The Travelling Salesman Problem

Ruben Mitchell s2065795, Emily Qiu s2091344, Sia Xu s1953913

Year 4 Project
School of Mathematics
University of Edinburgh
Supervisor: Dr. Kit Searle
January 2024

Abstract

The Traveling Salesman Problem (TSP), a pivotal question in combinatorial optimization, asks for the shortest possible route that visits a set of cities exactly once and returns to the original city. Practical applications including logistics, planning and the design of circuits. Despite extensive research, finding efficient solutions for large instances remains a challenge, highlighting a significant gap in applying theoretical solutions to practical, large-scale problems. This report aims to assess various algorithmic strategies against the TSPLIB dataset to identify the most effective approaches for symmetric TSP instances. Utilizing a combination of heuristic and exact algorithms, implemented through a structured Python project, this research delves into algorithmic performance with a focus on practical applicability. The key finding reveals that certain heuristic methods offer a viable balance between computational efficiency and solution accuracy for large instances. This insight suggests a promising direction for future research in optimizing routes in real-world scenarios, thereby contributing significantly to the fields of computational problem-solving and operations research.

Keywords: Algorithmic strategies, Operations research, Optimization

Contents

Chapter 1

Introduction

1.1 Why TSP

If you draw 14 dots on a page, there are more distinct routes to take between these dots than there are seconds in your life. Now, imagine the task of finding the shortest possible path that visits each of these exactly once and returns to the start. This is the essence of the Traveling Salesman Problem and it is precisely the realization that we don't have enough time to try every option that makes this problem so interesting.

Beyond the capitalist impulse to squeeze savings out of routing problems, the allure of the Traveling Salesman Problem (TSP) has captivated mathematicians with its complexity. The TSP exemplifies a class of problems for which solutions can be verified swiftly — in polynomial time, to be exact — yet finding these solutions appears to be significantly more challenging. This dichotomy between ease of verification and apparent difficulty of solution discovery is the foundation of the NP (Non-deterministic Polynomial-time) class of computational problems.

Discovering a polynomial-time solution to the TSP would not only revolutionize the field of optimization but could unravel the foundations of modern cryptography. Those intrigued by a less complicated reward could also claim the Millennium Prize from the Clay Mathematics Institute by having resolved the $P = NP$ problem formulated in Cook 1971, perhaps the most famous problem in all of computer science.

Back on earth, this complexity has challenged mathematicians for over a century, spurring the development of optimisation algorithms and heuristics which are used in a variety of applications and underscore entire fields of mathematics today.

1.2 Applications of TSP

We now highlight a subset of practical applications of the TSP to affirm its direct practical value.

- Logistics: the most direct application of the TSP, the optimization of routes and thus the maximisation of profits.

- Telecommunications: Network design and the optimization of cable layouts so as to minimise material use.

- Astronomy: the order in which stars or galaxies are observed by telescopes may be formulated as a TSP problem, with the goal of minimizing the movement between targets resulting in maximum observation time.

This paper explores a range of heuristic solutions, and crucially an exact solution procedure that can be used to solve reasonably large TSP problems in any of these applications.

1.3 Problem Statement

This paper embarks on an exploration of the Traveling Salesman Problem, situated firmly within the realm of mathematical optimization. Our goal is to elucidate the foundational principles behind the branch and cut algorithm, a linchpin of integer programming.

Through this investigation, we endeavor to shed light on the astronomical complexity of the TSP, and the NP class problems that it represents, articulating some sense of the reasoning and intuition which has persuaded 99% of experts in the domain that $P \neq NP$ Gasarch 2019. We will dissect the specifics of the engagement between the branch and cut algorithm and the TSP, alongside an examination of a specific set of algorithmic and heuristic augmentations.

We ultimately develop a version of our findings into a practical application which aims to solve TSP instances of moderate size (less than 1000 nodes) within a reasonable time frame (a few hours). Acknowledging the groundbreaking work embodied by Concorde (see D. Applegate, Bixby, and Chvátal 2006), the current state of the art TSP solver (holding this position for over two decades), we highlight that this solution implementation was developed through years of dedicated effort by the collective genius and hard work of a team of esteemed mathematicians David L. Applegate et al. 2007. This paper cannot realistically aim to compete with such a monumental feat in terms of raw performance. Instead, it seeks to contribute to the ongoing discourse in mathematical optimization by providing deeper insights into the algorithmic underpinnings and interplay that govern solution strategies for the TSP.

1.4 Objectives and Structure

This report is dedicated to an in-depth exploration of the Traveling Salesman Problem. The primary objectives of this report are to elucidate the theoretical foundations of the TSP, and assess various algorithmic strategies aimed at addressing the problem, focusing specifically on their performance against the TSPLIB dataset [1]. Through comprehensive analysis and comparison, this report endeavors to provide a thorough understanding of the strengths and limitations of each algorithmic approach, providing insights into their suitability for symmetric TSP instances and contributing to the broader understanding of optimization strategies within computational problem-solving contexts.

This paper is organized into six main sections, each offering a distinct perspective on the study of the TSP and its algorithmic solutions. Essential theoretical foundations, including optimization principles and graph theory, are crucial for understanding TSP solutions and are introduced in Chapter 2. In Chapter 3, we delve into the specifics of algorithmic approaches, ranging from integer linear programming formulations to heuristic and exact solutions. Then, in Chapter 4 we detail the application of these algorithms on a Python project, including the architecture and class structures used. Finally, the computational results from solving the symmetric TSP instances from the TSPLIB dataset are given in Chapter 5 and we conclude the report by summarizing the findings in Chapter 6, discussing the implication of the algorithmic performance, and suggesting directions for future research.

[1]A common test body of instances collected by Gerhard Reinelt. Available at: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Chapter 2

Preliminaries

This chapter introduces the relevant mathematical optimization and graph theory concepts central to this paper's analysis of the TSP. We begin by delineating the principles of linear programming (LP) and integer linear programming (ILP), and the frameworks of the branch & bound and branch & cut algorithms. Thereafter, we explore essential graph theory preliminaries, providing the requisite to the TSP and its cut generation approaches. Readers with extensive familiarity in mathematical optimization and graph theory may opt to proceed directly to Chapter 3 for a discussion specific to the TSP.

2.1 Optimisation & Polyhedral Theory

2.1.1 Linear programming

Linear programming (LP) is a technique for minimising some linear objective function subject to a set of linear inequalities (constraints).[1] That is LP allows us to:

$$\begin{array}{rr} \textbf{Minimise:} & c_1x_1 + \cdots + c_nx_n \\ \textbf{subject to:} & a_{11}x_1 + \cdots + a_{1n}x_n \leq b_1, \\ & \vdots \qquad\qquad\quad \vdots \quad\; \vdots \\ & a_{m1}x_1 + \cdots + a_{mn}x_n \leq b_m, \\ & x_1, ..., x_n \geq 0. \end{array}$$

Definition 2.1.1 (Closed Half-Space)**.** A *closed half-space* in $\mathbb{R}^n$ is a set: $\{x_1, \ldots, x_n \mid a_1x_1 + \ldots + a_nx_n \leq b\}$, where $\exists i \in \{1, \ldots, n\} : a_i \neq 0$.

Definition 2.1.2 (Polyhedron)**.** A *polyhedron* is a non-empty finite intersection of closed half-spaces.[2]

Clearly, each of the linear inequalities in an LP model represents a closed half-space of $\mathbb{R}^n$, where n is the number of variables in the objective function. If the intersection of these half-spaces is not null, that is if the constraints are not contradictory, their intersection delimits a polyhedron P in $\mathbb{R}^n$, referred to as the *feasible region* of the LP.

Definition 2.1.3 (Convex Hull)**.** The *convex hull* of a set of points S, in Euclidean space is the smallest convex set containing all the points in S, denoted $\text{conv}\{S\}$.

[1]For the purposes of this paper, discussions of LP are confined to minimization problems.
[2]All polyhedrons are convex as an intersection of half-spaces must be convex.

Definition 2.1.4 (Polytope)**.** A *polytope* is a bounded polyhedron, or equivalently the convex hull of some non-empty finite set.

Thus a bounded and feasible LP's feasible region P, is always a polytope, thereby reducing the problem to identifying the minimum of the objective function within P.

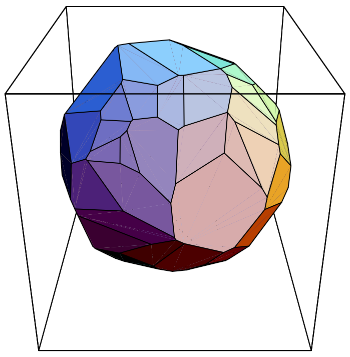


Figure 2.1: A polytope in $\mathbb{R}^3$.

Figure 2.1 demonstrates a polytope in $\mathbb{R}^3$ which can be viewed as the intersection of the half-planes delimiting its faces, or equivalently as the convex hull of its exterior vertices.

Theorem 2.1.1 (The Fundamental Theorem of Linear Programming)**.** *If a linear programming problem is feasible and bounded, then the objective function must achieve its minimum at a vertex of the feasible region (polytope).*

In order to minimise an objective function within a feasible region (polytope) a linear programming algorithm is typically employed. There exist many sophisticated such algorithms including IBM's ILOG CPLEX Optimiser, which we utilise in this paper.[3] To offer a high-level intuition of how these algorithms operate, we briefly describe Dantzig's simplex algorithm. This method leverages the Theorem 2.1.1 by evaluating the objective function at the vertices of the polytope and then 'walking' to adjacent vertices which decrease the objective function's value. This process continues until the algorithm converges to the minimum.

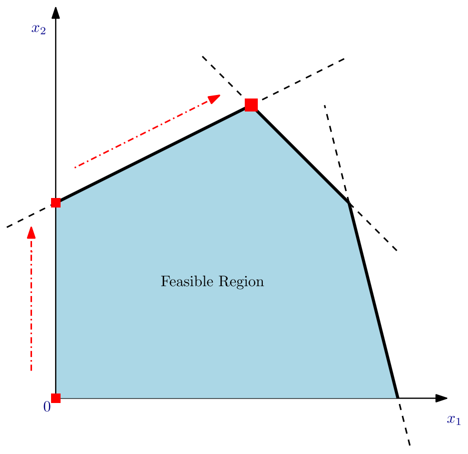


Figure 2.2: A visualisation of Dantzig's simplex algorithm on a polytope in $\mathbb{R}^2$.

Figure 2.2 gives an example of Dantzig's simplex algorithm wherein the feasible region is a convex polygon with the red vertices displaying the walk of a simplex algorithm in minimising objective function: $-(x_1 + 2x_2)$.[4]

[3]Other available LP algorithms include *CLP*, *GLPK* and *MINOS* (see Gearhart et al. 2013).
[4]Minimising $-(x_1 + 2x_2)$ is equivalent to maximising $x_1 + 2x_2$.

2.1.2 Integer programming

Integer Linear Programming (ILP) extends Linear Programming (LP) by restricting a subset of the variables to integer values. This formulation is used to address discrete optimization problems where certain variables must exist in binary or discrete states.

Definition 2.1.5 (ILP)**.** An *ILP* is an LP with the additional requirement that some subset of variables, $x_i, x_j, \ldots, x_k$, are constrained to integer values, i.e., $x_i, x_j, \ldots, x_k \in \mathbb{Z}$.

This restriction collapses the feasible regions into discrete subsets of Euclidean space,[5] significantly increasing the complexity of finding solutions. Traditional LP techniques cannot efficiently navigate these discrete spaces, necessitating the employment of specialized algorithms to systematically explore the discrete solution space.

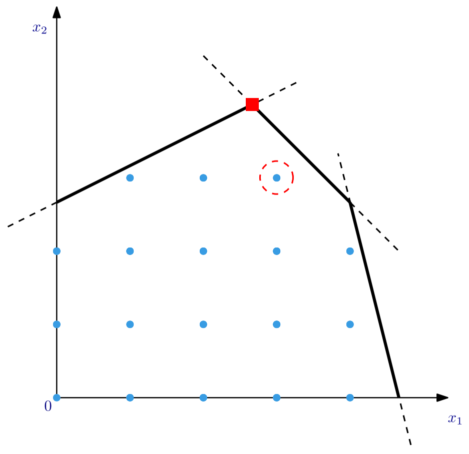


Figure 2.3: An example of how traditional LP techniques can fail in solving an ILP problem.

Figure 2.3 shows the LP problem from Figure 2.2 with the additional requirement that each variable be an integer. The feasible region is reduced to the set of blue points, among which the optimum is circled in red. Traditional LP techniques will yield the original solution at the red square, which is not integer in x_1, and thus not a feasible solution to the ILP problem.

Definition 2.1.6 (Relaxation)**.** The *relaxation* of an ILP is the LP that results from removing all integrality constraints on its variables.

