 = v_1$, we propose a path extension algorithm to increase the total prize of the tour. Increasing the prize has two goals: first, if $\mathcal{T}$ is not prize-feasible, then increase the prize to obtain feasibility; second, if $\mathcal{T}$ is prize-feasible, then explore the solution space in different areas of the graph by increasing the prize. Intuitively, in each iteration i, the algorithm searches for possible *extension paths* between two vertices of the tour, then chooses the extension path with the smallest cost-to-prize ratio. A new tour with larger prize is constructed from the extension path and the existing tour. The algorithm repeatedly increases the prize of the tour with extension paths until a termination criteria is met.

More specifically in each iteration i, for every $h \in \{1, \ldots, k - \beta\}$, path extension searches for an extension path $\mathcal{X}_h^\beta$ from u_h to $u_{h+\beta}$ where $\beta \in \mathbb{N}$ is a step size parameter and $k = |V(\mathcal{T})|$. The extension path $\mathcal{X}_h^\beta$ is not allowed to use vertices in $\mathcal{T}$ except for u_h and $u_{h+\beta}$. To find $\mathcal{X}_h^\beta$, we use a Breadth First Search (BFS) from u_h such that $|V(\mathcal{X}_h^\beta)| \geq 3$. Let us define the *internal path* $\mathcal{P}_h^\beta = (u_h, u_{h+1}, \ldots, u_{h+\beta})$ to be the sub-path of $\mathcal{T}$ that would be replaced by the extension path $\mathcal{X}_h^\beta$. A new tour can be created by replacing the internal path with the extension path. To compare extension paths, we propose a ratio of the cost to prize called the *unitary loss*:

$$L(\mathcal{X}_h^\beta) = \frac{c(E(\mathcal{X}_h^\beta)) - c(E(\mathcal{P}_h^\beta))}{p(V(\mathcal{X}_h^\beta)) - p(V(\mathcal{P}_h^\beta))} \qquad \text{where } p(V(\mathcal{X}_h^\beta)) > p(V(\mathcal{P}_h^\beta)) \tag{1}$$

[1] Suurballe's algorithm uses directed, asymmetric graphs. Transforming undirected to directed, asymmetric takes $\mathcal{O}(n + m)$.

In each iteration i, we construct a set $S_i^\beta = \{\mathcal{X}_h^\beta | p(V(\mathcal{X}_h^\beta)) > p(V(\mathcal{P}_h^\beta)), h \in \{1, \ldots, k-\beta\}\}$ of possible extension paths that have more prize than the internal path. Note we exclude from S_i^β extension paths $\mathcal{X}_h^\beta$ for which $h > k - \beta$ because removing the internal path $\mathcal{P}_h^\beta$ would also remove the root vertex from the tour. We choose the extension path $\mathcal{X}_h^\beta \in S_i^\beta$ that minimises $L(\mathcal{X}_h^\beta)$ and construct a new tour $\mathcal{T}' = (u_1, \ldots, u_{h-1}, \mathcal{X}_h^\beta, u_{h+\beta+1}, \ldots, u_k, u_1)$. We repeat steps for at most n iterations or until termination criteria (a) or (b) is met:

(a) If the prize of the initial tour is less than Q, path extension terminates when the tour becomes prize-feasible, or terminates when S_i^β is empty (the heuristic failed to find a prize-feasible tour).

(b) If the prize of the initial tour is at least Q, path extension terminates when there does not exist a path $\mathcal{X}_h^\beta \in S_i^\beta$ such that $L(\mathcal{X}_h^\beta) < \overline{L}$ where:

$$\overline{L} = \frac{1}{|S_{i=1}^\beta|} \sum_{\mathcal{X}_h^\beta \in S_{i=1}^\beta} L(\mathcal{X}_h^\beta) \tag{2}$$

$\overline{L}$ is calculated on the first iteration $i = 1$, but not re-calculated on subsequent iterations.

Each iteration i takes $\mathcal{O}(n(n+m))$ time, and there are at most n iterations, giving path extension worst-case time complexity $\mathcal{O}(n^2(n+m))$. We found through experimentation that if $\beta > 10$, the tour is rarely extended, thus we limit β to be at most $\beta_{\max} = 10$. We limit the maximum number of iterations to be n because the worst-case running time would otherwise be upper bounded by the number of times one can increase the prize of the tour by replacing a sub-path with an extension path. Depending on the prize function and topology of the graph, this upper bound could be much larger than n. Since we seek a fast heuristic (and since path extension will be called $\beta_{\max}$ times, see Section 3.4), we limit the maximum number of iterations to be n.

The path extension algorithm addresses limitations of previous work in the following ways. First, on sparse graphs, path extension can increase the prize of the tour by adding paths between non-adjacent vertices in the tour: it is not limited by adding a single vertex between adjacent vertices of the tour. For example, in Figure 2b, path extension recovers the optimal solution by finding an extension path with five edges that replaces an internal path with three edges. Second, the unitary loss function (1) does not assume the cost function is metric. Indeed if an edge $(u_h, u_{h+1}) \in E(\mathcal{T})$ is non-metric such that the cost of (u_h, u_{h+1}) is greater than the cost of the extension path $\mathcal{X}_h^1$, then $L(\mathcal{X}_h^1) < 0$ so path extension is likely to choose $\mathcal{X}_h^1$ as the extension path. In this case, extending the tour with $\mathcal{X}_h^1$ decreases the cost and increases the prize of the tour, which is a desirable greedy move for the algorithm to make.

3.3. Path Collapse

To reduce the cost of a given prize-feasible tour $\mathcal{T} = (u_1, \ldots, u_k, u_1)$, we propose the path collapse heuristic. The intuition is to swap a sub-path of the tour for an alternative path with less cost whilst maintaining a prize-feasible tour containing the root vertex. We iterate over k sub-paths of the tour (where $k = |V(\mathcal{T})|$) and choose the alternative path that produces the least-cost prize-feasible tour. More specifically, path collapse iterates over each $i \in \{1, \ldots, k\}$:

1. Start from vertex u_i, then take a sub-path $P = (u_i, \ldots, u_j)$ of $\mathcal{T}$ where $v_1 \in V(P)$ such that $p(V(P)) < Q$ and $p(V(P)) + p(u_{j+1}) \geq Q$.

2. For all neighbours $s_l \in N(u_i)$ of u_i, search for the least-cost path $\mathcal{S}_l = (s_1, \ldots, s_l)$ from $s_1 = u_j$ to s_l using vertices of G that are not in P.

3. From $s_l \in N(u_i)$, choose the path $\mathcal{S}_l^\star$ that minimises $c(E(\mathcal{S}_l)) + c(s_l, u_j)$ such that $p(V(P)) + p(V(\mathcal{S}_l)) - p(u_j) \geq Q$. If $c(E(P)) + c(E(\mathcal{S}_l^\star)) + c(s_l, u_j) < c(E(\mathcal{T}))$, then form a new collapsed tour $\mathcal{T}_i' = (u_i, \ldots, u_j, s_2, \ldots, s_l, u_i)$ by appending $\mathcal{S}_l^\star$ and (s_l, u_i) to P.

From the k collapsed tours $\mathcal{T}_i'$ returned by step 3 in each iteration $i \in \{1, \ldots, k\}$, path collapse returns the least-cost tour. Using Dijkstra's algorithm [23] for shortest paths, the running-time of path collapse is $\mathcal{O}(k \cdot m \log n)$. Note that our path collapse algorithm contains [19] as a special case when the shortest path from u_j to u_i has exactly two edges. path collapse is a generalization that allows shortest paths with more than two edges. This generalization is critical in a sparse graph when a two-edge path from u_j to u_i may not exist. If edge (u_j, s_l) exists and is non-metric, then the generalization is important because there likely exists a least-cost path $\mathcal{S}_l$ from u_j to $s_l \in N(u_j)$ that does not visit edge (u_j, s_l) such that the resulting tour $\mathcal{T}_i'$ has less cost and more prize (since $p(u_j) + p(s_l) < p(\mathcal{S}_l)$).

3.4. Suurballe's Path Extension & Collapse (SBL-PEC)

Algorithm 1: Path extension & collapse initialised with Suurballe's heuristic (SBL-PEC).

Input: Instance $\mathcal{I} = (G, c, p, Q, v_1)$; parameter $\beta_{\max} \in \mathbb{N}$. **Output:** Tour $\mathcal{T}^\star$.

1 Find a tour $\mathcal{T}$ using SBL ;
2 **if** $p(V(\mathcal{T})) < Q$ **then foreach** $\beta \in \{1, \ldots, \beta_{\max}\}$ **do** path extension with termination **(a)**;
3 $\mathcal{T}^\star \leftarrow$ path collapse $\mathcal{T}$;
4 **for** $\beta \in \{1, \ldots, \beta_{\max}\}$ **do**
5 $\quad \mathcal{T} \leftarrow$ path extension on $\mathcal{T}^\star$ with step size β with termination criteria **(b)**;
6 $\quad \mathcal{T} \leftarrow$ path collapse $\mathcal{T}$;
7 $\quad$ **if** $c(E(\mathcal{T})) < c(E(\mathcal{T}^\star))$ **then** $\mathcal{T}^\star \leftarrow \mathcal{T}$;

To summarise SBL-PEC (Algorithm 1), we generate an initial tour $\mathcal{T}$ with SBL; if $\mathcal{T}$ is not prize-feasible, we repeat path extension for each β until $p(V(\mathcal{T})) \geq Q$; then for each β, we alternate path extension with path collapse. SBL-PEC has a worst-case time complexity equal to the number of times we call path extension, which is $\mathcal{O}(\beta_{\max} n^2 (n + m))$.

4. Branch & Cut Algorithm

In this section, we develop a branch & cut algorithm to find the optimal solution of sparse, non-metric instances of Pc-TSP. Our primary contribution is a new conditional cost-cover inequality based on vertex-disjoint paths (Section 4.2). For a computational study on the effectiveness of other valid inequalities for Pc-TSP such as cover and comb inequalities, we refer the reader to [14]. Our branch & cut algorithm is implemented using SCIP v8.0.3 [24, 25, 26] and linear programs are solved using CPLEX v22.1.1 [27].

4.1. Integer Programming Formulation

We formulate Pc-TSP as an integer linear program (ILP) as follows [14]. Let variable $y_i = 1$ if vertex i is included in the tour, or zero otherwise. Similarly, let $x_{ij} = 1$ if edge (i, j) is included in the tour, or zero otherwise. We can summarise the variables as binary vectors $\boldsymbol{x} \in \{0, 1\}^m$ and $\boldsymbol{y} \in \{0, 1\}^n$. Similarly the cost and prize can be represented by vectors $\boldsymbol{c} \in \mathbb{N}_0^m$ and $\boldsymbol{p} \in \mathbb{N}_0^n$ respectively. Our ILP is

$$\min \quad \boldsymbol{c}^T \boldsymbol{x} \tag{3}$$

$$\text{s.t.} \quad \boldsymbol{p}^T \boldsymbol{y} \geq Q \tag{4}$$

$$y_1 = 1 \tag{5}$$

$$\sum_{j \in N(i)} x_{ij} = 2y_i \qquad \forall i \in V(G) \tag{6}$$

$$\sum_{(j,k) \in E(S)} x_{jk} \leq \sum_{j \in S} y_j - y_i \qquad \forall S \subset V(G),\ v_1 \in V(G) \backslash S,\ v_i \in S \tag{7}$$

$$\boldsymbol{x} \in \{0, 1\}^m,\ \boldsymbol{y} \in \{0, 1\}^n. \tag{8}$$

Constraint (4) enforces the total prize of the tour to be at least the quota. Constraint (5) requires the root vertex to be in every feasible solution. Constraint (6) ensures every vertex in the graph is visited at most once by the tour. Constraints (7) are called the sub-tour elimination constraints (SECs) which ensure the tour is connected. Constraints (8) requires the edge and vertex variables to be binary. The set of feasible solutions is given by $\mathcal{F} = \{(\boldsymbol{x}, \boldsymbol{y}) \in \{0,1\}^m \times \{0,1\}^n | (4) - (8)\}$. A solution $(\boldsymbol{x}^\star, \boldsymbol{y}^\star) \in \mathcal{F}$ is optimal if for all $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{F}$ we have $\boldsymbol{c}^T\boldsymbol{x}^\star \leq \boldsymbol{c}^T\boldsymbol{x}$. A relaxation is obtained by removing some of the constraints from the integer program and can be used to obtain a lower bound (LB) on the cost of the optimal solution. Given an upper bound (UB) on the optimal solution, we define the gap between the bounds as GAP = (UB - LB) / LB. A solution is optimal when the GAP is zero.

The linear program at the root node of the branch & bound tree is defined by the objective function (3) with constraints (4)-(6). The integrality constraints on $\boldsymbol{x}$ and $\boldsymbol{y}$ are relaxed to $\forall (i,j) \in E(G) : 0 \leq x_{ij} \leq 1$ and $\forall i \in V(G) : 0 \leq y_i \leq 1$. Sub-tour elimination constraints are only added to a linear program when a separation algorithm finds a violated inequality (Appendix E.1.1). The initial upper bound C_U is set by running the SBL-PEC heuristic from Section 3. If the value of the objective function of a solved linear program is greater than the current upper bound, the node belonging to the linear program is pruned from the branch & bound tree.

4.2. Cost-cover Inequalities

Given an upper bound C_U on the optimal solution of an instance of Pc-TSP and a subset S of vertices, a cost-cover inequality constrains the number of vertices from S that can be in the optimal solution. Let $l : S \to \mathbb{N}_0$ be a function that returns a lower bound on the cost of connecting vertices in S with any edges from $E(G)$. Note that C_U acts as the capacity in a knapsack constraint: $\sum_{(i,j)\in E(G)} x_{ij}c(i,j) \leq C_U$. Under the *condition* $l(S) > C_U$, S is a cover of the knapsack constraint, and no optimal solution contains every vertex from S. The following conditional inequality is then valid for the optimal solution (but not for every feasible solution):

$$\sum_{i \in S} y_i \leq |S| - 1 \qquad \forall S \subset V(G), \quad l(S) > C_U \tag{9}$$

Consider the case when S contains two vertices i and j. On a non-metric and sparse graph, if twice the cost of the shortest path $\mathcal{P}_{ij}$ between i and j is greater than C_U, then at most one of the endpoints i, j can be included in the optimal solution. Formally, the *shortest-path cost-cover* (SPCC) inequality is

$$y_i + y_j \leq 1 \qquad \forall i,j \in V(G), \quad 2c(E(\mathcal{P}_{ij})) > C_U. \tag{10}$$

We propose using the least-cost pair of vertex-disjoint paths between two vertices i, j instead of two times the cost of the shortest path. We first prove the cost of the disjoint paths are a lower bound on the shortest tour containing i and j:

Lemma 3. *In an undirected graph, the minimum cost of a tour $\mathcal{T}$ containing i and j is equal to the least-cost pair of vertex-disjoint paths $\mathcal{D}_1, \mathcal{D}_2$ between i and j.*

Proof. See Appendix D of the supplementary material. □

By Lemma 3, $l(\{i,j\}) = c(E(\mathcal{D}_1)) + c(E(\mathcal{D}_2))$ defines a lower bound on a tour containing i and j. We define the *disjoint-paths cost-cover* (DPCC) inequality as:

$$y_i + y_j \leq 1 \qquad \forall i,j \in V(G), \quad c(E(\mathcal{D}_1)) + c(E(\mathcal{D}_2)) > C_U. \tag{11}$$

Proposition 4. *Let C_u be an upper bound on the optimal solution to an instance $\mathcal{I}$. Let $\mathcal{F}^{(10)}$ and $\mathcal{F}^{(11)}$ be the set of feasible solutions given the SPCC (10) and DPCC (11) inequalities respectively, that is,*

$$\mathcal{F}^{(10)} := \{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{F} \mid (10)\}, \qquad \mathcal{F}^{(11)} := \{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{F} \mid (11)\}.$$

Then we have $\mathcal{F}^{(11)} \subseteq \mathcal{F}^{(10)}$.

Proof. To show $\mathcal{F}^{(11)} \subseteq \mathcal{F}^{(10)}$, we prove that if $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{F}^{(11)}$ then $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{F}^{(10)}$. Suppose otherwise. Then $\exists (\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{F}^{(11)}$ such that $(\boldsymbol{x}, \boldsymbol{y}) \notin \mathcal{F}^{(10)}$. This only happens if $\exists i, j \in V(G)$ such that $y_i + y_j \leq 1$ is a constraint in $\mathcal{F}^{(10)}$ but is not a constraint in $\mathcal{F}^{(11)}$. This means $2c(E(\mathcal{P}_{ij})) > C_U$ and $c(E(\mathcal{D}_1)) + c(E(\mathcal{D}_2)) \leq C_u$. Without loss of generality, let $c(E(\mathcal{D}_1)) \leq c(E(\mathcal{D}_2))$. Then $c(E(\mathcal{D}_1)) \leq \frac{1}{2} C_u$ which implies $\mathcal{D}_1$ is a shorter path from i to j than $\mathcal{P}_{ij}$, a contradiction to (10) that $\mathcal{P}_{ij}$ is the shortest path. □

Our branch & cut algorithm checks for violated cost-cover inequalities between the root vertex v_1 to every other $v_i \in V(G)$. Before running the branch & cut algorithm, we find the least-cost pair of vertex-disjoint paths from v_1 to every $v_i \in V(G)$ and store the costs in an array A with n elements. This pre-processing step takes $\mathcal{O}(m \log n)$ for disjoint paths [22]. Note that we decided against finding the least-cost pair of vertex-disjoint paths between every pair of vertices because it would take $\mathcal{O}(nm \log n)$ time complexity and $\mathcal{O}(n^2)$ space complexity. During the branch & cut algorithm, whenever a new upper bound C_U is discovered, if $A[i] > C_U$ then we set $y_i = 0$ due to (11). The time complexity of the separation algorithm (discarding the time for pre-processing) is $\mathcal{O}(n)$ since we must check every element of the array A. The cost-cover separation algorithms for shortest-path cost-cover inequalities (10) and disjoint-paths cost-cover inequalities (11) are the same, except we store the shortest path from v_1 to $v_i \in V(G)$ in array A using Dijkstra's algorithm [23].

5. Computational experiments

We evaluate our heuristics and cost-cover inequalities across multiple real-world and synthetic instances. For each instance and each algorithm, we measure the computational time in seconds (TIME) and the GAP between the upper and lower bounds. TIME is limited to four hours for each instance. For a given algorithm evaluated on a group of instances, we denote the mean GAP as $\overline{\text{GAP}}$, the mean TIME as $\overline{\text{TIME}}$, the number of feasible solutions as FEAS, and the number of optimal solutions as OPT. Our algorithms are implemented in C++ and Python[2]. The machine we use for experiments is a 2×10-core Xeon E5-2660 v3 at 2.6 GHz with 64GB RAM running Linux. Each run of an algorithm is allocated one core and 12GB of RAM using the slurm scheduler [28].

London Air Quality (LAQ) We generate instances of the LAQ dataset by mapping air pollution forecasts from a machine learning model onto the London road network. The air quality model of London [11] is a non-stationary mixture of Gaussian Processes that predicts air pollution (nitrogen dioxide) from data such as sensors, road traffic and weather. The model output consists of forecasts for each cell of a hexagonal grid which overlays the road network (see Figure 1). Every road has a length (in meters). The cost of running along a road is the mean pollution of the grid cells intersecting the road multiplied by the length of the road. The goal is to minimise air pollution exposure of a tour such that the total length of the tour is at least the quota and the tour starts and ends at the root vertex.

The LAQ dataset consists of four graphs[3] named laqbb, laqid, laqkx and laqws which are each created by only keeping vertices within 3000m of the root vertex. The root vertex of each graph is given respectively by the following four locations in London: Big Ben (bb), Isle of Dogs (id), King's Cross (kx), and Wembley Stadium (ws). The prize function of vertices is generated by splitting each edge (i, j) in the graph into two edges $(i, k), (k, j)$ and assigning the length of edge (i, j) to the prize of a newly created vertex k. The prize of i and j is zero. The cost of (i, k) is the same as (i, j) but the cost of (k, j) is set to zero. The metric surplus $\zeta(G, c)$ (see Definition 4) of instances in the LAQ dataset is approximately 0.9. For each of the four graphs, we run our algorithms on five different quotas (1000m, 2000m, 3000m, 4000m, or 5000m), giving us 20 different instances for the London air quality dataset.

[2]Code available at https://github.com/PatrickOHara/pctsp. Datasets available upon request.