This definition allows us to specify the LP problem in Figure 2.2 as the relaxation of the ILP problem in Figure 2.3.

2.1.3 Branch and Bound

The branch and bound algorithm (B&B), is a key framework developed to solve ILPs. It addresses the problem by solving a series of ILP relaxations in order to find a solution to the original ILP. This method is based on the observation that a solution to the relaxation of an ILP which happens to meet all integrality constraints is indeed an optimal solution to the ILP. This principle relies on the fact that the feasible region of relaxation always encompasses, and is larger than, the feasible region of the original ILP, as illustrated in Figure 2.3.

At a high level, B&B solves LP relaxations, yielding solutions that must be either integer feasible or violate some number of integrality constraints. Given a specific ILP problem, B&B initially solves exactly the LP relaxation of this problem, this is referred to as the root node:

[5] If all variables are integer restricted the feasible region consists of a finite set of points.

Definition 2.1.7 (Root Node)**.** The *root node* refers to the starting point of the LP relaxation, that is, the first problem of a branch and bound algorithm before any branching occurs.

Having solved the root node the B&B algorithm proceeds according to:

- **Integer feasible solutions**. If the value of the solution is less than the current incumbent (initialised to ∞), the algorithm updates the incumbent to this new solution value.[6]
- **Solutions violating integrality constraints**. Given the solution value v, and the current incumbent value z^*:

  i. If $v > z^*$, B&B *prunes* the solution, discarding this branch of the search space as it cannot contain an optimal solution.[7]

  ii. If $v \leq z^*$, B&B proceeds to *branch* by selecting a variable that violated its integer requirements and creating two subproblems. For a variable x with a non-integer value $x = a$, one subproblem imposes the constraint $x \leq \lfloor a \rfloor$, and the other $x \geq \lceil a \rceil$.[8] If either of these subproblems is rendered infeasible via the added constraint contradicting any existing constraints then this subproblem is pruned.

This recursive process of branching and pruning continues until the problem and all subproblems have been either resolved to an integer feasible solution, further branched, or pruned. In this way, B&B constructs a search tree, partitioning the solution space into progressively smaller subsets while iteratively refining the incumbent solution until all viable subproblems have been explored and the standing incumbent is deemed optimal. B&B uses this tactic to identify solutions with the added efficiency of pruning entire sections of the search space, thereby reducing computation time by avoiding regions that cannot contain an optimal solution.

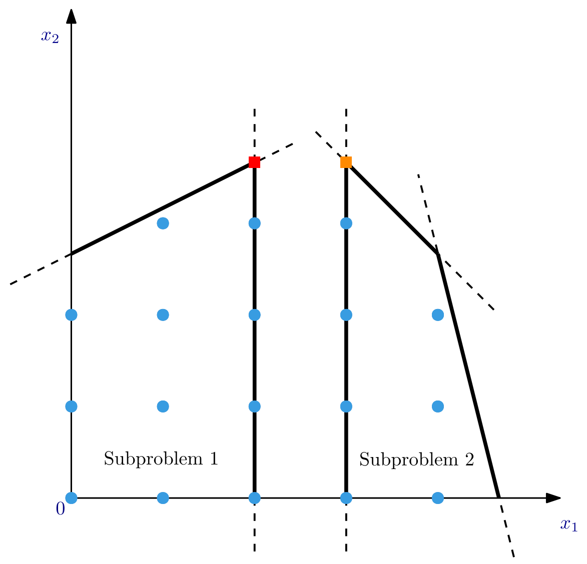


Figure 2.4: The first step of B&B applied to the ILP problem from Figure 2.3.

[6]The incumbent records the current best solution satisfying all ILP constraints.
[7]Adding constraints to an LP model cannot yield a more optimal solution.
[8]Branching collapses non-integer regions of the search space, retaining all feasible ILP solutions.

The relaxation of the problem in Figure 2.3 yields a solution with a fractional x_1, thus branching on x_1 creates the two subproblems in Figure 2.4. This step eliminates the non-integer strip wherein x_1 fell from all feasible domains while ensuring that all integer solutions remain within the set of feasible domains.

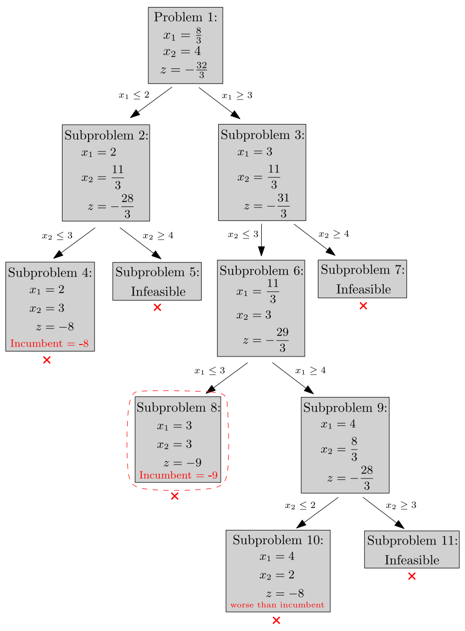

Problem 1: $x_1 = \frac{8}{3}$ $x_2 = 4$ $z = -\frac{32}{3}$
$x_1 \leq 2$
$x_1 \geq 3$
Subproblem 2: $x_1 = 2$ $x_2 = \frac{11}{3}$ $z = -\frac{28}{3}$
Subproblem 3: $x_1 = 3$ $x_2 = \frac{11}{3}$ $z = -\frac{31}{3}$
$x_2 \leq 3$
$x_2 \geq 4$
$x_2 \leq 3$
$x_2 \geq 4$
Subproblem 4: $x_1 = 2$ $x_2 = 3$ $z = -8$ Incumbent = -8
Subproblem 5: Infeasible
Subproblem 6: $x_1 = \frac{11}{3}$ $x_2 = 3$ $z = -\frac{29}{3}$
Subproblem 7: Infeasible
$x_1 \leq 3$
$x_1 \geq 4$
Subproblem 8: $x_1 = 3$ $x_2 = 3$ $z = -9$ Incumbent = -9
Subproblem 9: $x_1 = 4$ $x_2 = \frac{8}{3}$ $z = -\frac{28}{3}$
$x_2 \leq 2$
$x_2 \geq 3$
Subproblem 10: $x_1 = 4$ $x_2 = 2$ $z = -8$ worse than incumbent
Subproblem 11: Infeasible

Figure 2.5: Full B&B tree of the problem from Figure 2.3.

Advanced B&B implementations, including IBM's CPLEX, employ advanced strategies for key decisions within the algorithm. These include the selection of which variables to branch upon and the order in which to explore subproblems. Such choices are critical for the efficiency of the algorithm, as they alter the size and complexity of the search tree. For explicit details on CPLEX's implementation of B&B, please refer to IBM 2022.

To accelerate the B&B, a method of obtaining feasible solutions may be employed at any point in the process to quickly obtain integer feasible (incumbent) solutions. If these solutions improve upon the current incumbent, they can facilitate more effective pruning and lead to swifter solutions. This introduces a trade-off between the computation time for such heuristic solutions and their potential benefits at different points in the process.[9]

2.1.4 Branch and Cut

The branch and cut methodology (B&C) refines B&B by incorporating cutting-planes, equivalent to supplementary inequalities or *cuts*, which progressively narrow the feasible region of the LP relaxation, thereby accelerating the optimization process.

Definition 2.1.8 (Valid Inequlities)**.** A *valid inequality*, or cut, for a given ILP is an inequality $a_1x_1+\ldots+a_nx_n \leq b$, which is satisfied by all integer feasible solutions $x_1, \ldots, x_n$.

Figure 2.6 gives an example of a valid cut in that all integer solutions remain feasible, despite the feasible region of the relaxation being reduced. Adding such cuts before solving problems in the branch and bound tree can prevent certain non-integer solutions from occurring and may generate faster paths towards optimal solutions.

[9]Heuristic methods may fail to improve the incumbent solution, wasting computation.

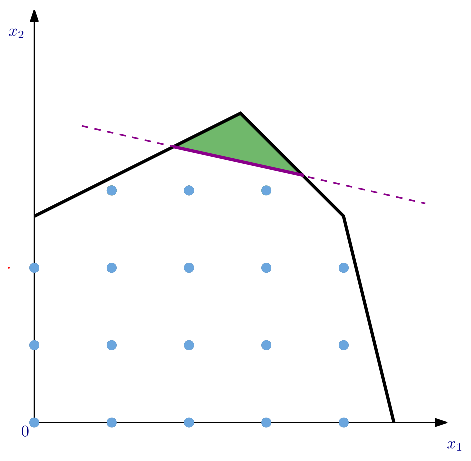


Figure 2.6: A valid cut for the ILP from Figure 2.3.

Definition 2.1.9 (Polytope Q)**.** The *polytope* Q, of a bounded feasible ILP is the convex hull of all feasible solutions to the ILP. Specifically:

$$Q = \text{conv}\{\{x = (x_1, \dots, x_n)|\ x \text{ satisfies all constraints of the ILP model}\}\}.$$

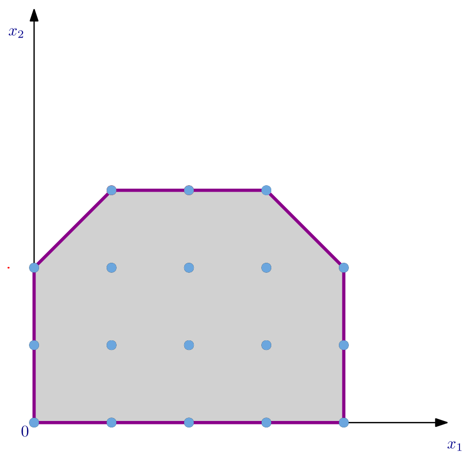


Figure 2.7: The polytope Q, of the ILP problem from figure 2.3.

By definition, Q of any bounded feasible ILP contains all integer feasible solutions to the ILP and has such solutions at each of its vertices. Thus solving the LP problem over Q will by Theorem 2.1.1 yield an integer feasible optimal solution to the ILP problem. The significance of this cannot be overstated; it implies that if Q can be precisely defined, the ILP problem can be solved as a single LP problem, thereby leveraging the computational efficiency of LP solvers.

Definition 2.1.10 (Face)**.** A face F of a polytope Q is defined as follows: for any closed half-space H that contains Q ($H \supset Q$), if the intersection of Q with the boundary of H is non-empty, this intersection constitutes a face of Q. Formally, if $Q \subset H$ and $F = Q \cap \{x_1, \dots, x_n \mid a_1x_1 + \ldots + a_nx_n = b\} \neq \emptyset$, then F is considered a face of Q.

Definition 2.1.11 (Affine Independence)**.** A set of points $S \subseteq \mathbb{R}^n$ is said to be *affinely independent* if for every point $\{x_1, \dots, x_k\} \in S$ the equalities $\sum_{i=1}^{k} \lambda_i x_i = 0$ and $\sum_{i=1}^{k} \lambda_i = 0$ imply $\lambda_i = 0$, for $i = 1, \dots, k$.

Definition 2.1.12 (Dimension of a Polytope)**.** The dimension of a polytope Q is one less than the maximum number of affinely independent points in the polytope. That is if Q can contain a maximum of k affinely independent points then $\dim(Q) = k - 1$.

Definition 2.1.13 (Facet)**.** A *facet* of an n-dimensional Q is a face F of Q that has dimension $n-1$.

In the traditional three-dimensional context, Definition 2.1.10 collectively categorizes the edges, vertices, and faces of standard polyhedra as faces, while Definition 2.1.13 identifies the traditional faces of standard polyhedra as facets.

Returning to B&C it seems apparent that the ideal cutting-planes are those that reduce the feasible region to the polytope Q. The minimal set of such cuts is exactly the set of cuts corresponding to the *facets* of Q, called the set of *facet-inducing cuts*. These cuts are optimal in that they constitute the minimal set delimiting Q, and thus provide the most efficient avenue to turn any ILP problem into an LP problem over Q. This approach offers promise but encounters limitations when applied to large-scale ILP problems. The sheer number of facets and intricate geometry of Q can make identification of all facet-inducing cuts computationally infeasible. Crucially, however, the optimal solution can be found without adding every facet defining cut.