[3]We constructed smaller graphs because running our algorithms on the entire London road network (which has hundreds of thousands of vertices and edges) was too computationally expensive.

TSPLIB A well-known benchmark dataset used in multiple papers [29, 14, 16], TSPLIB [21] is a collection of complete graphs. We chose nine graphs with a range of sizes from the original TSPLIB dataset: eil51[4], st70, rat195, tsp225, a280, pr439, rat575, gr666, pr1002. We construct sparse graphs with a metric surplus of zero (meaning every edge is non-metric). Each vertex is labelled from $1, \ldots, n$ and is assigned an x and y co-ordinate. The root vertex is labelled $v_1 = 1$. Let the Euclidean distance between the co-ordinates of vertices i and j be $||i - j||_2$. The prize $p(i)$ of vertex i is defined by three different generations [14, 16]: (i) $p(i) = 1$; (ii) $p(i) = 1+(7141i+73) \mod 100$; and (iii) $p(i) = 1+\lfloor \frac{99}{\theta} ||v_1 - i||_2 \rfloor$ where $\theta = \max_{j \in V(G)} ||v_1 - j||_2$. We set $Q = \alpha \cdot p(V(G))$ where $\alpha \in \{0.05, 0.10, 0.25, 0.50, 0.75\}$. To make the graphs sparse, we remove edges from TSPLIB instances with independent and uniform probability until the number of edges is equal to κn where $\kappa \in \{5, 10, 15, 20, 25\}$.

The cost function is called MST with $\zeta(G, c) = 0$ and is defined as follows. First, assign all edges cost $||i - j||_2$. Next, find the minimum spanning tree T of the sparse graph G. Now, find the shortest path $\mathcal{P}_{ij}^T$ from vertex i to j using only edges in $E(T)$. Finally, assign costs such that edges in T are metric and edges not in T are non-metric:

$$c(i,j) = \begin{cases} \lceil ||i-j||_2 \rceil & \forall (i,j) \in E(T) \\ \lceil ||i-j||_2 \rceil + c(E(\mathcal{P}_{ij}^T)) & \forall (i,j) \in E(G) \backslash E(T) \end{cases}$$

In summary, there are 675 instances in the modified TSPLIB dataset: five levels of sparsity, three prize generating functions, nine distinct graphs, and five values of α.

5.1. Empirical Heuristic Results

In this section, we compare SBL-PEC against a baseline (BFS-EC) from the literature. The BFS-EC baseline initialises the Extension & Collapse (EC) [19] with the first cycle detected by a Breadth First Search (BFS). We also run Suurballe's heuristic (SBL) as a stand alone heuristic. When calculating the GAP, the upper bound (UB) is the total cost of the heuristic solution. The lower bound (LB) is the cost of the largest lower bound found by running our branch & cut algorithm in Section 4 with disjoint-paths cost-cover (DPCC) cuts for a maximum of four hours. On instances where the optimal solution is found within four hours, LB will be the cost of the optimal solution (denoted with a * in Table 1 and Table 2). For a given instance, each heuristic is compared against the same LB when calculating the GAP. From Table 1 and Table 2, we conclude the following:

- *SBL finds low-cost initial tours.* On the LAQ dataset, the $\overline{\text{GAP}}$ is equal for SBL and SBL-PEC on every instance, implying PEC does not improve the SBL tour because SBL finds a solution that is optimal or close-to-optimal, so little improvement is possible. On TSPLIB, PEC is needed to make the SBL tour prize-feasible.

- *BFS-EC [19] cannot sufficiently increase the prize.* On both LAQ and TSPLIB datasets, BFS-EC finds less feasible tours than SBL-PEC. On TSPLIB when G is sparse ($\kappa = 5$), BFS-EC finds 8/135 feasible tours compared to 123/135 for SBL-PEC.

- *SBL-PEC is the best heuristic on TSPLIB.* In addition to finding the most feasible tours, when $\kappa \geq 10$, the $\overline{\text{GAP}}$ of BFS-EC is consistently four times more than the $\overline{\text{GAP}}$ of SBL-PEC.

To conclude our heuristic analysis, SBL is effective as a standalone heuristic on the LAQ dataset, whilst SBL-PEC outperforms all other algorithms on the TSPLIB dataset across varying levels of sparsity.

[4]Code "eil51" means there are 51 vertices in the original TSPLIB instance.

Table 1
Heuristic comparison on all 4 LAQ instances. "nan" means "not a number" where we could not calculate the $\overline{\text{GAP}}$ due to the heuristic finding zero feasible solutions to the four instances. $\overline{\text{GAP}}$ is marked by * if LB is the optimal solution to all four instances found by branch & cut; else $\overline{\text{GAP}}$ is unmarked such that LB is the largest lower bound and the $\overline{\text{GAP}}$ is an overestimate.

	BFS-EC		SBL		SBL-PEC	
Quota	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS
1000	$0.066^\star$	3	$0.000^\star$	4	$0.000^\star$	4
2000	$0.000^\star$	1	$0.040^\star$	4	$0.040^\star$	4
3000	nan	0	$0.027^\star$	4	$0.027^\star$	4
4000	nan	0	$0.049^\star$	4	$0.049^\star$	4
5000	nan	0	0.117	4	0.117	4

Table 2
Heuristic comparison for different sparsity levels κ in the TSPLIB dataset with the MST cost function. Each row corresponds to 135 instances. $\overline{\text{GAP}}$ is calculated using the largest lower bound (LB), and so the $\overline{\text{GAP}}$ is an overestimate for each entry.

	BFS-EC		SBL		SBL-PEC	
κ	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS
5	2.555	8	0.097	46	1.075	123
10	5.430	71	0.124	48	1.333	135
15	8.217	122	0.153	51	1.421	135
20	6.619	120	0.183	51	1.469	135
25	8.125	126	0.239	54	1.466	135

5.2. Cost-cover Inequalities

To evaluate our disjoint-paths cost-cover (DPCC) cuts, we run our branch & cut algorithm from Section 4 with DPCC against shortest-path cost-cover (SPCC) cuts. To find an upper bound C_U, we run SBL-PEC. In addition to TIME and GAP, we record PRE-CUTS: the number of cost-cover cuts added to the first linear program of the root node of the branch & cut tree before the solver is started due to the initial upper bound C_U. We found PRE-CUTS to be a more informative metric than total number of cost-cover cuts because the latter can be affected by factors such as the number of times an upper bound is found during the solving process. From Table 3 and Table 4, we conclude:

- *DPCC adds more $\overline{\text{PRE-CUTS}}$ than SPCC.* Across all quotas on LAQ dataset, DPCC adds at least 300 more $\overline{\text{PRE-CUTS}}$ than SPCC. On TSPLIB for $\alpha = 0.05$ and $\alpha = 0.10$ respectively, DPCC adds 61 and 23 more $\overline{\text{PRE-CUTS}}$ than SPCC, resulting in DPCC finding 5 more and 3 more optimal solutions than SPCC respectively. This is empirical evidence of Proposition 4.

- *On LAQ, DPCC finds the optimal solution three times faster than SPCC when $Q \leq 4000$.* Furthermore, the $\overline{\text{GAP}}$ of DPCC is three times less than SPCC when $Q = 5000$. Both DPCC and SPCC find 17 out of 20 optimal solutions.

- *Few $\overline{\text{PRE-CUTS}}$ are added when $\alpha = 0.25, 0.50, 0.75$.* The result is less than 90 second $\overline{\text{TIME}}$ difference between DPCC and SPCC for $\alpha = 0.25, 0.50, 0.75$. Few $\overline{\text{PRE-CUTS}}$ are added because larger α means collecting more prize, which will generally incur a greater cost, thus increasing the initial upper bound $C_U^\star$ at the root of the branch & bound tree. For all least-cost vertex-disjoint paths $\mathcal{D}_1, \mathcal{D}_2$ from v_1 to $t \in V(G)$, if $c(E(\mathcal{D}_1)) + c(E(\mathcal{D}_2)) < C_U^\star$, then zero DPCC cuts will be added because the condition in (11) is never satisfied. A similar argument can be made for SPCC.

Table 3
Evaluation of our branch & cut algorithm using disjoint-paths vs shortest-path cost-cover inequalities on the London air quality dataset for five different quotas. There are four instances for each quota.

	Disjoint-paths cost-cover cuts				Shortest-path cost-cover cuts			
Quota	$\overline{\text{PRE-CUTS}}$	$\overline{\text{TIME}}$ (s)	$\overline{\text{GAP}}$	OPT	$\overline{\text{PRE-CUTS}}$	$\overline{\text{TIME}}$ (s)	$\overline{\text{GAP}}$	OPT
1000	8260	26.8	0.000	4	7952	70.9	0.000	4
2000	8207	28.6	0.000	4	7840	104.3	0.000	4
3000	7968	95.9	0.000	4	7401	3157.8	0.000	4
4000	7406	1413.0	0.000	4	6788	3976.2	0.000	4
5000	6654	10 870.6	0.042	1	5967	10 981.1	0.151	1

Table 4
Evaluation of our branch & cut algorithm using disjoint-paths vs shortest-path cost-cover inequalities on the TSPLIB dataset for five different values of α. Each row corresponds to 135 instances.

	Disjoint-paths cost-cover cuts				Shortest-path cost-cover cuts			
α	$\overline{\text{PRE-CUTS}}$	$\overline{\text{TIME}}$ (s)	$\overline{\text{GAP}}$	OPT	$\overline{\text{PRE-CUTS}}$	$\overline{\text{TIME}}$ (s)	$\overline{\text{GAP}}$	OPT
0.05	192	1922.4	0.046	120	131	2513.6	0.062	115
0.10	59	4993.9	0.349	92	37	5216.5	0.359	89
0.25	2	9031.0	0.769	54	1	8990.1	0.738	56
0.50	0	9272.2	0.827	51	0	9363.8	0.862	50
0.75	0	8531.1	0.669	61	0	8457.2	0.652	61

6. Conclusion

To summarise, we have designed algorithms for the Prize-collecting Travelling Salesperson Problem on sparse graphs with non-metric cost functions. Suurballe's heuristic found optimal or close-to-optimal solutions on the London Air Quality dataset and proved to be a good starting solution for PEC on the TSPLIB dataset. Our combined SBL-PEC heuristic found a solution with less cost than Extension & Collapse [19] on every instance across both datasets. We derived a disjoint-paths cost-cover (DPCC) cut that tightens the lower bound, then proved the size of the set of solutions defined by the DPCC inequality is at most the size of the set of solutions defined by the SPCC inequality. On the London Air Quality dataset, our DPCC branch & cut algorithm finds optimal solutions three times faster than SPCC for quotas 1km to 4km, whilst the $\overline{\text{GAP}}$ is three times less when the quota is 5km. On the TSPLIB dataset with non-metric costs, DPCC solves more instances to optimality than SPCC.

Our methods can be applied to other family members of TSPs with Profits [5, 16] and other related routing problems [30, 31]. One example is the Orienteering Problem (OP): maximise the total prize of a tour starting and ending at the root vertex such that the total cost is at most an upper bound $Z \in \mathbb{N}$. If the least cost of two vertex-disjoint paths between vertices i, j is greater than Z, then i and j cannot both be in a feasible solution to OP. Therefore, the DPCC inequality for Pc-TSP can be re-written as a valid inequality for OP which is stronger than the SPCC inequality (by Proposition 4).

Future work on the sparse, non-metric Prize-collecting Travelling Salesperson Problem could exploit the rich problem structure offered by our motivating example of running routes that minimise air pollution: dynamic algorithms that optimise a route as air pollution changes over time; multi-objective algorithms that optimise over multiple pollutants; and stochastic optimisation algorithms that consider the uncertainty in predictions from the air quality model.

Acknowledgments

PO and TD acknowledge support from a UKRI Turing AI Acceleration Fellowship [EP/V02678X/1] and a Turing Impact Award from the Alan Turing Institute. PO also acknowledges support from the

Department of Computer Science Studentship of the University of Warwick. MSR acknowledges support from UKRI EPSRC Research Grant [EP/V044621/1]. Batch Computing facilities were provided by the Department of Computer Science of the University of Warwick. The authors would like to thank Oliver Hamelijnck for providing air quality forecasts of London for the experiments. For the purpose of open access, the authors have applied a Creative Commons Attribution (CC-BY) license to any Author Accepted Manuscript version arising from this submission

References

[1] A. Hottung, K. Tierney, Neural large neighborhood search for the capacitated vehicle routing problem, in: ECAI 2020, IOS Press, 2020, pp. 443–450.

[2] B. Nebel, J. Renz, A fixed-parameter tractable algorithm for spatio-temporal calendar management, in: Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI'09, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009, p. 879–884.

[3] X. Pan, Y. Jin, Y. Ding, M. Feng, L. Zhao, L. Song, J. Bian, H-tsp: Hierarchically solving the large-scale traveling salesman problem, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 37, 2023, pp. 9345–9353.

[4] Z. Zhang, H. He, Z. Luo, H. Qin, S. Guo, An efficient forest-based tabu search algorithm for the split-delivery vehicle routing problem, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 29, 2015.

[5] D. Feillet, P. Dejax, M. Gendreau, Traveling salesman problems with profits, Transportation Science 39 (2005) 188–205.

[6] E. Balas, The prize collecting traveling salesman problem, Networks 19 (1989) 621–636.

[7] M. Fischetti, P. Toth, An additive approach for the optimal solution of the prize collecting traveling salesman problem, Vehicle routing: Methods and studies 231 (1988) 319–343.

[8] B. Awerbuch, Y. Azar, A. Blum, S. Vempala, Improved approximation guarantees for minimum-weight k-trees and prize-collecting salesmen, in: Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing, STOC '95, ACM, New York, NY, USA, 1995, pp. 277–283.

[9] L. V. Giles, M. S. Koehle, The health effects of exercising in air pollution, Sports Medicine 44 (2014) 223–249.

[10] S. Vardoulakis, P. Kassomenos, Sources and factors affecting pm10 levels in two european cities: Implications for local air quality management, Atmospheric Environment 42 (2008) 3949 – 3963. Fifth International Conference on Urban Air Quality.

[11] O. Hamelijnck, T. Damoulas, K. Wang, M. Girolami, Multi-resolution multi-task gaussian processes, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. dÁlché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 32, 2019, pp. 14025–14035.

[12] E. Balas, The prize collecting traveling salesman problem: II. Polyhedral results, Networks 25 (1995) 199–216.

[13] E. Balas, The traveling salesman problem and its variations, volume 1, Springer US, Boston, MA, 2007, pp. 663–695.

[14] J.-F. Bérubé, M. Gendreau, J.-Y. Potvin, A branch-and-cut algorithm for the undirected prize collecting traveling salesman problem, Networks 54 (2009) 56–67.

[15] G. Pantuza, M. C. de Souza, Formulations and a lagrangian relaxation approach for the prize collecting traveling salesman problem, International Transactions in Operational Research 29 (2022) 729–759.

[16] M. Fischetti, J. J. S. González, P. Toth, Solving the orienteering problem through branch-and-cut, INFORMS Journal on Computing 10 (1998) 133–148.

[17] O. Pedro, R. Saldanha, R. Camargo, A tabu search approach for the prize collecting traveling salesman problem, Electronic Notes in Discrete Mathematics 41 (2013) 261–268.

[18] A. A. Chaves, L. A. N. Lorena, Hybrid metaheuristic for the prize collecting travelling salesman

problem, in: European Conference on Evolutionary Computation in Combinatorial Optimization, Springer, 2008, pp. 123–134.
[19] M. Dell'Amico, F. Maffioli, A. Sciomachen, A lagrangian heuristic for the prize collectingtravelling salesman problem, Annals of Operations Research 81 (1998) 289–306.
[20] M. Gendreau, A. Hertz, G. Laporte, New insertion and postoptimization procedures for the traveling salesman problem, Operations Research 40 (1992) 1086–1094.
[21] G. Reinelt, TSPLIB—A Traveling Salesman Problem Library, INFORMS Journal on Computing 3 (1991) 376–384.
[22] J. W. Suurballe, R. Tarjan, A quick method for finding shortest pairs of disjoint paths, Networks 14 (1984) 325–336.
[23] E. W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.
[24] T. Achterberg, T. Berthold, T. Koch, K. Wolter, Constraint integer programming: A new approach to integrate cp and mip, in: L. Perron, M. A. Trick (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 6–20.
[25] T. Achterberg, Scip: solving constraint integer programs, Mathematical Programming Computation 1 (2009) 1–41.
[26] K. Bestuzheva, M. Besançon, W.-K. Chen, A. Chmiela, T. Donkiewicz, J. van Doornmalen, L. Eifler, O. Gaul, G. Gamrath, A. Gleixner, L. Gottwald, C. Graczyk, K. Halbig, A. Hoen, C. Hojny, R. van der Hulst, T. Koch, M. Lübbecke, S. J. Maher, F. Matter, E. Mühmer, B. Müller, M. E. Pfetsch, D. Rehfeldt, S. Schlein, F. Schlösser, F. Serrano, Y. Shinano, B. Sofranac, M. Turner, S. Vigerske, F. Wegscheider, P. Wellner, D. Weninger, J. Witzig, The SCIP Optimization Suite 8.0, ZIB-Report 21-41, Zuse Institute Berlin, 2021.
[27] CPLEX, IBM ILOG, V12. 1: User's manual for CPLEX, International Business Machines Corporation 46 (2009) 157.
[28] A. B. Yoo, M. A. Jette, M. Grondona, Slurm: Simple linux utility for resource management, in: D. Feitelson, L. Rudolph, U. Schwiegelshohn (Eds.), Job Scheduling Strategies for Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 44–60.
[29] D. Applegate, R. Bixby, V. Chvátal, W. Cook, Finding cuts in the TSP (A preliminary report), Technical Report 95-05, DIMACS, 1995.
[30] D. Sinha Roy, A. Masone, B. Golden, E. Wasil, Modeling and solving the intersection inspection rural postman problem, INFORMS Journal on Computing 33 (2021) 1245–1257.
[31] D. Sinha Roy, C. Defryn, B. Golden, E. Wasil, Data-driven optimization and statistical modeling to improve meter reading for utility companies, Computers & Operations Research 145 (2022) 105844.
[32] M. Dell'Amico, F. Maffioli, P. Värbrand, On prize-collecting tours and the asymmetric travelling salesman problem, International Transactions in Operational Research 2 (1995) 297–308.
[33] J. Hopcroft, R. Tarjan, Algorithm 447: efficient algorithms for graph manipulation, Communications of the ACM 16 (1973) 372–378.
[34] A. V. Goldberg, R. E. Tarjan, A new approach to the maximum-flow problem, Journal of the ACM (JACM) 35 (1988) 921–940.
[35] T. Achterberg, T. Koch, A. Martin, Branching rules revisited, Operations Research Letters 33 (2005) 42–54.

Table 5
Summary of the assumptions made by the literature on Pc-TSP. Assumptions include is the cost function metric? Is the graph complete? Is the returned solution a simple cycle? Is the graph directed?

Authors	Method	Metric?	Complete?	Simple cycle?	Directed?
Our work	Heuristics + branch & cut	no	no	yes	no
Awerbuch et al. [8]	Approximation algorithm	yes	no	no	no
Balas [6, 12, 13]	Polyhedral results	no	yes	yes	yes
Bérubé et al. [14]	Branch & cut	yes	yes	yes	no
Chaves and Lorena [18]	Clustering meta-heuristic	no	yes	yes	no
Dell'Amico et al. [19, 32]	Extension & Collapse	yes	yes	yes	no
Fischetti and Toth [16]	Lower bounds	no	yes	yes	yes
Pantuza and de Souza [15]	Lagrangian relaxation	yes	yes	yes	yes
Pedro et al. [17]	Meta-heuristic	no	yes	yes	yes

Appendix A Complete graph constructions

As is shown in Table 5, the literature on Pc-TSP assumes the input graph is complete and/or the cost function is metric. Given a sparse graph G with a non-metric cost function, one may ask why we cannot simply construct a complete graph G' with metric costs, then run an algorithm from the literature on G'? We give two such graph constructions in this section and explain why they do not work.

Least-cost path construction The first construction is as follows: for every pair of vertices u, v in G, find the least-cost path P from u to v in G, then set the cost of a new edge from u to v in G' equal to the cost of P. The problem with such a construction is that a feasible solution to the constructed instance on G' is not guaranteed to be a simple cycle in the original input graph G. Moreover, this construction may not be desirable when G is sparse because the number of edges in G' will be far greater than G, resulting in more variables to optimise when calling algorithms such as linear programs.