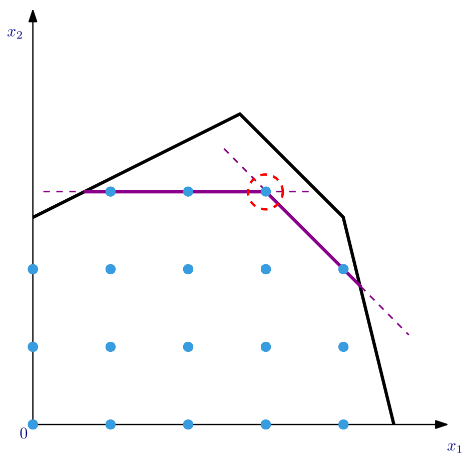


Figure 2.8: The 2 ideal cuts for the ILP problem from Figure 2.3

Adding the 2 cuts shown in Figure 2.8 leads to B&B solving the ILP at the first problem rather than requiring the 11 subproblems detailed in Figure 2.5. The cuts added are precisely the 2 facet defining cuts of the Q (Figure 2.7), violated by the solution to the original LP relaxation. This suggests efficiency gains from adding only violated cuts during B&C. Formalising this method; only cuts violated by the current solution are added, and then the resultant LP problem is solved. This process is repeated, with violated cuts added recursively until either an integer solution is reached, or a fractional solution emerges for which no additional cuts can be identified, a juncture at which B&B proceeds by branching. In this approach, only violated cuts are incorporated, significantly reducing the computational load relative to the full enumeration of the polytope's facets.

Definition 2.1.14 (Separation Problem)**.** Given an ILP with polytope $Q \subseteq \mathbb{R}^n$, whose relaxation has feasible region $P \subseteq \mathbb{R}^n$, and a solution $x^* = (x_1^*, \ldots, x_n^*) \in P$, determine if $x^* \in Q$ and if not, find a valid inequality for Q violated by x^*.

Solving the problem of separating facet defining cuts for the solutions to LP relaxations allows us to refine the feasible region, generally leading to less branching and thus smaller B&B trees. This targeted approach accelerates the convergence towards an optimal solution, as in most cases only a fraction of all facet defining cuts are required to reach an optimal solution (as demonstrated by Figure 2.8).

2.2 Graph Preliminaries

In this section we follow Wilson 2012.

Definition 2.2.1 (Undirected Graph)**.** An *undirected graph* $G = (V, E)$, consists of:

- V, a non-empty set whose elements v are *vertices* (also referred to as *nodes*),
- E, a set of unordered pairs $\{v_1, v_2\}$ of vertices whose elements are *edges*.

We may also refer to edges using notation e_i, where each $e_i \in E$ represents some distinct edge $\{v_j, v_k\} \in E$. We also say that the edge e_i is *incident* to vertices v_j and v_k.

Definition 2.2.2 (Degree of a vertex)**.** The degree of a vertex v is the number of edges incident v. That is $\deg(v) = |\{e \in E \mid v \in e\}|$.

Definition 2.2.3 (Handshaking Lemma)**.** In any graph G the sum of the degrees of all vertices is equal to exactly twice the number of edges in E. That is:

$$\sum_{v_i \in V} \deg(v_i) = 2|E|.$$

We call a graph simple, if there are a finite number of vertices, each edge is distinct and the two vertices making up any edge are distinct. Formally:

Definition 2.2.4 (Simple Graph)**.** A graph G is *simple* if V finite and $\forall e_i \in E$:

- $v_k \neq v_l$, where $e_i = \{v_k, v_l\}$;
- $e_i \neq e_j, \quad \forall e_j \in E \setminus \{e_i\}$.

Accordingly, simple graphs preclude loops and multiple edges between vertices, attributes which are not relevant to our analysis of the TSP. From this point, any graph G is assumed to be a simple undirected graph.

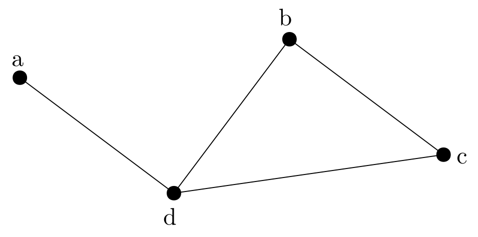


Figure 2.9: A graph G with $V = \{a, b, c, d\}$ and $E = \{\{a, d\}, \{b, d\}, \{b, c\}, \{c, d\}\}$.

Definition 2.2.5 (Complete Graph)**.** A *complete graph* is a graph G in which every two distinct vertices are connected by an edge. The complete graph on n vertices is denoted K_n, and has exactly $\binom{n}{2} = \frac{n(n-1)}{2}$ edges since each edge is a choice of 2 distinct vertices.

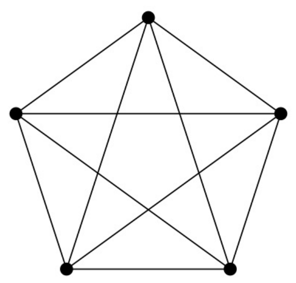


Figure 2.10: K_5

Definition 2.2.6 (Weighted Graph)**.** We define a *weighted graph* to be a graph G in which each edge $e \in E$ has been assigned a value $w(e) \in \mathbb{R}$, called its weight.

Definition 2.2.7 (Walk)**.** A *walk* in a given G is given by a sequence of vertices: $v_0, v_1, \dots, v_m$, where every pair of subsequent vertices $\{v_i, v_{i+1}\}$ represents an edge, $\{v_i, v_{i+1}\} \in E$.

Definition 2.2.8 (Cycle)**.** A *cycle* is defined as a walk in which the initial and final vertex are equal, $v_0 = v_m$, and all other vertices are distinct.

A cycle in G is said to be *Hamiltonian* if it passes through every vertex in V.[10]

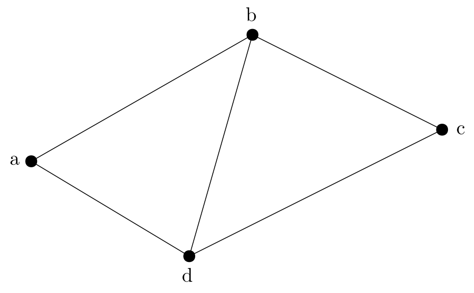


Figure 2.11: A graph G with a non-Hamiltonian cycle $a \to b \to d \to a$, and a Hamiltonian cycle $a \to b \to c \to d \to a$.

Definition 2.2.9 (Union of Graphs)**.** Given two graphs G_1, G_2 with disjoint vertex sets V_1, V_2 respectively,[11] their *union* $G_1 \cup G_2$ is the graph with vertex set $V_1 \cup V_2$ and edge set $E_1 \cup E_2$.

Definition 2.2.10 (Connected Graph)**.** A graph is *connected* if it cannot be expressed as a union of other graphs or equivalently if there exists a walk from v_i to v_j, $\forall v_i, v_j \in V$.

Every disconnected graph can be expressed as a union of connected graphs each of which is called a *connected component.*

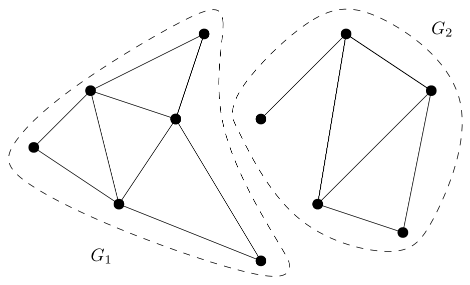


Figure 2.12: A disconnected graph G, with connected components G_1 and G_2.

Definition 2.2.11 (Edge Set of a Subset)**.** Given a set $S \subseteq V$, the set of all edges within S is denoted as $E(S)$. Specifically, this refers to the set of all edges having both ends in the subset S.

Definition 2.2.12 (Cut-set)**.** Given a graph G with $k \geq 1$ connected components, a *cut-set* $C \subseteq E$, is a set of edges such that G with the edges in C removed, has at least $k+1$ connected components.[12]

[10]The TSP is ultimately a problem of finding the minimum weight Hamiltonian cycle
[11]V_1, V_2 disjoint if $\forall v_i \in V_1, v_i \notin V_2$
[12]For a connected graph G a cut-set is equivalent to any set of edges which disconnects G.

In the context of graph theory, the term 'cut' commonly refers to a cut-set, distinct from the 'cut' defined by a cutting plane in optimization contexts (see Section 2.1.4).

Definition 2.2.13 (Cut-set of a Subset)**.** Given a set $S \subseteq V$, the set of edges having exactly one end in S is denoted $\delta(S)$.[13]

Definition 2.2.14 (Weight of A Cut)**.** In a weighted graph G, the *weight of a cut* $C \subseteq E$ is the sum of the weights of its constituent edges, given by $w(C) = \sum_{e \in C} w(e)$.

Definition 2.2.15 (Minimum Cut)**.** In a weighted graph G, *a minimum cut* is a cut-set $C \subseteq E$ of minimum weight. That is, $\forall C' \subseteq E$, $\sum_{e \in C} w(e) \leq \sum_{e \in C'} w(e)$.

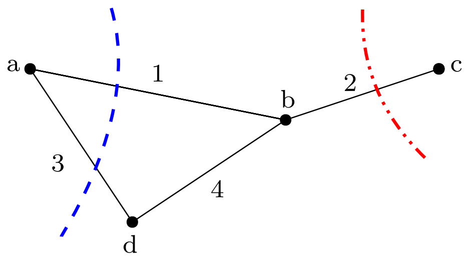


Figure 2.13: A weighted graph G, with a cut depicted in blue and a minimum cut depicted in red.

Definition 2.2.16 (Minimum s-t cut)**.** Given a weighted graph G and two distinct vertices $s, t \in V$, *a minimum s-t cut* is a minimum cut $C \subseteq E$ amongst all cuts that partition V such that s and t are separated into disjoint subsets of vertices. That is, a minimum s-t cut is a minimum cut that disconnects s and t.

The cut indicated in blue in Figure 2.13 is a minimum a-b cut (it is in fact the only such minimum cut).

Definition 2.2.17 (Most Tightly Connected)**.** In weighted graph G given a nonempty set of vertices $A \subset V$, the vertex adjacent to A with the greatest weight is called the *most tightly connected* vertex to A. Specifically the weight is the sum of all edges between the vertex and a vertex within the set A: Thus we say $z \in V \backslash A$ is the most tightly connected vertex to A if $w(A, z) = \max\{w(A, y) \mid y \in V \backslash A\}$.[14]

Minimum cut algorithm (the Stoer-Wagner algorithm)

To find the global minimum cut of the graph, we introduce the Stoer-Wagner Algorithm, which is a method focused on undirected but non-negative weighted graphs. It uses vertex contraction to shrink graphs. The Stoer-Wagner algorithm consists of two components, one is to find the local minimum cut in each phase, and the other is to determine whether the local minimum cut is a global minimum cut.

For the first problem, the strategy used by Stoer and Wagner 1997 is the Maximum Adjacency Search, whose main idea is similar to the Maximum Spanning Tree. First, we construct a non-empty subset A, starting from containing one arbitrary vertex, here we choose a. We are interested in finding the vertex that is most tightly connected with A since the subset A including the edges with a larger weight implies that the edges with smaller weights could be cut. We consider the adjacency as the sum of edge weights in this case. The subset is continuously being expanded until $A = V$.

[13]If $\delta(S) \neq \emptyset$, then $\delta(S)$ forms a cut.

[14]Here $w(S, v)$ denotes the sum of the weights of the edges between elements of the set S and vertex v

By considering graph (a) in Figure 2.14, we find that the adjacent vertices connected to the subset A are b and d, where $w_{(a,b)} > w_{(a,d)}$, so we should add b to the subset A. At this point, the subset A contains multiple vertices, here we can consider the growing subset A as a super node, and then we require a formal definition for the weight of the edges to the super node.

Definition 2.2.18 (Super Node Edge Weight)**.** The *weight of the edges to the super node* is the sum of the weights of all edges coming from vertices outside A and connecting to each individual vertex within the set A.

Hence, we ignore the edges between the internal vertices of the super node, and only consider the edges between the super node and the other vertices $V \setminus \{i \mid i \in V\}$ outside the set A. In Graph (c) as shown in Figure 2.14, the weight of the edge connecting vertex a and vertex d is equal to 2, and that of connecting vertex c and vertex d is equal to 2. Obviously, both a and c are included within the super node, and they are all pointing to the external vertex d, thus the current graph can be updated by Definition 2.2.18, where the weight of the edge from the super node to vertex d becomes 4. We then repeat the above process until all vertices have been considered as shown in Figure 2.14. The edges that divide the last pair of vertices (s,t) being added to the super node into two disjoint sets, define the minimum s-t cut for the current graph, often referred to as the local minimum cut. In the 1st phase (or iteration), the minimum s-t cut is 3.