Dummy edge construction The second construction takes as input a sparse graph G and constructs a complete graph G' as follows. Add a vertices in G to G' and add all edges in G to G' with the same cost. Now, for all pairs of vertices u, v in G such that (u, v) is not an edge in G, we add a *dummy edge* (u, v) to G' and set the cost of (u, v) to be some very large constant C (e.g. we could take $C = \sum_{(i,j)\in E(G)} c(i, j)$. The constructed graph G' is clearly a complete graph and the optimal solution to any instance does not contain dummy edges. However, the major problem with this construction is that naively running a heuristic from the literature on G' may return a solution that contains a dummy edge, that is, the returned solution is not guaranteed to be feasible for the original instance. Moreover, the cost function is still non-metric, which naturally rules out any algorithm from the literature which requires a metric cost function. Finally, the dummy edge construction also increases the computational complexity of algorithms from the literature because the number of edges has been increased from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$.

Appendix B Proofs of Section 2

For completeness, we give proofs of Lemma 1 and a proof of correctness for the preprocessing algorithm.

B.1 Number of metric edges

Lemma 1. *Let G be a connected, undirected graph and $c : E(G) \to \mathbb{N}_0$ be a cost function on the edges. The number of metric edges in $E(G)$ is at least $n - 1$.*

Proof. Let $F \subseteq E(G)$ be the set of metric edges in G. Let the sub-graph G_F be defined by vertices $V(G_F) = V(G)$ and edges $E(G_F) = F$. To show $|F| \geq n - 1$, it is sufficient to show that G_F

is connected. By contradiction, suppose G_F is not connected and let $G_1, \ldots, G_k$ be the connected components of G_F. Since G is connected, there exists a least-cost path $\mathcal{P}_{uv} = (u, \ldots, x, y, \ldots, v)$ in G from $u \in V(G_i)$ to $v \in V(G_j)$ such that $i \neq j$ visits an edge (x, y) where $x \in V(G_a)$, $y \in V(G_b)$ and $a \neq b$. Note that every sub-path of $\mathcal{P}_{uv}$ is itself a least-cost path. Then edge (x, y) defines a least-cost path from x to y, so (x, y) is a metric edge and should have been in F. This implies every two adjacent edges in $\mathcal{P}_{uv}$ are in the same connected component of G_F, so all $u, v \in V(G_F)$ are in the same connected component and G_F is connected. □

B.2 Biconnected component pre-processing algorithm

A *biconnected component* of G is a maximal sub-graph such that the removal of a single vertex (and all edges incident on that vertex) does not disconnect the sub-graph. The biconnected components of an undirected graph can be found in time $\mathcal{O}(n + m)$ [33]. Let $B = \{C_1, \ldots, C_b\}$ be the set of biconnected components of G. Our pre-processing algorithm removes a vertex r from the input graph if there does not exist a $j \in \{1, \ldots, b\}$ such that r and v_1 are both in C_j. To see why our pre-processing algorithm is correct, note that every vertex u_i in a feasible tour $\mathcal{T}$ must be biconnected to v_1 (otherwise there would exist some vertex in the tour which must be visited more than once). A trivial example of a vertex not in the same biconnected component as v_1 is a leaf vertex with degree one.

We give a complete proof of correctness for our pre-processing algorithm. Given a subset of vertices $S \subseteq V(G)$, the *induced sub-graph* $G[S]$ contains all vertices in S and contains all edges in $E(G)$ that have both endpoints in S. Formally, a *biconnected component* of G is a maximal sub-graph such that the removal of a single vertex (and all edges incident on that vertex) does not disconnect the sub-graph. We define $C_1, \ldots, C_b \subseteq V(G)$ to be vertex sets such that $G[C_1], \ldots, G[C_b]$ are the biconnected components of G.

Lemma 5. *Let $G[C_1], G[C_2]$ be any two maximal biconnected components of G. Then $|C_1 \cap C_2| \leq 1$.*

Proof. Suppose that $|C_1 \cap C_2| > 1$ and let $C' = C_1 \cup C_2$. Our claim is that $G[C']$ is a biconnected component. To prove this claim, take two vertices $i, j \in C'$. If $i, j \in C_1$ (or if $i, j \in C_2$), then i and j are already in the same biconnected component. If $i \in C_1 \backslash (C_1 \cap C_2)$ and $j \in C_2 \backslash (C_1 \cap C_2)$, then remove a vertex $k_1 \in C_1 \cap C_2$. This removal does not disconnect the sub-graph $G[C']$ because there exists a path from i to j via some distinct $k_2 \in C_1 \cap C_2$ where $k_2 \neq k_1$ (since $|C_1 \cap C_2| > 1$). $G[C']$ is therefore a larger biconnected than $G[C_1]$ and $G[C_2]$, so $G[C_1], G[C_2]$ are not *maximal* biconnected component components. □

Lemma 6. *Assume G is connected. Let $G[C_1], G[C_2]$ be any two maximal biconnected components of G such that $C_1 \neq C_2$. Let tour $\mathcal{T}$ be a feasible solution to an instance of Pc-TSP. If $\mathcal{T}$ visits at least two vertices excluding v_1 in C_1, i.e. $|(V(\mathcal{T}) \cap C_1) \backslash \{v_1\}| \geq 2$, then $\mathcal{T}$ does not visit any vertices in $C_2 \backslash (C_1 \cap C_2)$.*

Proof. From Lemma 5, we have two cases: (i) $C_1 \cap C_2$ contains exactly one vertex r; or (ii) $C_1 \cap C_2 = \emptyset$. For case (i), if $\mathcal{T}$ visits a vertex $j \in C_2 \backslash (C_1 \cap C_2)$, then $\mathcal{T}$ must visit vertex r twice (once on the way to j, and once on the way back) which is contradiction to the definition of a tour. For case (ii), there exists exactly one biconnected component[5] $G[C_3]$ such that $C_3 \neq C_1 \neq C_2$ and $C_1 \cap C_3 \neq \emptyset$. Any tour that visits a vertex in C_2 must also visit a vertex in $j \in C_3 \backslash (C_1 \cap C_3)$. By the same logic as case (i), no tour can visit j, therefore no tour can visit C_2. □

Applying Lemma 6 to every biconnected component that does not contain the root leads to our main pre-processing result:

Corollary 7. *Let $R = \bigcup \{C_k | v_1 \in C_k, k \in \{1, \ldots, b\}\}$ be the set of vertices in the same biconnected component as the root vertex. There does not exist a feasible solution $\mathcal{T}$ that visits a vertex in $V(G) \backslash R$. Vertices in $V(G) \backslash R$ (and their adjacent edges) can safely be removed from the input graph G without removing any feasible solutions.*

[5] We can construct an undirected graph H by representing each biconnected components as a vertex in H. Note H will be acyclic. If G is connected, then H is a connected tree.

Appendix C Proofs of Section 3

Proposition 2. *Assume $\mathcal{P} \neq \mathcal{NP}$. Let G be any undirected graph. No polynomial-time algorithm A is guaranteed to find a feasible solution (if one exists) for every instance of Pc-TSP.*

Proof. By contradiction, assume A is guaranteed to find a feasible solution for every instance of Pc-TSP in polynomial-time. Let $\mathcal{I} = (G, c, p, Q, v_1)$ be an instance such that $Q = p(V(G))$. The only feasible solutions to $\mathcal{I}$ are Hamiltonian cycles of G. Then A is guaranteed to find a Hamiltonian cycle in polynomial-time for any simple, undirected graph G. Since finding a Hamiltonian cycle $\mathcal{NP}$-hard, this implies $\mathcal{P} = \mathcal{NP}$, a contradiction to our assumption. □

Appendix D Proofs of Section 4.2

Lemma 3. *In an undirected graph, the minimum cost of a tour $\mathcal{T}$ containing i and j is equal to the least-cost pair of vertex-disjoint paths $\mathcal{D}_1, \mathcal{D}_2$ between i and j.*

Proof. Split $\mathcal{T}$ into two vertex-disjoint paths $\mathcal{D}'_1, \mathcal{D}'_2$ starting at i and ending at j. If $c(E(\mathcal{D}'_1)) + c(E(\mathcal{D}'_2)) < c(E(\mathcal{D}_1)) + c(E(\mathcal{D}_2))$, then $\mathcal{D}_1, \mathcal{D}_2$ are not the least-cost vertex-disjoint paths between i and j. If $c(E(\mathcal{D}'_1)) + c(E(\mathcal{D}'_2)) > c(E(\mathcal{D}_1)) + c(E(\mathcal{D}_2))$, then there exists a tour $\mathcal{T}'$ containing i and j with less cost than $\mathcal{T}$, because we can construct $\mathcal{T}'$ by reversing the order of $\mathcal{D}_2$ and appending it to $\mathcal{D}_1$. So $\mathcal{D}'_1, \mathcal{D}'_2$ are least-cost vertex-disjoint paths and the minimum cost of the tour is $c(E(\mathcal{T})) = c(E(\mathcal{D}'_1)) + c(E(\mathcal{D}'_2)) = c(E(\mathcal{D}_1)) + c(E(\mathcal{D}_2))$. □

Appendix E Details of branch & cut algorithm

E.1 Separation Algorithms

In a branch & cut algorithm, a separation algorithm identifies violated inequalities. We give polynomial-time separation algorithms for the sub-tour elimination constraints (SECs).

E.1.1 Sub-tour Elimination Constraints

Definition 5. *The* support graph $G^\star$ *of the solution $(x^\star, y^\star)$ to linear program LP is defined by the vertex set $V(G^\star) = \{i \in V(G) : y^\star_i > 0\}$ and edge set $E(G^\star) = \{(i,j) \in E(G) : x^\star_{ij} > 0\}$.*

Constraints (7) are the SECs. Since there are an exponential number of SECs, we add violated SECs as a cutting plane. Given the support graph $G^\star$ of a linear program LP, we run the separation algorithm on $G^\star$ to check if a set $S \subseteq V(G^\star)$ violates (7). First, suppose the support graph is not connected. Let $G^\star_1, \ldots, G^\star_k$ be the k connected sub-graphs of $G^\star$. For every sub-graph $G^\star_l$ where $v_1 \notin V(G^\star_l)$ and $l \in \{1, \ldots, k\}$, apply (7) to each vertex $i \in V(G^\star_l)$ by setting $S = V(G^\star_l)$. Now suppose the support graph is connected. Treat the value of the edge variables $\boldsymbol{x}$ as the capacity of an edge. For each $i \in V(G^\star)$, find a minimum capacity (v_1, i)-cut in $G^\star$ with a min-cut max-flow algorithm in $\mathcal{O}(|V(G^\star)|^3)$ [34]. Let $(S_i, V(G^\star) \backslash S_i)$ be the minimum capacity cut, where $v_1 \in V(G^\star) \backslash S_i$. If Eq. (7) is violated for S_i, add it to the set of constraints in the linear program.