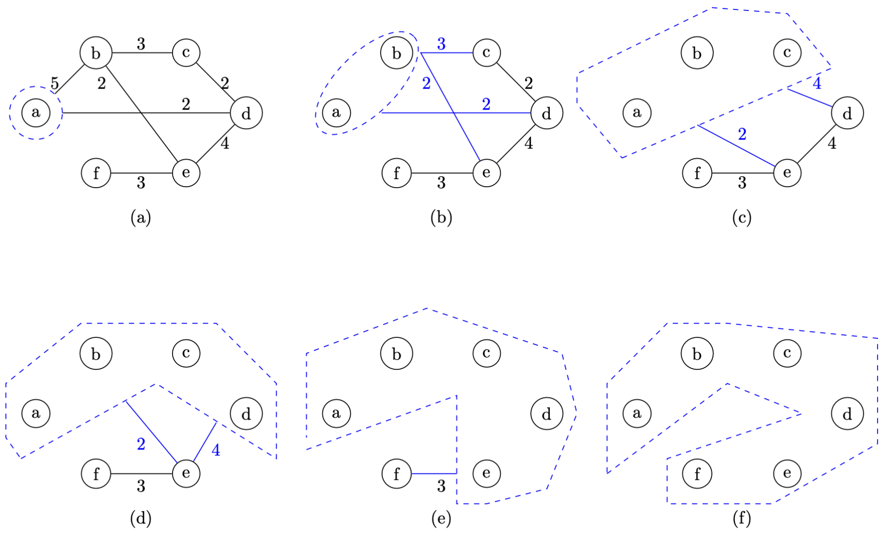


Figure 2.14: Graph $G_1(V,E)$ in the 1st iteration by Maximum Adjacency Search.

For the second problem mentioned before, we need to figure out the minimum s-t cut of which subgraph is the global minimum. After each phase, the graph can be simplified by merging the last pair of vertices (s,t) obtained by Maximum Adjacency Search into a new vertex, which can also be similarly considered as another super node to shrink the graph. Then the algorithm will proceed to the simplified graph with $|V|-1$ vertices.

Observing Graph (e) and (f) of Figure 2.14, we know that the last pair of vertices (s,t) is represented by (e,f) in the current graph, forming a local minimum s-t cut. To prepare the simplified graph of the next phase, we delete the edges between vertices e and f, update the weight by Definition 2.2.18, and reconnect the merged vertex ef to other vertices. We then implement the Maximum Adjacency Search on the updated graph again to obtain a new minimum s-t cut, that is, the sum of the weights of all edges dividing vertices d and ef, is equal to 6. As illustrated in Figure 2.15, we repeat the above process until $|V| = 1$.

Each iteration will dynamically generate a new minimum s-t cut, leading to a graph with a smaller size. The minimum$s$-t cut in each phase is called *the cut of the phase*. The global minimum cut is then the smallest one among all the cut-of-the-phase, which is equal to 3 for our example according to Table 2.1.

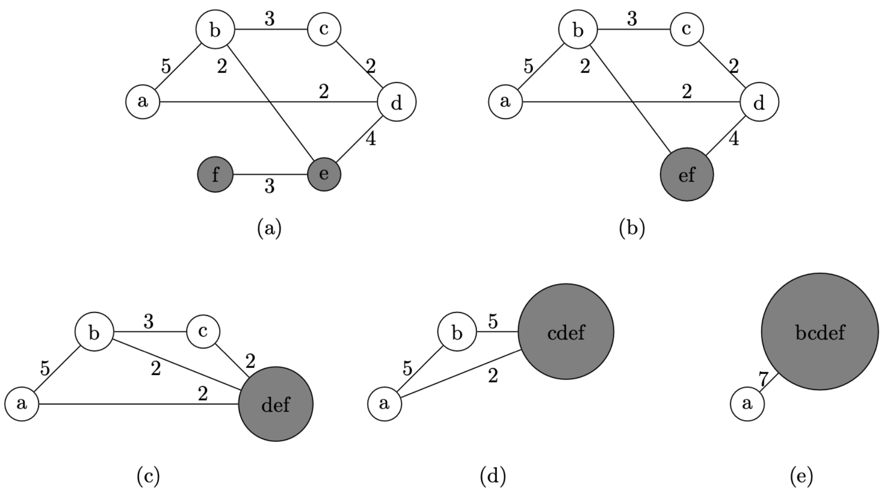


Figure 2.15: Simplified Graph $G_1(V, E)$ in each phase.

Table 2.1: The last pair of vertices (s, t) in each phase and their corresponding cut-of-the-phase

s	t	cut-of-the-phase
e	f	3
d	fe	6
c	def	6
b	$cdef$	7
a	$bcdef$	7

In general, the Stoer-Wagner algorithm is an exact approach for solving minimum cut problems efficiently, which can always find the optimal solution. The pseudocode below provides implementation details of this process.

Algorithm 1 Stoer-Wagner Algorithm

```
1: Input: $G, w, a$
2: Init: Super node $A = \{a\}$; current-minimum-cut = $\infty$;
3: while $|V| > 1$ do
4:     while $A \neq V$ do
5:         add the most tightly connected vertex to $A$
6:         merge the last pair of vertices being added
7:     end while
8:     cut-of-the-phase = weight of the final super node and the last vertex
9:     if cut-of-the-phase < current-minimum-cut then
10:        current-minimum-cut = cut-of-the-phase
11:    end if
12: end while
13: return current-minimum-cut
```

The time per iteration we are using the Maximum Adjacency Search is $O(|E| + |V| \log |V|)$, Since we need to merge the last two vertices added in the growing set at the end of each iteration, there are $|V| - 1$ iterations. Then multiplying them we get the total running time of $O(|V||E| + |V|^2 \log |V|)$.

Chapter 3

The TSP

The traveling salesman problem (TSP) asks a deceptively simple question; given a set of vertices and the distances between each pair, what is the shortest route that visits each vertex exactly once and returns to the origin? We state this more precisely as:

Find the shortest Hamiltonian cycle in a given complete weighted graph.

This problem is broadly categorized into the symmetric and the asymmetric TSP. The symmetric TSP (STSP) addresses scenarios where the distance between any two vertices remains the same, irrespective of the travel direction. Conversely, the asymmetric TSP (ATSP) allows for the possibility that traveling from A to B may differ in length to traveling from B to A, reflecting scenarios affected by directional factors. This paper deals exclusively with the STSP, focusing on solutions within undirected graphs, wherein the distance between any two vertices is constant.

3.1 Formulation

In this section, we base our formulation on the approach introduced by Dantzig, Fulkerson, and S. M. Johnson 1954 in their paper *Solution of a large-scale traveling salesman problem*.

The symmetric traveling salesman problem, henceforth the **TSP**, is defined on a simple undirected graph $G = K^n = (V, E)$,[1] wherein:

- n is the number of vertices (referred to as *nodes*) in V.
- $\frac{n(n-1)}{2}$ is the number of edges in E (by Definition 2.2.5 as G complete).
- Each edge $e \in E$ has a weight c_e, referred to as it's *cost*, encapsulating the notion of length or traversal time.

We refer to any Hamiltonian cycle within G as a *tour*, allowing us to restate the objective of the TSP as identifying the minimum cost tour in a given G. In order to formulate this problem as an integer linear programming (ILP) problem we introduce binary decision variables for each edge. Specifically we define:

$$x_e = \begin{cases} 1, & \text{if } e \text{ is included in the tour,} \\ 0, & \text{otherwise.} \end{cases} \tag{3.1}$$

These variables provide a vital interface to mathematically represent whether or not an edge is part of a tour. Finally, we note that this paper focuses exclusively on TSP instances with $n \geq 6$. This is due to many of the definitions and concepts explored being undefined for some $n < 6$ as well as the relative triviality of such instances.

[1]Defining the TSP on incomplete graphs is equivalent to setting missing edge weights to ∞.

3.1.1 Objective Function

We define the objective function to be minimised as the sum of all edge costs multiplied by their respective binary variables:

$$\min \quad \sum_{e \in E} c_e x_e. \tag{3.2}$$

Given a tour T, for any edge $e \in E$, 3.1 implies:

$$c_e x_e = \begin{cases} c_e, & \text{if } e \in T; \\ 0, & \text{otherwise.} \end{cases}$$

In this way the objective function sums the cost of only the edges included in the tour T, measuring the total tour cost. Thus the minimisation in 3.2 perfectly represents the objective of the TSP: to find the minimum cost tour.

3.1.2 Degree Constraints

$$\sum_{e \in \delta(v)} x_e = 2, \quad \forall v \in V \tag{3.3}$$

This set of $|V| = n$ constraints ensures that, for every vertex $v \in V$, the sum of the binary decision variables corresponding to each edge incident to v equals exactly 2. To provide an intuition for what this implies recall:

- $\delta(v)$ is the set of edges with exactly one end in v, or equivalently the edges *incident* to v (see Definition 2.2.13);
- Each variable x_e equals either 0 or 1 (see 3.1).

This allows us to interpret the degree constraints as enforcing that for every vertex $v \in V$ exactly two of the edges incident to v have their corresponding $x_e = 1$, or rather that there be exactly 2 edges incident to every vertex. This is a crucial requirement of Hamiltonian cycles as it ultimately ensures that every vertex is visited in the tour and that no vertex is visited more than once.

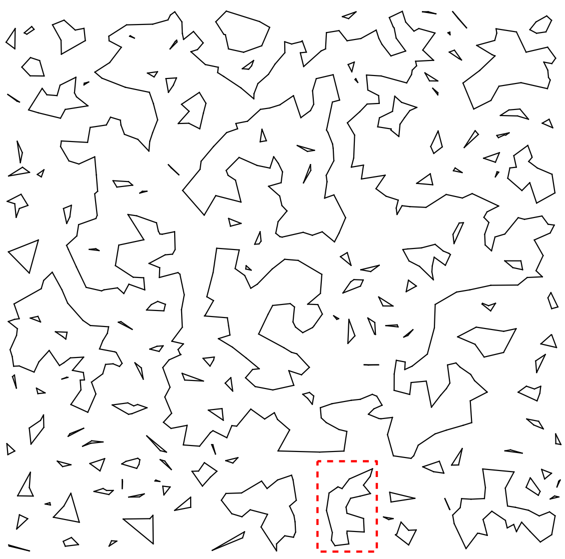


Figure 3.1: Optimal solution to a 1000 node, random, euclidean TSP instance with only degree & integer constraints.

3.1.3 Subtour Elimination Constraints

Motivation

Definition 3.1.1 (Subtour)**.** In a graph G a *subtour* is any cycle that does not visit all vertices in G, specifically any cycle visiting $k < n$ vertices in G.

Solving the ILP formulated thus far, i.e. minimising the objective function (3.2) subject to the degree and integrality constraints (3.3 & 3.1), yields solutions akin to Figure 3.1. In such solutions all mentioned constraints are satisfied, yet clearly the solution does not form a Hamiltonian cycle. This is due to the presence of *subtours*, an example of which is contained within the red border. To motivate the formulation of a set of constraints to eliminate subtours we observe a primary feature of Hamiltonian cycles. For any non-empty subset of vertices $S \subset V$ of a graph G whose edges form a tour, there must be at least 2 edges leaving S, formally: $|\delta(S)| \geq 2$ (in fact this cardinality must also be even). To understand this we use the following somewhat redundant equivalences about sets and their complements:

$$S \neq \emptyset \iff V \backslash S \neq V,$$

$$S \neq V \iff V \backslash S \neq \emptyset.$$

These equivalences ultimately imply that if $S \subset V$ is proper and non-empty, $V \backslash S$ must inherit these classifications. This means that there will always be vertices in $V \backslash S$, and as a tour must visit all vertices it is clear that at some point an edge must "leave" S to reach the set $V \backslash S$ and another edge must "return" to S in order to complete the cycle (in fact this "leaving" and "returning" can happen a positive real number of times; with each iteration adding exactly 2 edges to $\delta(S)$ forcing even cardinality). Clearly, the set of vertices within the red border in Figure 3.1 violates this requirement as there are no edges crossing the border.

Initial formulation

Thus to preclude subtours we introduce the following set of subtour elimination constraints (**SECs**):

$$\sum_{e \in \delta(S)} x_e \geq 2, \ \forall S \subset V, \text{ where } S \neq \emptyset. \tag{3.4}$$

Technically these constraints enforce that for every non-empty proper subset of vertices $S \subset V$, the sum of the variables x_e corresponding to the set edges with exactly one end in S, must be at least 2. To intuit explicitly how it is that this prevents subtours we highlight that every subtour of length k is exactly a set S of k vertices with no edges leaving S. Thus adding the SEC specified in 3.4 to S ensures that $\delta(S) \neq \emptyset$ meaning there can no longer exist a subtour of length $|S|$ in this set. In this way, if we add the SEC for all proper non-empty subsets we prevent all subtours of all lengths ($k < n$) from existing.

In theory, this formulation precludes every possible subtour in a graph G; however, it necessitates the addition of a constraint for every proper non-empty subset $S \subset V$. Given that a subset of V represents a binary choice of inclusion for n elements, it can be deduced that there are 2^n possible subsets of V, one of which is V itself and another is the empty set $\emptyset$. Consequently, there are exactly $2^n - 2$ proper non-empty subsets of V, each of which requires an SEC according to this formulation. The exponential number of constraints here motivates the pursuit of identifying a set of SECs, of minimal cardinality, which collectively precludes the existence of all subtours — a set we will refer to as a *non-redundant* set of SECs.

Non-redundant SEC domain

We proceed to give the classification of a non-redundant set of SECs which prevails in the literature (see Martin Grötschel and M. Padberg 1979).

First note that adding the SEC for any set $S \subset V$ is equivalent to adding it to the set $V \backslash S$. This follows from the equality between edge sets $\delta(S) = \delta(V \backslash S)$, with both denoting the collection of edges spanning the partition of V into S and $V \backslash S$.[2] In order to account for this redundancy we define a new set which includes only one of S or $V \backslash S$:

$$\mathcal{NR} = \{S \subset V \mid \; S \in \mathcal{NR} \iff (V \backslash S \notin \mathcal{NR} \text{ and } 1 \leq |S| \leq n-1)\}.$$

This definition ensures that exactly one of S or $V \backslash S$ is included for any proper non-empty $S \subset V$ (S proper and nonempty enforced by $1 \leq |S| \leq n-1$). Thus using $\mathcal{NR}$ as the domain over which the SECs are added provides a succinct way to mitigate the redundancy of S and $V \backslash S$ defining the same constraint.

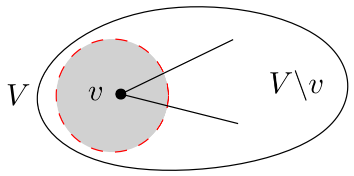


Figure 3.2: Degree constraints enforce $\delta(S) = 2$ for any set S where $|S| = 1$.

Furthermore, note that the SECs are redundant for any set $S \subset V$ where $|S| = 1$. The degree constraints in 3.3 ensure that for every vertex $v \in V$ there are exactly 2 edges in the set $\delta(v)$. Thus if any v is selected as the sole member of the set S, the SEC enforces: $\sum_{e \in \delta(v)} x_e \geq 2$, and the degree constraint enforces: $\sum_{e \in \delta(v)} x_e = 2$. Clearly there are never violated SECs for sets $S = \{v\}$ and thus we can disregard these SECs. Given this redundancy holds $\forall S$ where $|S| = 1$, and considering that adding the SEC for S is equivalent to adding the SEC for $V \backslash S$, the redundancy must also be true $\forall V \backslash S$, or rather $\forall S$ where $|S| = n-1$. This allows us to refine the sets constituting $\mathcal{NR}$ to those with cardinality between 2 and $n-2$ as follows:

$$\mathcal{NR} = \{S \subset V \mid \; S \in \mathcal{NR} \iff (V \backslash S \notin \mathcal{NR} \text{ and } 2 \leq |S| \leq n-2)\}.$$

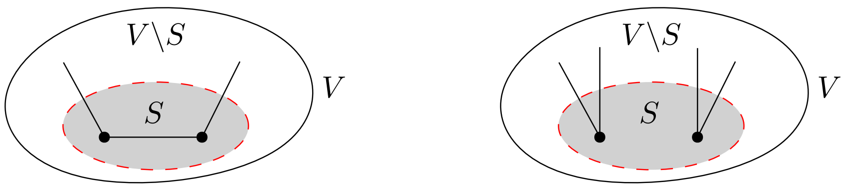
This completes the classification of the non-redundant domain for SECs generally identified in the literature. Of particular interest is the cardinality of $\mathcal{NR}$ as it represents the minimal number of SECs needed to preclude all subtours in a graph on n vertices. To enumerate this cardinality we notice that we have removed all sets of size 1 of which there are n, as well as all sets of size $n-1$ of which there are also n. Furthermore, the number of sets considered has been halved by the inclusion of exactly one of S and $V \backslash S$. Thus the cardinality is given by:

$$|\mathcal{NR}| = \frac{2^n - 2 - 2n}{2}. \tag{3.5}$$

We see that despite the refinements the cardinality of the minimal domain for SECs is still exponential in n suggesting that enumerating all SECs within the ILP model of a TSP instance becomes rapidly infeasible as n increases.

We now propose a refinement on the classification of the non-redundant domain for SECs of Symmetric TSP instances. We focus on the SECs for sets $S \subset V$ where $|S| = 2$. Without loss of generality, we denote the two vertices in any such set by $S = \{v_i, v_j\}$. Notice that for any $v_i, v_j \in S$, there either exists an edge connecting these vertices, or

[2] An edge has exactly one end in $S \iff$ the edge has exactly one end in $V \backslash S$.



(a) $v_i, v_j \in S$ share an edge.

(b) No edge between $v_i, v_j \in S$.

Figure 3.3: The two possibilities of edge structure for any set $S \subset V$ where $|S| = 2$.

there does not, as indicated in Figure 3.3. In the case that the vertices share an edge (3.3a), the degree constraints enforce that each vertex has exactly 1 other incident edge, which must be incident to a vertex outside of the set S as the only possible edge within S has already been used. In this case, there will be exactly 2 edges in the set $\delta(S)$, satisfying the SEC for this set. In the case that the vertices do not share an edge (3.3b), the degree constraints force that each vertex has exactly 2 edges leaving the set, resulting in there being exactly 4 edges in the set $\delta(S)$, again satisfying the SEC of this set. Thus for all cases where $|S| = 2$ the SEC is redundant, and as before we can extend this result to the sets $V\backslash S$, specifically all sets where $|S| = n - 2$.[3] Incorporating this further refinement we define a new set $\mathcal{NR}'$ to exclude sets of size 2 and $n - 2$ as follows:

$$\mathcal{NR}' = \{S \subset V \mid \; S \in \mathcal{NR}' \iff (V\backslash S \notin \mathcal{NR}' \text{ and } 3 \leq |S| \leq n-3)\}. \tag{3.6}$$

To enumerate the cardinality of $\mathcal{NR}'$ we notice that we have removed from $\mathcal{NR}$ all sets of size 2 and $n - 2$. Choosing sets of size 2 there are n choices for the first element and $(n-1)$ choices for the second element with each pair counted twice. Thus there are $\frac{n(n-1)}{2}$ distinct sets of size 2 or equivalently of size $n-2$ (number of sets of size $n-2$ is equivalent to the number of ways of selecting 2 elements to exclude). Since each pair of these sets was counted only once in 3.5 we obtain the cardinality:

$$|\mathcal{NR}'| = \frac{2^n - 2n - 2}{2} - \frac{n(n-1)}{2} = \frac{2^n - n^2 - n - 2}{2}. \tag{3.7}$$

This improves upon $|\mathcal{NR}|$ by $\frac{n(n-1)}{2}$, a term exhibiting polynomial growth in comparison to the exponential growth of 2^n. The result is that any meaningful reduction in the number of non-redundant SECS fades rapidly as n grows.

Table 3.1: The growth of $|\mathcal{NR}|$ and $|\mathcal{NR}'|$ in n.

n	$\lvert\mathcal{NR}\rvert$	$\lvert\mathcal{NR}'\rvert$
6	25	10
7	56	35
8	119	91
9	246	210
10	501	456
15	16368	16263
20	524267	524077
30	0.5×10^9	0.5×10^9

Ultimately the minimum number of SECs needed to preclude all subtours, given by 3.7, is still exponential in n. Thus adding all SECs to a model definitively becomes infeasible

[3]This result does not extend to the asymmetric case as v_i, v_j may have 2 edges between them.

as n increases (for $n = 50$ there are more than 10^{15} SECs). Managing this exponential number of constraints will be discussed in Section 3.3.2.

3.1.4 Full ILP formulation

We consolidate this section by presenting a full ILP formulation for the TSP, integrating the objective function, degree constraints, refined SECs and integer restrictions on each of x_e. Specifically for a simple complete undirected $G = (V, E)$ on n vertices in which each edge $e \in E$ has weight c_e and the set $\mathcal{NR}$ is defined according to 3.6; the following model is an ILP formulation of the TSP:

$$\begin{aligned}
\min \quad & \sum_{e \in E} c_e x_e && \text{(ILP.1)}\\
\text{s.t.} \quad & \sum_{e \in \delta(v)} x_e = 2, \;\; \forall v \in V; && \text{(ILP.2)}\\
& \sum_{e \in \delta(S)} x_e \geq 2, \;\; \forall S \in \mathcal{NR}; && \text{(ILP.3)}\\
& x_e \in \{0, 1\}, \quad \forall e \in E. && \text{(ILP.4)}
\end{aligned}$$

This model encapsulates all essential elements of the TSP within an optimization framework, minimizing the total tour cost while ensuring each vertex is visited exactly once and all subtours are eliminated.

3.1.5 Comb Inequalities

We now introduce a class of valid inequalities for the TSP known as the *general comb inequalities* first formulated in this respect in Martin Grötschel and M. Padberg 1979 as a generalisation of the comb inequality presented by Chvátal 1973 which was ultimately a generalisation of the 2-matching constraints formulated in Edmonds 1965. We first elucidate the concept of a comb within a graph:

Definition 3.1.2 (Comb)**.** A *comb* is defined as a structure within a graph G consisting of a handle $H \subset V$ associated with a set of teeth $T_j \subset V, j = 1, \dots, t$, satisfying:

1. $|T_j \cap H| \geq 1, \quad j = 1, \dots, t;$
2. $|T_j \backslash H| \geq 1, \quad j = 1, \dots, t;$
3. $T_i \cap T_j = \emptyset, \quad 1 \leq i < j \leq t;$
4. $t \geq 3$ and odd.

Less formally this definition requires that a comb consist of some handle of vertices with an odd number > 3 of teeth in which each tooth is vertex disjoint, all teeth share at least one vertex with the handle and every tooth has at least one vertex that is not in the handle. We now state the comb inequality according to (A. N. Letchford and Lodi 2002), and then proceed to give a loose derivation.

Definition 3.1.3 (Comb Inequality)**.** Given a comb with handle H and teeth T_j, $j = 1, \dots, t$, the *Comb Inequality* is given by:

$$\sum_{e \in E(H)} x_e + \sum_{j=1}^{t} \sum_{e \in E(T_j)} x_e \leq |H| + \sum_{j=1}^{t} |T_j| - \left\lceil \frac{3t}{2} \right\rceil .$$

The comb inequality ultimately provides an upper bound on the sum of the number of edges within the handle and each of the teeth of any comb. Thus we consider the maximal edge structure of a comb to facilitate a loose derivation of this inequality. First we notice that a handle H can have at most $|H|-1$ edges within the set $E(H)$ (as otherwise it would contain a subtour), and equivalently each tooth T_j can have at most $|T_j|-1$ edges within the set $E(T_j)$. This provides the initial bound:

$$\sum_{e\in E(H)} x_e + \sum_{j=1}^{t} \sum_{e\in E(T_j)} x_e \leq |H| - 1 + \sum_{j=1}^{t}(|T_j|-1) = |H| + \sum_{j=1}^{t}|T_j| - 1 - t. \qquad (3.8)$$

We now investigate the case of equality in this bound for $t = 3$.

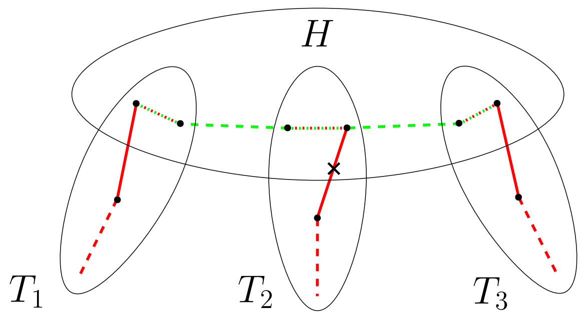


Figure 3.4: A maximal edge structure for a general comb with 3 teeth.

Figure 3.4 demonstrates the case of equality for $t = 3$, wherein without loss of generality, each tooth has exactly $|T_j|-1$ edges (drawn in red) and the handle H has exactly $|H|-1$ edges (drawn in green). For generality, the dashed edges represent paths with some non-negative integer of vertices on them while dotted edges (i.e. the red & green edges) inherit this definition with the added possibility that the two vertices they join are in fact the same vertex (this allows for the case of $|T_j \cap H| = 1$). Clearly there is a violation in that the vertex circled in purple has degree 3. A minimal edge removal required to amend this violation of the degree constraints comes from removing the solid red edge in T_2. The result is that the maximum edge sum in this comb is 1 less than the bound in Equation 3.8 enforces. To formalize this pattern we investigate the general case of $t = 2k+1$ for some $k \in \mathbb{N}$.