E.2 Variable Branching

When a linear program LP at node N in the branch & bound tree is solved, we may choose to branch on a variable x_{ij} that is fractional in LP. Branching on a variable means creating two new child nodes N^+, N^- of N with linear programs LP^+ and LP^- respectively. LP^+ is defined by inheriting all variables and constraints in LP and then rounding variable x_{ij} up to the nearest integer ($x_{ij} = 1$). LP^- is defined similarly by rounding x_{ij} down ($x_{ij} = 0$). Choosing which fractional variable to branch on has been the subject of extensive research [35]. In their branch & cut algorithm for Pc-TSP, Bérubé et al. [14] branch on the most fractional variable. However, more advanced branching strategies can

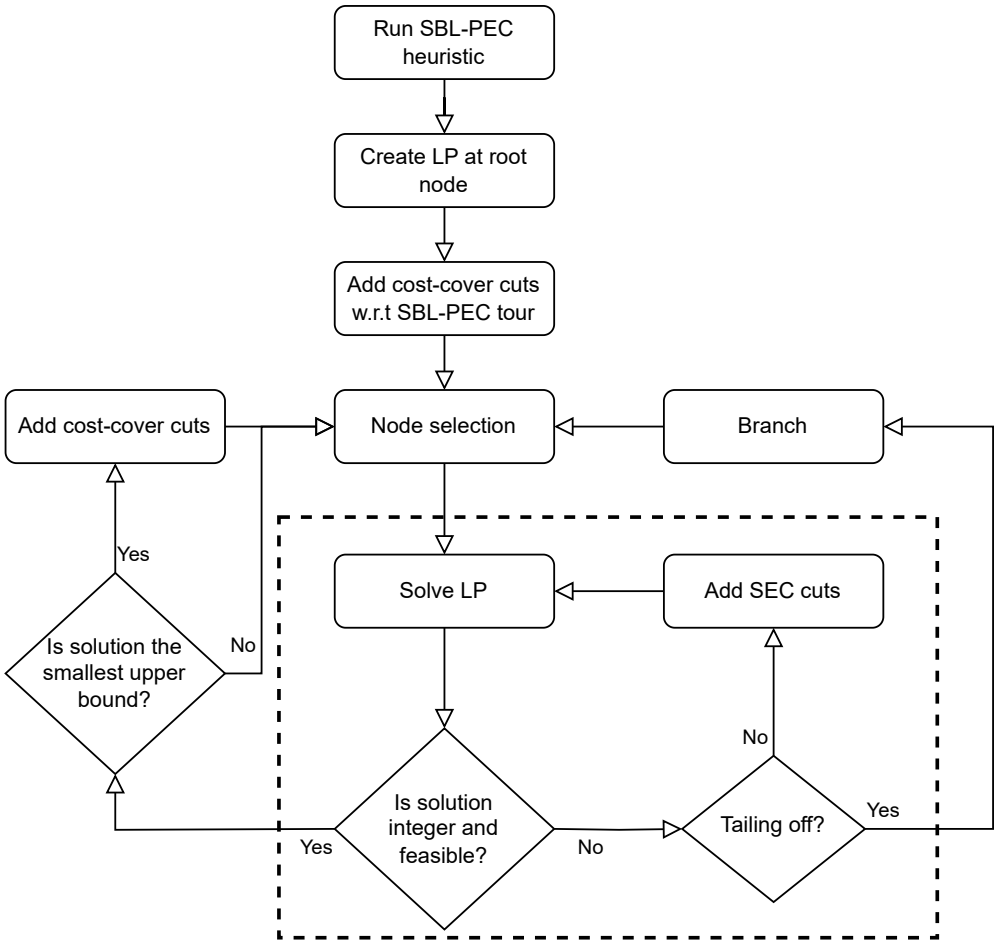


Figure 3: A flow diagram of our branch & cut algorithm. The dashed square box denotes operations that are repeated within a single node of the branch & bound tree.

Table 6
Statistics for the LAQ dataset. The original input graph is G and the pre-processed graph is H. The column $\frac{p(V(H))}{p(V(G))}$ denotes the total prize of H as a ratio of G. Column $D(G)$ is defined by (12).

Graph name	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	$p(V(G))$	$\lvert V(H)\rvert$	$\lvert E(H)\rvert$	$p(V(H))$	$\frac{p(V(H))}{p(V(G))}$	$\zeta(G,c)$	$D(G)$
laqbb	14,530	16,791	595,242	11,575	13,786	501,059	0.842	0.911	0.021
laqid	11,373	12,541	481,922	6,939	8,046	342,243	0.710	0.857	0.041
laqkx	13,431	15,427	579,233	10,396	12,328	474,423	0.819	0.909	0.022
laqws	6,770	7,519	365,947	4,223	4,912	256,929	0.702	0.899	0.036

yield smaller branch & bound trees, reducing the total computational time for solving a problem to optimality. Our branch & cut algorithm uses strong branching [29] at the top node of the branch & bound tree, then uses reliability branching [35] on all subsequent nodes in the tree.

We make one final addition to our branch & cut algorithm to prevent the phenomena of *tailing off*. This is where adding cuts to consecutive relaxed linear programs from the same node does little to improve the lower bound. To test if a node in the branch & bound tree is tailing off, we check if $\text{GAP}_t - \text{GAP}_{t-\tau} \leq \gamma$ where GAP_t is the GAP between the lower bound and upper bound after t iterations of adding cuts, $\tau \in \mathbb{N}$ is a parameter controlling the maximum number of iterations cuts can be added to a linear program without showing improvement, and $\gamma \in \mathbb{R}$ is the LP gap improvement threshold. If we find a node is tailing off, we branch on a variable, instead of waiting for more cuts to be added. After some experimentation (see Section J), we set $\tau = 5$ and $\gamma = 0.001$. To summarise, if after

Table 7
Statistics for the TSPLIB dataset aggregated by the sparsity level κ and the cost function (EUC or MST). The original input graph is G and the pre-processed graph is H. The column AVG $\left(\frac{p(V(H))}{p(V(G))}\right)$ denotes the average total prize of H as a ratio of G. The column AVG$(D(G))$ is an average of $D(G)$ as defined by (12).

	EUC		MST	
κ	AVG$(D(G))$	AVG$\left(\frac{p(V(H))}{p(V(G))}\right)$	AVG$(D(G))$	AVG$\left(\frac{p(V(H))}{p(V(G))}\right)$
5	0.062	0.912	0.138	0.912
10	0.050	0.993	0.140	0.993
15	0.044	0.999	0.164	0.999
20	0.041	1.000	0.189	1.000
25	0.037	1.000	0.184	1.000

five iterations of adding cuts to a linear program the GAP has improved by at most 0.001, we branch on a variable instead of adding more cuts. Figure 3 shows a visualisation of our branch & cut algorithm.

Appendix F Datasets and preprocessing

Before outlining the complementary experiments, we give more details about the LAQ and TSPLIB datasets. Moreover, for the TSPLIB dataset, we introduce a *metric* cost function called EUC. It is defined simply as the Euclidean distance between the coordinates of two vertices $i, j \in V(G)$ in the TSPLIB dataset: $c(i,j) = ||i - j||_2$.

Table 6 shows a summary of instances in the LAQ dataset and Table 7 shows a summary of instances in the TSPLIB dataset. The table includes a graph property $D : G \to [0,1]$ which is defined to be the largest prize of a pair of least-cost vertex-disjoint paths $\mathcal{D}_1, \mathcal{D}_2$ from v_1 to $t \in V(G)$ as a ratio of the total prize of the graph:

$$D(G) = \frac{1}{p(V(G))} \max_{t \in V(G)} \{p(V(\mathcal{D}_1)) + p(V(\mathcal{D}_2)) - p(t)\} \tag{12}$$

F.1 Pre-processing

The biconnected component pre-processing algorithm described in Section B.2 removes more vertices (and more prize) from the LAQ input graph than on the TSPLIB input graph because the LAQ graphs are more sparse than TSPLIB. From Table 7, we see that on TSPLIB the average prize removed from the graph during pre-processing increases as the graph becomes more sparse (κ decreases). Even when $\kappa = 5$ on TSPLIB, less than 10% of the prize is removed from the input graph during pre-processing. In contrast on the LAQ dataset in Table 6, our pre-processing algorithm removes more than 25% of the prize on laqid and laqws, and removes 15.8% and 18.1% on laqbb and laqkx. To summarise, our pre-processing algorithm is effective at reducing the input size when the graph is sparse.

Appendix G Additional heuristic results

Section G shows heuristics results in the same manor as the TSPLIB results of Table 2 but with the extra cost function EUC and with two addtional heuristics called BFS-PEC and SBL-EC. After generating an initial tour with BFS, the BFS-PEC heuristic increases the prize with the path extension heuristic, then decreases the cost with the path collapse heuristic. After generating an initial tour with SBL, SBL-EC increases the prize with `Extension`, then decreasing the cost with `Collapse` [19].

Both BFS-PEC and SBL-PEC find feasible solutions to every instance of TSPLIB when $\kappa \geq 10$ and $\alpha \leq 0.50$. BFS-PEC and SBL-PEC do not find feasible solutions to every instance when the graph is

Table 8
The $\overline{\text{GAP}}$ and FEAS for five heuristics on the TSPLIB dataset across both the EUC and MST cost functions and across different sparsity levels κ. Each row corresponds to 135 instances. $\overline{\text{GAP}}$ is calculated using the largest lower bound (LB) and so the $\overline{\text{GAP}}$ is an overestimate for each entry.

		BFS-EC		BFS-PEC		SBL-EC		SBL		SBL-PEC	
Cost	κ	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS
EUC	5	0.506	11	1.399	42	0.675	124	0.041	18	0.580	125
	10	2.465	79	2.536	93	0.750	135	0.162	15	0.735	135
	15	3.358	117	3.372	121	0.803	135	0.193	12	0.797	135
	20	3.036	106	3.594	120	0.906	135	0.242	11	0.799	135
	25	3.170	107	2.919	121	0.816	135	0.460	8	0.776	135
MST	5	2.555	8	3.325	62	1.642	122	0.097	46	1.075	123
	10	5.430	71	6.109	97	1.790	135	0.124	48	1.333	135
	15	8.217	122	6.617	125	1.863	135	0.153	51	1.421	135
	20	6.619	120	6.141	133	1.800	135	0.183	51	1.469	135
	25	8.125	126	5.623	133	1.825	135	0.239	54	1.466	135

Table 9
For a given α, each row reports the number of feasible solutions (FEAS) found by a heuristic on the TSPLIB dataset across both the EUC and MST cost functions. The largest possible value of FEAS is 270.

α	BFS-EC	BFS-PEC	SBL	SBL-PEC
0.05	204	270	181	270
0.10	194	270	107	270
0.25	187	270	26	270
0.50	160	270	0	270
0.75	122	246	0	248

sparse ($\kappa = 5$) and the quota is large ($\alpha = 0.75$): BFS-PEC finds 246 out of 270 feasible solutions and SBL-PEC finds 248 out of 270 across both cost functions. The $\overline{\text{GAP}}$ of SBL-PEC is smaller than BFS-PEC across every κ for both cost functions, but the difference in the $\overline{\text{GAP}}$ between the two algorithms is more noticeable on the MST cost function. When the cost function is non-metric (MST), the difference in the $\overline{\text{GAP}}$ between BFS-PEC and SBL-PEC is greater than 0.33 for all κ; and when the cost function is metric (EUC), the difference in the $\overline{\text{GAP}}$ between BFS-PEC and SBL-PEC is less than 0.11 for all κ. This suggests *initialising PEC with SBL is particularly effective on non-metric cost functions.*

The GAP of SBL-EC is greater than BFS-PEC and SBL-PEC across both cost functions and all values of κ. Moreover, SBL-EC finds less feasible solutions than BFS-PEC and SBL-PEC. Compared to BFS-EC, SBL-EC finds more feasible solutions, suggesting it is better to initialise EC with SBL than BFS: this is the same conclusion we came to for initialising PEC. The $\overline{\text{GAP}}$ of SBL-EC is generally greater than BFS-EC on the EUC cost function, but less than BFS-EC on MST cost function (however these figures are not directly comparable because SBL-EC finds more feasible solutions).

Whilst SBL is highly effective on the LAQ dataset, Table 9 shows SBL is less likely to find a feasible solution as α increases on the TSBLIB dataset. This is unsurprising because a pair of least-cost vertex-disjoint paths between v_1 and $t \in V(G)$ is unlikely to collect a lot of prize, thus if α is large then SBL will not collect enough prize to meet the quota. More specifically, if $D(G) > \alpha$ then SBL will not find a feasible solution. For example, Table 7 shows that AVG$(D(G)) = 0.062$ on the EUC cost function when $\kappa = 5$, so if $\alpha = 0.75$ then SBL will not find a feasible solution. One reason SBL is highly effective on the LAQ dataset is that α is small: for example, a quota of 5000 on the laqbb graph gives $\alpha \approx 0.01$ which is less than $D(G) = 0.021$. To conclude our analysis of heuristics, SBL is effective as a standalone heuristic on the LAQ dataset, whilst SBL-PEC outperforms all other algorithms on the TSPLIB dataset across two cost functions and varying levels of sparsity.

Table 10
Evaluation of our branch & cut algorithm using no cost-cover (NoCC) inequalities on the London air quality dataset for five different quotas. There are four instances for each quota.

	No cost-cover cuts			
Quota	$\overline{\text{PRE-CUTS}}$	$\overline{\text{TIME}}$ (s)	$\overline{\text{GAP}}$	OPT
1000	0	3782.8	0.544	3
2000	0	6594.3	0.348	3
3000	0	12 576.0	0.336	1
4000	0	14 432.3	0.269	0
5000	0	14 434.4	0.325	0

Table 11
Evaluation of our branch & cut algorithm using no cost-cover (NoCC) inequalities on the TSPLIB dataset for five different values of α and two different cost functions. Each row corresponds to 135 instances.

		No cost-cover cuts			
Cost	α	$\overline{\text{PRE-CUTS}}$	$\overline{\text{TIME}}$ (s)	$\overline{\text{GAP}}$	OPT
EUC	0.05	0	2566.8	0.071	118
	0.10	0	3883.4	0.133	104
	0.25	0	4606.2	0.149	101
	0.50	0	5393.6	0.123	95
	0.75	0	5032.7	0.088	94
MST	0.05	0	3449.7	0.080	111
	0.10	0	5604.6	0.373	87
	0.25	0	9048.4	0.742	54
	0.50	0	9285.3	0.859	51
	0.75	0	8507.0	0.649	60

Table 12
Evaluation of our branch & cut algorithm using disjoint-paths vs shortest-path cost-cover inequalities on the TSPLIB dataset with EUC cost function for five different values of α. Each row corresponds to 135 instances.

	Disjoint-paths cost-cover cuts				Shortest-path cost-cover cuts			
α	$\overline{\text{PRE-CUTS}}$	$\overline{\text{TIME}}$ (s)	$\overline{\text{GAP}}$	OPT	$\overline{\text{PRE-CUTS}}$	$\overline{\text{TIME}}$ (s)	$\overline{\text{GAP}}$	OPT
0.05	57	2239.8	0.075	120	50	2238.9	0.084	120
0.10	9	3846.8	0.139	103	8	3782.4	0.141	104
0.25	0	4630.1	0.130	101	0	4715.3	0.137	101
0.50	0	5464.1	0.125	93	0	5265.7	0.113	96
0.75	0	5167.2	0.056	96	0	4940.3	0.091	96

Appendix H Additional cost-cover results

In this section, we conduct two additional experiments. Firstly, we ask: does running a branch & cut algorithm with cost-cover inequalities actually make a different? We find an affirmative answer by running our branch & cut algorithm with no cost-cover (NoCC) inequalities and compare it against DPCC and SCPCC. Second, we analyse DPCC and SPCC on a *metric* cost function (EUC).

H.1 No cost cover inequalities

Table 10 shows that, on the LAQ dataset, adding no cost cover inequalities (NoCC) vastly *underperforms* DPCC and SPCC from Table 3. NoCC finds just 7 out of 20 optimal solutions, whereas DPCC and SPCC both find 17 out of 20. We conclude that adding cost cover inequalities is highly advantageous on the

LAQ dataset.

On TSPLIB with MST costs when $\alpha = 0.05$ and $\alpha = 0.10$, NoCC adds zero cost-cover cuts, so DPCC finds 9 more and 5 more optimal solutions than NoCC for $\alpha = 0.05$ and $\alpha = 0.10$ respectively. The $\overline{\text{TIME}}$ of DPCC is less than NoCC when $\alpha = 0.05$ and $\alpha = 0.10$: DPCC terminates 1,530 and 610 seconds faster respectively than NoCC.

However, when $\alpha \in \{0.25, 0.50, 0.75\}$ for MST costs, the number of $\overline{\text{PRE-CUTS}}$ is small or zero for both DPCC and SPCC. Indeed when DPCC and SPCC add zero cost-cover cuts in total, then DPCC, SPCC, and NoCC are nearly equivalent branch & cut algorithms (minus the added computational time needed to run the cost-cover separation algorithm). This results in less than 90 second $\overline{\text{TIME}}$ difference between DPCC, SPCC, and NoCC for all $\alpha \in \{0.25, 0.50, 0.75\}$. Small variations in the $\overline{\text{GAP}}$ and OPT further support that there is little difference on MST costs between DPCC, SPCC, and NoCC when $\alpha \in \{0.25, 0.50, 0.75\}$: SPCC has the smallest $\overline{\text{GAP}}$ by 0.004 when $\alpha = 0.25$; DPCC has the smallest $\overline{\text{GAP}}$ by 0.032 when $\alpha = 0.50$; and NoCC has the smallest $\overline{\text{GAP}}$ by 0.003 when $\alpha = 0.75$.

H.2 Metric costs

The EUC cost function on the TSPLIB dataset, as defined in Section F, is a metric cost function (see Definition 3). The cost-cover results for EUC on TSPLIB are shown in Table 12. For each α, less $\overline{\text{PRE-CUTS}}$ are added by DPCC on EUC compared to when the cost function is MST. For example, when $\alpha = 0.05$, the number of $\overline{\text{PRE-CUTS}}$ added by DPCC is 56.7 on EUC compared to 192 on MST. Moreover, for all values of α, the difference in the number of $\overline{\text{PRE-CUTS}}$ between DPCC and SPCC on the EUC cost function is smaller than on the difference MST cost function. As a result, conclude that for the EUC cost function, small variations across in the $\overline{\text{TIME}}$, $\overline{\text{GAP}}$ and OPT show there is little difference between DPCC, SPCC, and NoCC.

Appendix I Varying α on LAQ dataset

Our results show a discrepancy between the LAQ and TSPLIB datasets. For completeness, we run the same experiments as Section 5 on the LAQ dataset for both our heuristics and our branch & cut algorithm, but instead of setting the quota to be $Q \in \{1000, 2000, 3000, 4000, 5000\}$, we set the quota $Q = \alpha \cdot p(V(G))$ in the same manor as our experiments on TSPLIB with $\alpha \in \{0.05, 0.10, 0.25, 0.50, 0.75\}$. The experimental results for heuristics are summarised in Table 13 and for different cost-cover cuts in Table 14.

First, as we noted in Section F.1, our biconnected component pre-processing algorithm described in Section 2 removes more vertices (and more prize) from the LAQ input graph than on the TSPLIB input graph because the LAQ graphs are more sparse. From Table 7, we see that on TSPLIB the average prize removed from the graph during pre-processing is less than 10% across all sparsity levels κ. Contrast this to the LAQ dataset in Table 6, where on two LAQ instances (laqid and laqws), our pre-processing algorithm removes more than 25% of the prize in the original input graph. This means when $\alpha = 0.75$ for laqid and laqws, the total prize of the pre-processed graph is not greater than the quota, which trivially shows the original instance does not have a feasible solution. On laqbb and laqkx, the pre-processing algorithm removes 15.8% and 18.1% of the total prize of the original graph. Whilst in theory the two graphs contains sufficient prize for a quota set with $\alpha = 0.75$, the bottom row of Table 14 shows our branch & cut algorithm quickly determines there are no feasible solutions to the two instances. Moreover, when $\alpha = 0.50$, neither our heuristics nor our branch & cut algorithm found a feasible solution to any of the four instances. The branch & cut algorithm was able to obtain a lower bound, but not able to prove infeasibility within the four hour time limit, and it remains open whether there exists a feasible solution to these four instances when $\alpha = 0.50$.