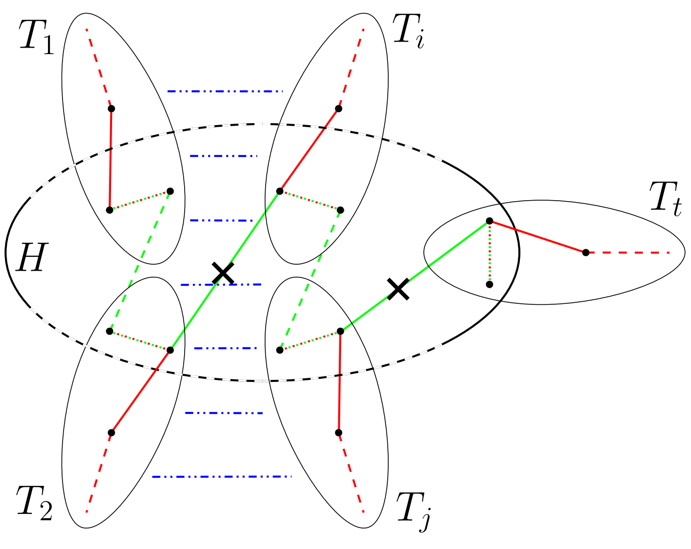


Figure 3.5: A maximal edge structure for a general comb with $t = 2k+1$ teeth.

Figure 3.5 displays a generalised comb adhering to the same conventions as Figure 3.4, with the addition of the blue dashed section representing a space wherein some non-negative number of pairs of teeth exist. Additionally, the comb is structured such that all vertices of degree 3 (of which there must be exactly $2k$ in any comb with these edge requirements), are directly connected to another vertex of degree 3. It is precisely this requirement that allows us to identify a minimum possible edge removal in such a graph, as removing these edges amends 2 degree violations (more than this is not possible as edges are only incident to 2 vertices). This supports the claim that this figure constitutes a generalisation of the upper bound on the number of edges any comb can contain. We can see in this case that all teeth except T_1 contain a vertex of degree 3. A minimal edge removal required to amend this violation corresponds to the removal of every green solid edge, of which there is one for every pair of teeth. Thus the number of edge removals required is exactly $\lfloor \frac{t}{2} \rfloor$ (recalling that t odd). This allows us to tighten the bound given in 3.8 to:

$$\sum_{e\in E(H)} x_e + \sum_{j=1}^{t} \sum_{e\in E(T_j)} x_e \leq |H| + \sum_{j=1}^{t} |T_j| - 1 - t - \left\lfloor \frac{t}{2} \right\rfloor . \tag{3.9}$$

Notice that since t is integer we have:

$$-t - \left\lfloor \frac{t}{2} \right\rfloor = - \left\lfloor \frac{3t}{2} \right\rfloor .$$

Additionally since t is odd we state the equality

$$- \left\lfloor \frac{3t}{2} \right\rfloor = - \left\lceil \frac{3t}{2} \right\rceil + 1.$$

Ultimately this allows us to rewrite the bound in 3.9 as:

$$\sum_{e\in E(H)} x_e + \sum_{j=1}^{t} \sum_{e\in E(T_j)} x_e \leq |H| + \sum_{j=1}^{t} |T_j| - 1 \left\lfloor \frac{3t}{2} \right\rfloor = |H| + \sum_{j=1}^{t} |T_j| - \left\lceil \frac{3t}{2} \right\rceil \tag{3.10}$$

This completes our loose derivation of the comb inequality.

3.2 Heuristic Solutions

Heuristic algorithms are methods designed to quickly find good, feasible, but not necessarily optimal, solutions to the TSP. In situations where obtaining exact solutions is computationally expensive or infeasible, heuristics serve as the only method of obtaining good feasible solutions rendering them invaluable. More pertinent to our analysis of the TSP however, heuristics are able to contribute valuable upper bounds that enhance the performance of exact solution frameworks, such as branch and cut, introduced in section 2.1.4 and detailed in Section 3.3.

In this Section, we detail several commonly used heuristics based on the foundational work presented by D. Johnson and McGeoch 1995 and Hwang, Alidaee, and J. D. Johnson 1999 in their comprehensive study for the STSP. We begin with the discussion on construction algorithms, which construct a heuristics solution tour from scratch.

Definition 3.2.1 (Construction Heuristic)**.** A *construction heuristic* for the TSP is an algorithm that constructs a feasible solution by starting with an empty solution and iteratively adding cities to the tour according to a specific criterion, until all cities are included. The goal of construction heuristics is to keep the total distance as short as possible at each step.

Definition 3.2.2 (Improvement Heuristic)**.** An *improvement heuristic* for the TSP is an algorithm that iteratively enhances the quality of an existing solution by making small modifications that reduce the total tour length, often through the exchange of edges or rearranging parts of the tour.

3.2.1 Greedy Algorithm

Greedy Algorithm, just like its name, embodies the principle of being 'greedy' and choosing the option that appears the best at the moment. It initiates the tour by connecting two vertices with the lowest associated cost, typically the shortest distance. The process continues by sequentially choosing edges with the next lowest costs until all vertices are connected, forming a tour. This algorithm prioritizes immediate cost savings, potentially overlooking long-term efficiencies.

Algorithm 2 Greedy Algorithm

```
1: Sort edges by distance in ascending order
2: Initialize $visited = [\,]$; $E$[vertex] be the number of edges connected to each vertex
3: while $|visited| <$ the total number of cities -1 do
4:     for each edge $e$ in sorted edges do
5:         Let $u$ and $v$ be the vertices connected by $e$
6:         if adding $e$ does not increase $E[u]$ or $E[v]$ above 2 and does not form a
           premature cycle then
7:             Add $e$ to $visited$ and increment $E[u]$ and $E[v]$
8:         end if
9:     end for
10: end while
```

3.2.2 Nearest Neighbor

The Nearest Neighbor Algorithm is a variant of the Greedy Algorithm that emphasizes proximity. Starting from an arbitrary vertex, it continuously connects to the nearest unvisited vertex, forming a singular path. This approach is straightforward and efficient, but it may lead to suboptimal tours if the path encounters 'dead ends' or areas of high density late in the sequence.

Algorithm 3 Nearest Neighbor Algorithm

```
1: Initialize the starting point $start$
2: Initialize $tour = [start]$, $visited = \{start\}$, $current = start$
3: while $|visited| <$ the total number of cities do
4:     $nearest = \text{argmin}\{\text{dist\_mat}[current][i] : i \notin visited\}$
5:     $tour.append(nearest)$, $visited.add(nearest)$, $current = nearest$
6: end while
7: Close the tour by connecting the last and first cities if not already connected
```

Here we introduce an altered version of the Nearest Neighbour heuristic for scenarios where we are given a partial solution and an incomplete set of edges. Initially, we first count how many times each node has been visited within the partial solution, identifying those unsaturated nodes with only one connection. We then locate the nearest unconnected node to each unsaturated node, merge the components containing these two nodes, and connect the edge between these two nodes. This iterative procedure persists until all unsaturated nodes are integrated, resulting in the formation of a complete tour path.

3.2.3 Nearest Insertion

Unlike previous algorithms, the Nearest Insertion algorithm starts with a pair of connected vertices. Then for each iteration, it evaluates all unconnected vertices and inserts the one closest to any vertex in the existing tour, ensuring the additional cost is minimized. This iterative process expands the tour until all vertices are included, aiming for local optimality at each insertion.

3.2.4 Cheapest Insertion

Similar to Nearest Insertion but with a focus on overall cost, the Cheapest Insertion Algorithm also starts with an initial connected edge with two points. Instead of only considering those nearest points, it assesses the insertion cost for every vertex at every possible position along the existing tour, selecting the insertion that yields the lowest additional expense. This method strives for a balance between tour proximity and cost efficiency.

3.2.5 Random Insertion

The Random insertion initiates with two connected vertices and keeps adding the remaining vertices at random positions within the existing tour. Although this approach may seem unsystematic, it introduces variability that can possibly uncover unexpectedly efficient tours, especially when used together with improvement algorithms.

3.2.6 Farthest Insertion

The Farthest insertion begins with a single vertex and prioritizes the inclusion of vertices that are farthest from any point in the current tour. By evaluating the additional cost of inserting each candidate vertex, it chooses the one that minimizes the increase in tour cost. This method is particularly useful for spreading the tour evenly across the problem space and avoiding clustering.

After constructing a heuristics solution tour, we can further refine our solution by implementing improvement algorithms. Below, we focus on those improvement algorithms based on edge exchange. Typically referred to as k-opt techniques, these methods involve the substitution of k edges with a different set of k edges during each step of the process. Two commonly used strategies 2-opt and 3-opt algorithms are discussed here.

3.2.7 2-opt Algorithm

The 2-opt algorithm is a simple yet effective local search method used to improve the solution of the TSP problem. The algorithm's core principle is to iteratively remove two edges from the tour and then reconnect the two resulting paths in a different way that reduces the overall tour length (See Figure 3.6 for an illustration). We keep swapping until we can no longer improve the tour. This process is repeated for every possible pair of edges in the tour, with the aim of reducing the tour's total cost.

Algorithm 4 Nearest Neighbor Algorithm with Partial Edges

```
1: Input: $partial\_edges$, $connected\_comps$
2: Initialize $unsaturated \leftarrow []$; $tour\_edges \leftarrow partial\_edges$
3: Initialize $visited\_count[i] \leftarrow 0$ for each vertex $i$ in graph
4: for each edge $(i, j)$ in $partial\_edges$ do
5:     $visited\_count[i] \leftarrow visited\_count[i] + 1$
6:     $visited\_count[j] \leftarrow visited\_count[j] + 1$
7: end for
8: for each vertex $i$ do
9:     if $visited\_count[i] = 1$ then
10:        Append $i$ to $unsaturated$
11:    end if
12: end for
13: while length of $unsaturated > 2$ do
14:    $cur\_node \leftarrow unsaturated[0]$
15:    Remove the first element from $unsaturated$
16:    Initialize $options \leftarrow []$; $current\_comp \leftarrow$ None
17:    for each component $comp$ in $connected\_comps$ do
18:        if $cur\_node$ in $comp$ then
19:            $current\_comp \leftarrow comp$
20:            break
21:        end if
22:    end for
23:    for each node $i$ in $unsaturated$ do
24:        if $i$ not in $current\_comp$ then
25:            Append $i$ to $options$
26:        end if
27:    end for
28:    $nearest\_node \leftarrow$ min($options$, key=lambda node: distance between $cur\_node$
   and $node$)
29:    Remove $nearest\_node$ from $unsaturated$
30:    Initialize $joining\_comp \leftarrow []$
31:    for each component $comp$ in $connected\_comps$ do
32:        if $nearest\_node$ in $comp$ then
33:            $joining\_comp \leftarrow comp$
34:            break
35:        end if
36:    end for
37:    Append $current\_comp + joining\_comp$ to $connected\_comps$
38:    Remove $joining\_comp$ and $current\_comp$ from $connected\_comps$
39:    Append edge between $cur\_node$ and $nearest\_node$ to $tour\_edges$
40: end while
41: Connect remaining unsaturated nodes
42: Reconstruct tour path from $tour\_edges$
43: Assign path to $current\_tour$
```

Algorithm 5 Nearest Insertion

```
1: Initialize $tour = [city1, city2]$, $visit = \{city1, city2\}$
2: Initialize $total\_cost$ with the distance between $city1$ and $city2$
3: while $|visited| <$ the total number of cities do
4:     Find the nearest city $city \notin tour$
5:     Determine the best insertion point in $tour$ for $city$ which minimizes the insertion cost $total\_cost$
6:     Insert $city$ into $tour$ at the determined point
7: end while
8: Close the tour by connecting the last and first cities if not already connected
```

Algorithm 6 Cheapest Insertion

```
1: Initialize $tour = [city1, city2]$, $visit = \{city1, city2\}$
2: Initialize $total\_cost$ with the distance between $city1$ and $city2$
3: while $|visited| <$ the total number of cities do
4:     Compute insertion cost for each city $city$ that is not in $visit$
5:     Determine the best insertion point in $tour$ for $city$ which minimizes the insertion cost
6:     Insert $city$ into $tour$ at the determined point
7:     Add $city$ to $visited$
8: end while
9: Close the tour by connecting the last and first cities if not already connected
```

Algorithm 7 Random Insertion

```
1: Initialize $tour = [city1, city2]$, $visit = \{city1, city2\}$
2: Initialize $total\_cost$ with the distance between $city1$ and $city2$
3: while $|visited| <$ the total number of cities do
4:     Select a city $city$ randomly from cities not in $visit$
5:     Determine the best insertion point in $tour$ for $city$ which minimizes the insertion cost
6:     Insert $city$ into $tour$ at the determined point
7:     Add $city$ to $visited$
8: end while
9: Close the tour by connecting the last and first cities if not already connected
```

Algorithm 8 Farthest Insertion

```
1: Initialize $tour = [city1, city2]$, $visit = \{city1, city2\}$
2: Initialize $total\_cost$ with the distance between $city1$ and $city2$
3: while $|visited| <$ the total number of cities do
4:     Select the farthest $city$ from cities not in $visit$
5:     Determine the best insertion point in $tour$ for $city$ which minimizes the insertion cost
6:     Insert $city$ into $tour$ at the determined point
7:     Add $city$ to $visited$
8: end while
9: Close the tour by connecting the last and first cities if not already connected
```

Algorithm 9 2-opt Algorithm

```
1: Given an initial tour $T$
2: while improvements can be made do
3:     for each unique combination of three edges $(e_1, e_2)$ in $T$ do
4:         Determine all possible reconnections of the segments not forming a cycle,
           excluding the current configuration
5:         for each reconnection configuration do
6:             Calculate the total cost of the tour with this reconnection
7:             if this reconnection offers a lower cost then
8:                 Update $T$ with this reconnection
9:                 break                    ▷ Exit and restart with the improved tour
10:            end if
11:        end for
12:    end for
13: end while
14: return Improved tour $T$
```

3.2.8 3-opt Algorithm

The 3-opt algorithm extends the 2-opt approach by considering swaps involving three edges instead of two. This method allows for more complex rearrangements of the tour, potentially unlocking further improvements in the tour cost by examining a broader set of possible changes (See Figure 3.7). The pseudocode for the 3-opt algorithm is similar to that of the 2-opt but includes an additional consideration for a third edge, e_3, thus we omit its detailed presentation for brevity.

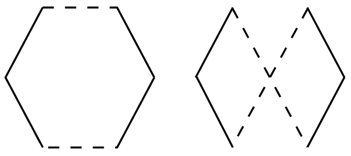


Figure 3.6: An example of a 2-opt move. The left one is the original tour and the right one is the resulting tour after swapping edges using 2-opt.

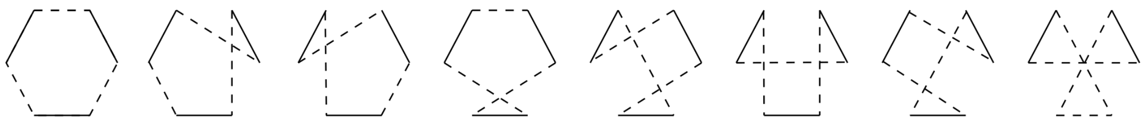


Figure 3.7: All possible reconnections using 3-opt.

Through a combination of construction and improvement heuristics, we develop efficient and practical solutions for the TSP, optimizing our approach to balance computational feasibility with solution quality. Within the scope of our project, we have adopted the Nearest Neighbor algorithm as our foundational heuristic for constructing initial solutions. Subsequently, we refine these solutions by incorporating both 2-opt and 3-opt techniques, which serve to iteratively improve the solution quality. The culmination of this heuristic process yields a final cost, which we then establish as the upper limit for our Branch and Bound algorithm. This approach not only streamlines the problem-solving process but also ensures that our solutions are both practical and computationally viable.

3.3 Exact Solutions

Exact solution procedures for the Traveling Salesman Problem (TSP) guarantee to find provably optimal solutions. Unlike heuristic methods, which are invaluable in practice for generating near-optimal tours with relatively low computational effort, the pursuit of exact solutions is not driven by practical applicability but by their significant contributions to the fields of optimization and mathematical programming. The TSP has been a fundamental benchmark for optimization methods, spurring the development of many key techniques in the field (see Dantzig, Fulkerson, and S. M. Johnson 1954).

Perhaps the most intuitive exact solution procedure, *brute force*, involves calculating all possible tour costs and simply identifying the minimum tour among them. This method is guaranteed to provide an optimal solution, via exhaustive evaluation, however it quickly becomes prohibitively expensive from a computational standpoint. The number of distinct Hamiltonian tours in a complete graph on n vertices is $\frac{(n-1)!}{2}$, rendering the brute force approach practically infeasible for anything over 15 vertices (there are more than 4×10^{10} distinct tours in a complete graph on 15 vertices). This computational reality underscores the necessity for more refined exact solution strategies, which aim to achieve the precision of brute force with reduced computational demand.

This paper focuses on the exact solution methodology of the Branch & Cut algorithm, introduced in Section 2.1.4, applied to the ILP formulation of the TSP detailed in Section 3.1. We begin by defining TSP polytopes and exploring their characteristics, providing a foundation for a detailed discussion of the Branch & Cut methodology in the context of the TSP.

3.3.1 TSP and related polytopes

Definition 2.1.4 defined the polytope Q of an ILP as the minimal convex hull encompassing all feasible integer solutions. We now proceed to define this polytope for the TSP and explore its characteristics.

Definition 3.3.1. Given a TSP instance on n nodes we define $\mathcal{H}_n$ as the set of points satisfying constraints ILP.2, ILP.3 & ILP.4. Specifically:

$$\mathcal{H}_n := \left\{ (x_e)_{e\in E} \middle| \left(\sum_{e\in\delta(v)} x_e = 2, \forall v \in V \right), \left(\sum_{e\in\delta(S)} x_e \geq 2, \forall S \in \mathcal{NR} \right), (x_e \in \{0,1\}, \forall e \in E) \right\}.$$

This defines $\mathcal{H}^n$ as the set of sets $(x_e)_{e\in E}$ which correspond to Hamiltonian cycles in the graph K^n, or equivalently as the set of integer feasible solutions to an n node TSP. Thus given Definition 2.1.9 we define:

Definition 3.3.2 (TSP Polytope)**.** The n node TSP polytope is given by:

$$Q_T^n := \text{conv}\{\mathcal{H}_n\}.$$

Q_T^n denotes the convex hull of all feasible solutions to an n node TSP instance. Thus if Q^n can be completely defined the TSP is transformed into an LP problem of finding the minimum cost vertex of Q_T^n. We now define the polytope of the degree relaxation to the TSP, that is the polytope delimited by the relaxation to the ILP formulation of the TSP in which SECs are disregarded:

Definition 3.3.3 (LP Relaxation Polytope)**.** The n node TSP relaxation polytope Q_R^n is the set of feasible solutions to constraint ILP.2 and the relaxation of integer requirements

ILP.4. Specifically:

$$Q_R^n := \left\{ (x_e)_{e\in E} \middle| \left(\sum_{e\in\delta(v)} x_e = 2, \forall v \in V \right), (0 \leq x_e \leq 1, \forall e \in E) \right\}.$$

Since Q_R^n represents a relaxation of Q_T^n, we write: $Q_T^n \subseteq Q_R^n$.

We introduce a minor variation of a pivotal theorem from Lawler et al. 1985 which classifies a system of non-redundant facet defining inequalities for Q_T^n.

Theorem 3.3.1. *For a TSP instance on $n \geq 6$ nodes with the set $\mathcal{NR}$ defined as in Definition 3.6, the following defines a system of non-redundant facets for Q_T^n:*

a) $x_e \geq 0, \quad \forall e \in E;$

b) $x_e \leq 1, \quad \forall e \in E;$

c) $\sum_{e\in\delta(S)} x_e \geq 2, \quad \forall S \in \mathcal{NR};$

d) $x(E(H)) + \sum_{j=1}^{t} x(E(T_j)) \leq |H| + \sum_{j=1}^{t} |T_j| - \lceil \frac{3t}{2} \rceil$

$\forall H \in \mathcal{NR}$ *and* $T_1, \dots, T_t \subset V$ *satisfying:*

$d_1)$ $|H \cap T_j| \geq 1, \quad j = 1, \dots, t;$

$d_2)$ $|T_j \backslash H| \geq 1, \quad j = 1, \dots, t;$

$d_3)$ $T_i \cap T_j = \emptyset, \quad 1 \leq i \leq i \leq t;$

$d_4)$ $t \geq 3$ *and odd.*

Recalling that the facets of a polytope constitute the minimal set describing the polytope, this theorem provides a new level of detail regarding the structure of Q_T^n. It affirms that all non-redundant SECs and comb inequalities define distinct facets of Q_T^n the implications of which are twofold. Firstly it justifies these inequalities as ideal cutting planes (see Section 3.3.2), and secondly recalling the exponential cardinality of $\mathcal{NR}$ (see 3.7) it provides a small glimpse of the complexity of Q_n^T ending any hope of a full description and subsequent LP formulation.

Definition 3.3.4 (Extension Complexity)**.** The *extension complexity* of a polytope P is the minimum number of facets needed to describe a polytope P' such that P can be obtained from P' via projection.

In our case, it suffices to notice that the extension complexity of a polytope serves as a lower bound to the minimal number of facets needed to fully describe the polytope (within its own dimension). Thus to emphasize the complexity of Q_T^n we cite the following lower bound from (Fiorini et al. 2015):

Theorem 3.3.2. *A lower bound for extension complexity of Q_T^n is 2^n.*

Thus the number of facets of Q_T^n is at least exponential in n.

3.3.2 Branch and Cut

We now detail the application of the branch and cut algorithm introduced in Section 2.1.4, as an exact solution to the TSP. Recall that the branch and cut methodology is built upon the branch and bound framework which operates by solving a series of LP relaxations of the core ILP problem. In the case of the TSP we define the LP relaxation of the ILP formulated in Section 3.1.4 as follows:

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ \text{s.t.} \quad & \sum_{e \in \delta(v)} x_e = 2, \;\; \forall v \in V; \\ & 0 \le x_e \le 1, \quad \forall e \in E. \end{aligned} \tag{LP}$$

Here the SECs have been completely removed from the model as a result of their exponential quantity rendering them infeasible to fully enumerate for almost all n. Additionally, the integrality requirement for each variable x_e has been relaxed to the domain $[0,1]$. The result is a computationally feasible LP model with a convex feasible region given by Q_R^n (see Definition 3.3.3). This model serves as the core LP solved at the root node of the branch and bound solution to the TSP, however it does not guarantee optimal solutions in that they may violate some number of SECs.

SEC Enforcement

To obtain optimal solutions to the TSP, we must identify integer feasible solutions that do not violate any SECs, specifically solutions within the TSP polytope Q_T^n. The LP relaxation above provides solutions within $Q_R^n \supseteq Q_T^n$, and thus in order to ensure that we ultimately obtain solutions within Q_T^n we adopt a strategy similar to the implementation cutting planes within the branch and cut framework.

Technically for every solution $(x_e)_{e\in E}$ obtained in the branch and bound tree that is not pruned (i.e. which is less than the standing incumbent) we augment the typical execution according to:

- **Integer feasible solutions** (solutions wherein $x_e \in \{0,1\}, \forall e \in E$):
  - If $(x_e)_{e\in E}$ violates *any* SECs then at least one of these SECs *must* be identified and added as a constraint in the LP model.
  - If $(x_e)_{e\in E}$ does not violate *any* SECs then update the incumbent solution.
- **Solutions violating some positive number of integrality constraints**:
  - Identify SECs violated by $(x_e)_{e\in E}$ and add these constraints to the LP model.

In this process, a solution is accepted if and only if it violates no SECs and thus constitutes a feasible solution to the TSP, with the strategy's efficacy hinging upon an optimal solution being reached without requiring all SECs to be added to a model. In practice (as will be shown in Chapter 5) optimal solutions are reached after only a very small fraction of the possible SECs are added.

We note that the SECs here are implemented in a similar manner to the typical cutting planes of the branch and cut framework with the crucial distinction of being added to ensure that the solution obtained is *feasible* rather than simply expediting the solution process. Furthermore, any implementation of this strategy requires a version of the separation problem to be solved for the SECs. Strictly speaking Definition 2.1.14 requires that a solution fall inside the feasible region of the full LP relaxation (i.e. with SECs included) for it to be the subject of a separation problem and in this case we are dealing with solutions which exist outside of this region (if they violate any SECs). However given that the SECs induce facets of Q_T^n (see Theorem 3.3.1) and thus ultimately force solutions into the Q_T^n, which is exactly the objective of any other cutting plane, we will refer to this as the *SEC separation problem.* The mechanics of this separation will be covered along with the separation of the typical TSP cutting planes in section 3.4.