Focusing on $\alpha \in \{0.05, 0.10, 0.25\}$, Table 13 shows that both BFS-EC and SBL heuristics find no feasible solutions to these LAQ instances. Contrast this to TSPLIB (Table 9) where BFS-EC finds 204, 194, 187 feasible solutions for $\alpha = 0.05, 0.10, 0.25$ respectively. Similarly SBL finds 181, 107, 26 feasible solutions on TSPLIB for $\alpha = 0.05, 0.10, 0.25$ respectively. As noted in Section 5.1, EC on LAQ cannot

Table 13
A comparison of heuristic performance on the LAQ dataset for each $\alpha \in \{0.05, 0.10, 0.25, 0.50, 0.75\}$. We measure the $\overline{\text{GAP}}$ and FEAS across all four instances (laqbb, laqid, laqkx, laqws). "nan" means "not a number" where we could not calculate a $\overline{\text{GAP}}$ due to the heuristic finding zero feasible solutions to the four instances.

	BFS-EC		BFS-PEC		SBL		SBL-PEC	
α	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS	$\overline{\text{GAP}}$	FEAS
0.05	nan	0	2.286	4	nan	0	5.697	4
0.10	nan	0	1.601	4	nan	0	2.141	4
0.25	nan	0	0.609	3	nan	0	0.392	4
0.50	nan	0	nan	0	nan	0	nan	0
0.75	nan	0	nan	0	nan	0	nan	0

Table 14
Evaluation of our branch & cut algorithm with DPCC and SPCC on the London air quality dataset for $\alpha \in \{0.05, 0.10, 0.25, 0.50, 0.75\}$. There are four instances for each quota. "nan" means "not a number" and represents an upper bound (UB) that could not be calculated for all four instances for a given α (the corresponding $\overline{\text{GAP}}$ also cannot be calculated). "inf" means "infeasible" and represents instances for which no feasible solution exists for all four instances.

	Disjoint-paths cost-cover cuts				Shortest-path cost-cover cuts			
α	$\overline{\text{GAP}}$	$\overline{\text{LB}}$	$\overline{\text{UB}}$	FEAS	$\overline{\text{GAP}}$	$\overline{\text{LB}}$	$\overline{\text{UB}}$	FEAS
0.05	0.292	4.30×10^5	5.54×10^5	4	0.287	4.32×10^5	5.54×10^5	4
0.10	0.222	9.18×10^5	1.12×10^6	4	0.221	9.18×10^5	1.12×10^6	4
0.25	0.159	2.47×10^6	2.88×10^6	4	0.159	2.47×10^6	2.88×10^6	4
0.50	nan	5.51×10^6	nan	0	nan	5.54×10^6	nan	0
0.75	inf	inf	inf	0	inf	inf	inf	0

increase the prize of the initial tour generated by BFS due to the sparsity of the graph, resulting in zero feasible solutions to $\alpha \in \{0.05, 0.10, 0.25\}$. To explain why SBL does not find feasible solutions for $\alpha \in \{0.05, 0.10, 0.25\}$, notice that for all instances in the LAQ experiment where $\alpha \geq 0.05$, the function $D(G)$ (see (12) for definition) is less than α: this implies there does not exist a pair of least-cost vertex-disjoint paths $\mathcal{D}_1, \mathcal{D}_2$ from v_1 to any vertex $t \in V(G)$ such that $p(V(\mathcal{D}_1))+p(V(\mathcal{D}_2))-p(t) \geq Q$. Consequently in Table 13, SBL reports finding zero feasible solutions. Similarly in Table 14 the branch & cut algorithm adds zero DPCC cuts and zero SPCC cuts for all values of α.

Appendix J Branching strategy & tailing off

In this section, we justify our *Strong at Tree Top* (STT) branching strategy from Section E.2 against *Relative Pseudo Cost* (RPC) branching with empirical experiments across both datasets. We also include an analysis of how the parameters for preventing tailing off were chosen. To isolate the effects of the branching strategy, we do not add any cost-cover cuts and SCIP features are turned off. Our brief experiments with most fractional branching for Pc-TSP as proposed by [14] yielded very large branch & bound trees at great computational cost and are excluded from this analysis. For the STT strategy, we set a depth limit of $\Delta \in \{1, 5, 10\}$. RPC has no depth limit (marked as N.A.). For both branching strategies, we set the tailing off threshold to be $\gamma \in \{0.001, 0.01\}$ and the maximum number of iterations to be $\tau \in \{5, 10, \infty\}$. This corresponds to 24 different parameter configurations. We run each parameter configuration on a subset of both datasets. On LAQ, we test each graph (laqbb, laqid, laqkx, laqws) for quotas $Q \in \{1000, 2000, 3000\}$. For the TSPLIB dataset, we restrict the instances to four graphs (att48, st70, eil76, rat195) with $\kappa \in \{10, 20\}$ and $\alpha \in \{0.25, 0.75\}$.

Table 15 and Table 16 summarise the results for each parameter configuration for the LAQ and

Table 15
Branching strategy and tailing off results on the LAQ dataset. The bold row in both tables highlights the parameter configuration for STT as chosen for our branch & cut algorithm in Section E.2.

Δ	Strategy	γ	τ	$\overline{\text{TIME}}$	$\overline{\text{GAP}}$	OPT	FEAS	$\overline{\text{NODES}}$	$\overline{\text{SEC}}$
N.A.	RPC	0.001	∞	6976.32	0.34	8	15	225.60	664.60
			5	7864.46	0.37	7	15	224.20	673.39
			10	6969.96	0.35	8	15	206.53	1035.79
		0.010	∞	7148.73	0.35	9	15	180.07	4157.94
			5	7866.65	0.36	7	15	234.33	352.34
			10	7325.07	0.35	9	15	228.87	1331.51
1	STT	0.001	∞	6537.38	0.30	10	15	2288.00	3951.24
			5	**5837.82**	**0.30**	**10**	**15**	**1756.67**	**777.40**
			10	5968.30	0.31	10	15	2002.53	1690.16
		0.010	∞	7044.70	0.31	9	15	1756.13	1272.70
			5	6447.92	0.30	10	15	1469.93	1053.50
			10	7346.26	0.29	10	15	2278.87	3187.37
5	STT	0.001	∞	6505.01	0.30	9	15	2285.27	1065.81
			5	6879.43	0.30	10	15	2209.20	4084.65
			10	6682.06	0.30	10	15	2514.60	6930.11
		0.010	∞	5929.53	0.31	10	15	1916.07	1713.18
			5	5996.21	0.30	10	15	2096.87	869.78
			10	6220.06	0.30	10	15	1682.47	882.89
10	STT	0.001	∞	6161.93	0.33	10	15	1543.93	673.33
			5	6825.21	0.34	9	15	1252.47	583.90
			10	6552.19	0.35	9	15	1295.20	743.81
		0.010	∞	5835.25	0.34	10	15	1398.40	622.91
			5	6302.87	0.32	10	15	1198.33	587.86
			10	6094.66	0.34	10	15	2583.47	451.92

TSPLIB datasets respectively. The bold row in both tables highlights the parameter configuration for STT where $\Delta = 1$, $\gamma = 0.001$ and $\tau = 5$ as chosen for our branch & cut algorithm in Section E.2. Across both datasets, the tables show the $\overline{\text{GAP}}$ of STT is less than RPC. On TSPLIB, STT is strongest when $\Delta \in \{1, 5\}$. On LAQ, the smallest $\overline{\text{GAP}}$ is 0.04 when STT uses $\Delta = 1, \gamma = 0.001$ and $\tau = 5$. Since STT with $\Delta = 1, \gamma = 0.001$ and $\tau = 5$ also performs well on TSPLIB, we chose this as our branching and tailing off parameter configuration. STT with $\Delta = 5, \gamma = 0.001$ and $\tau = 10$ also has a small $\overline{\text{GAP}}$ of 0.05 on LAQ, but the $\overline{\text{NODES}}$, $\overline{\text{SEC}}$ and $\overline{\text{TIME}}$ on TSPLIB were larger.

Table 16
Branching strategy and tailing off results on the TSPLIB dataset. The bold row in both tables highlights the parameter configuration for STT as chosen for our branch & cut algorithm in Section E.2.

Δ	Strategy	γ	τ	$\overline{\text{TIME}}$	$\overline{\text{GAP}}$	OPT	FEAS	$\overline{\text{NODES}}$	$\overline{\text{SEC}}$
N.A.	RPC	0.001	∞	2678.02	0.13	41	48	605.81	109.29
			5	2740.72	0.11	41	48	780.85	79.16
			10	2691.51	0.11	41	48	562.29	85.60
		0.010	∞	2824.69	0.10	40	48	716.02	107.75
			5	2631.68	0.11	41	48	607.96	72.49
			10	2675.12	0.12	41	48	601.38	86.88
1	STT	0.001	∞	2036.82	0.08	42	47	3370.28	53.36
			5	**1967.12**	**0.04**	**43**	**48**	**4173.98**	**53.19**
			10	1935.05	0.08	43	48	3520.27	56.41
		0.010	∞	1839.44	0.10	42	47	2773.40	53.18
			5	2312.07	0.09	42	48	3996.08	49.19
			10	2165.22	0.09	43	48	3617.98	60.11
5	STT	0.001	∞	2043.23	0.08	43	48	4211.52	47.72
			5	1923.07	0.07	43	48	3594.90	46.65
			10	2104.86	0.05	43	48	4278.96	47.23
		0.010	∞	2171.19	0.10	43	48	3844.83	48.92
			5	2278.60	0.09	42	48	3671.90	42.75
			10	2171.48	0.08	43	48	3867.56	47.05
10	STT	0.001	∞	2342.35	0.08	42	48	2798.96	58.84
			5	2429.36	0.09	42	48	2993.62	42.78
			10	2511.78	0.08	42	48	3373.04	58.43
		0.010	∞	2474.73	0.08	42	48	2372.92	60.15
			5	2345.09	0.08	42	48	2752.56	47.58
			10	2266.14	0.07	43	48	2716.92	56.13