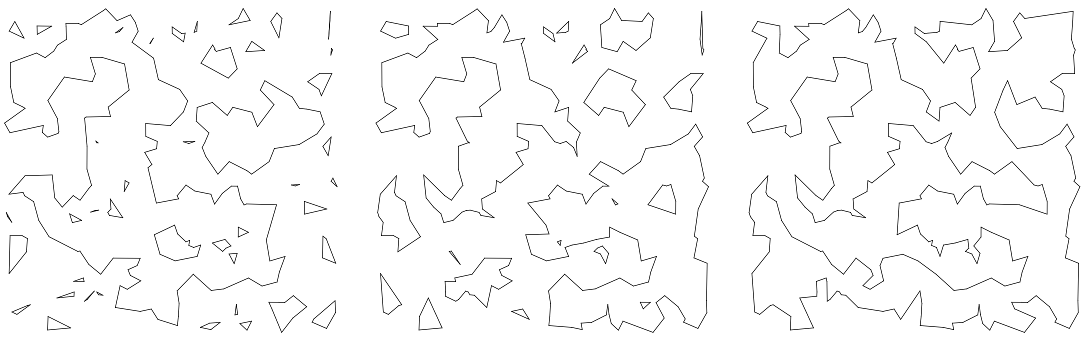

(a) Initial LP Relaxation (b) Iteration 1 (c) Optimal solution (Iteration 9)

Figure 3.8: Evolution of an optimal solution to a 300 node random TSP instance after 9 iterations and 106 SECs.

Figure 3.8 depicts a branch and bound solution to the TSP wherein violated SECs were added at each integer feasible solution,[4] a slight simplification of the strategy detailed in that SECs are not identified at fractional solutions. This is done for illustrative purposes resulting in a slightly less efficient solution algorithm, but providing a more intuitive visual representation. The process resulted in the optimal solution being attained after only 9 iterations with a total of 106 SECs added to the model. These SECs represent $1.04 \times 10^{-88}\%$ of the total non-redundant SECs for this model (given by 3.7) highlighting the efficacy of this selective enforcement strategy.

Cutting Planes

Having outlined a functional branch and bound methodology for the Traveling Salesman Problem (TSP), we now turn our attention to the cutting plane augmentations, detailed in Section 2.1.4. Cutting planes in the context of the TSP are valid inequalities added to the problem's formulation at non-integer feasible solutions in an attempt to steer the solution path towards the feasible region Q^n_T. In this sense we view the facets of Q^n_T as "ideal cuts", in that they minimize the feasible region to the greatest extent possible amongst all half-spaces sharing their orientation, aligning exactly with a maximal face of Q^n_T. Over the last fifty years, there have been significant advances and developments in the field of TSP cutting planes, with new facets being continually discovered, a further testament to the complexity of Q^n_T. Among these cutting planes we highlight some pertinent examples:

- **Clique Tree Inequalities**: Introduced by M. Grötschel and Pulleyblank 1986, these generalize the comb inequalities detailed in Section 3.1.5, identifying a broader set of constraints based on the clique structures within the graph.
- **Path Inequalities**: Documented by Cornuejols, Fonlupt, and Denis Naddef 1984, these inequalities provide a generalisation of the comb inequalities, distinct from clique trees in that they identify inequalities not identified by and exclude some inequalities identified by clique trees.
- **Crown Inequalities**: Formulated by D. Naddef and Rinaldi 1992, these inequalities constitute a novel class of cutting planes for the TSP.
- **Circlet Inequalities**: A recent addition to the cutting-plane arsenal, identified by Gutekunst and Williamson 2020, these inequalities introduce constraints based

[4]Here a simple breadth first search (BFS) was used to identify subtours.

on circulant properties of the graph, offering a novel perspective on trimming the solution space.

In integrating such cutting planes into the branch and bound framework, the "branch and cut" methodology attempts to manage the astronomical complexity of Q_T^n in the hope of reducing the computation required for the branch and bound tree to identify an optimal solution.

The efficacy of cutting planes in expediting the convergence to an optimal solution is well-documented; however, their practical application is not without challenges. The primary hurdle is the computational expense associated with the separation problem (see Section 3.4, that is the task of finding a violated cutting plane at any iteration. Thus in any cutting plane implementation for the TSP, there is a trade-off between the quality of cuts identified and the computation required to identify them. In this paper, we implement and detail the separation procedure for the specific 2-matching case of the comb inequalities (due to Edmonds 1965).

Integrating Heuristics

In order to reduce the computation required within the branch and bound tree, heuristics may be employed to potentially improve the incumbent solution, thus facilitating more effective pruning. More specifically at any stage of the tree, a combination of the heuristics detailed in 3.2 or others can be called upon to provide a feasible solution to the problem. The construction of these solutions broadly falls into two categories:

- **Full construction**: Heuristics are used to construct a feasible solution from scratch. This is particularly useful at the root node of the branch and bound process when no incumbent has yet been identified.

- **Refinement of a current solution**: Heuristics are used to create feasible solutions from current infeasible solutions or to refine current feasible solutions. This is particularly useful for fractional solutions and those violating some number of SECs.

In practice, the timing and selection of heuristics to integrate into the branch and bound tree is dictated by the balance between the potential improvements the heuristics can provide and the computation required to execute them.

3.4 Cut separation procedures

Having outlined the branch and cut solution strategy for the TSP, it is apparent that the efficacy of this solution strategy relies heavily on the precision and efficiency of the underlying cut separation procedures. To obtain optimal solutions to the TSP using these methodologies it is a requirement that violated SECs be identified from integer feasible solutions whenever they exist. However, the incredible complexity of the TSP polytope Q_T^n revealed by Section 3.3.1, suggests that performing such separations may be a very difficult and computationally expensive task. This tension between necessity and looming complexity prompts the following categorisation of separation procedures:

- **Exact Separation Procedures**: procedures which guarantee the detection of at least one violated constraint of their target type, provided such a violation exists within the solution.

- **Heuristic Separation Procedures**: procedures which do not guarantee the identification of a violated constraint in the event that such a violation exists.

This distinction, in general, affords the ability to utilise the guarantee associated with the typically high computational expense of exact separation procedures, while benefiting from the efficiency and speed of heuristic separation procedures when completeness is non-essential.

3.4.1 SEC Separation

The following details a system of SEC separation procedures which attempts to balance efficiency and rigor through conditional adoption of exact and heuristic separations.

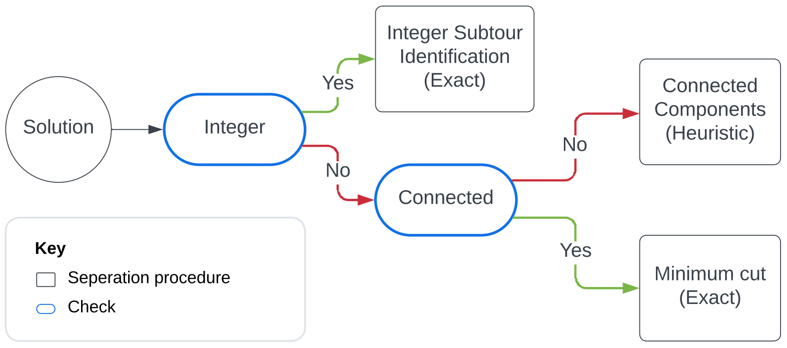


Figure 3.9: Conditional SEC separation process.

As mentioned, identifying SEC violations at integer feasible solutions is paramount, necessitating the use of an exact separation procedure. The inherent simplicity of integer solutions affords a straightforward separation procedure in which subtours are identified, with each representing an SEC violation. This simple exact separation requires relatively low computation and is thus an ideal implementation. Its efficacy does not however extend to fractional solutions thus requiring us to condition its use upon a solution being integer feasible.

To handle fractional solutions, the strategy becomes contingent upon the connectivity of a solution's vertices. Specifically, If a fractional solution is found to be disconnected, the Connected Component separation heuristic is employed. This method efficiently extracts violated SECs from fractional solutions with low computational expense. Conversely, should the solution be fully connected, we resort to the more computationally intensive Minimum Cut exact separation. This guarantees the identification of violated SECs within a fractional solution, provided any exist.

This process, as detailed in Figure 3.9, ensures the detection of violated SECs across all solutions while minimizing computational overhead in cases of fractional disconnected or integer feasible solutions.

Integer Solution Subtour Identification

This section uncovers the mechanics of the subtour identification within integer solutions as an exact separation SEC procedure. This process entails identifying all subtours in the given solution, with the set of vertices through which each passes constituting a set S for which there exists a violated SEC. Note that every such subtour must yield a violated SEC (see Section 3.1.3).

Any integer solution which violates some positive number of SECs *must* contain subtours. This implies that if the subtours in an integer solution can be reliably identified, this separation procedure will always yield a set of violated SECs (in the event that a solution violates any SECs). This is exactly the requirement for an exact solution procedure and ultimately justifies this procedure's use for this indispensable separation.

To identify subtours within an integer solution, a common graph traversal algorithm such as Depth-First Search (DFS) or Breadth-First Search (BFS) can be employed effectively. DFS, in particular functions by selecting an arbitrary node as the starting point and exploring as deep as possible into each walk from this node before backtracking. We provide a more intuitive subtour identification routine:

Algorithm 10 Subtour Identification from Integer Solution

```
 1: $subtours \leftarrow []$
 2: while $edges$ is not empty do
 3:     $(x, y) \leftarrow firstelementofedges$
 4:     $cycle \leftarrow [(x, y)]$
 5:     Remove$((x, y))$ from $edges$
 6:     while $x \neq y$ do
 7:         for $edge$ in $edges$ do
 8:             if $x$ in $edge$ then
 9:                 Insert$(cycle, 0, edge)$
10:                 $x \leftarrow edge[0]$ if $edge[0] \neq x$ else $edge[1]$
11:                 Remove$(edges, edge)$
12:             else if $y$ in $edge$ then
13:                 $cycle$.Append$(edge)$
14:                 $y \leftarrow edge[0]$ if $edge[0] \neq y$ else $edge[1]$
15:                 Remove$(edges, edge)$
16:             end if
17:         end for
18:     end while
19:     $subtours$.Append$(cycle)$
20: end while
21: return $subtours$
```

This algorithm functions by selecting an edge from the graph as the first edge in a walk and then successively gluing edges onto either end of this walk until the two ends meet and thus a subtour has been identified; this proceeds until all edges in the graph have been added to some subtour. Despite a theoretical worst-case complexity of $\mathcal{O}(n^2)$, preliminary tests on graphs with fewer than 2000 vertices have shown this algorithm to perform very similarly to traditional DFS (which has time-complexity $\mathcal{O}(n)$) in terms of average execution time.

Connected Components

Given a disconnected fractional solution, the connected components heuristic separation can be used to rapidly identify violated SECs. Specifically, upon identifying each connected component within the solution, the vertices making up each component can be used as the set S over which to find a violated SEC. Recall that the SECs require that the sum of edges leaving a subset of vertices (i.e. edges in $\delta(S)$) be at least 2. In the case of a disconnected graph, there are exactly 0 edges leaving each connected component and thus each such component constitutes a violated SEC. This provides a very low cost identification routine for violated SECs. In practice, a simple DFS algorithm is generally used to identify the connected components.

Minimum Cut

Recalling that a cut in the graph is exactly a set of edges which disconnects the graph, or rather partitions the graph into two disjoint sets of vertices, we can utilize the minimum cut algorithm detailed in Section 1 to identify violated SECs. Specifically, if the minimum cut is calculated to have a value of less than 2; this implies that the sum of the edges bridging the two sets in which the cut partitions must be less than 2; which is exactly a violated SEC. Thus, by choosing either of the partition sets and labeling it S, we can add the violated SEC over S. This constitutes an exact separation routine, as if there exists a violated SEC, then the minimum cut in the graph must have a value of less than 2.

3.4.2 2-Matching Inequality

This section details a common separation procedure for the 2-Matching inequality of Edmonds 1965. First we must specify what is meant by a 2-matching and its associated inequality. For this we define a simplification of the structure of a comb given in Definition 3.1.2.

Definition 3.4.1. A *2-matching* within a graph G is a comb in which:

1. $|T_j \cap H| = 1, \quad j = 1, \ldots, t;$
2. $|T_j \backslash H| = 1, \quad j = 1, \ldots, t.$

Thus 2-Matchings are combs in which all teeth consist of exactly 1 edge, or equivalently in which $T_j \in \delta(H)$, $\forall j \in [1, t]$. This simplification naturally facilitates a refinement of the comb inequality given in Definition 3.1.3 as follows: we notice that in a 2-Matching every tooth consists of exactly 2 vertices (those joined by its edge), or specifically $|T_j| = 2, \forall j \in [1, t]$. This allows us to rewrite the comb inequality for the specific case of 2-Matchings as:

$$\sum_{e \in E(H)} x_e + \sum_{j=1}^{t} \sum_{e \in E(T_j)} x_e \leq |H| + \sum_{j=1}^{t} 2 - \left\lceil \frac{3t}{2} \right\rceil = |H| + 2t - \left\lceil \frac{3t}{2} \right\rceil . \tag{3.11}$$

Next we give the equality:

$$2t - \left\lceil \frac{3t}{2} \right\rceil = t - \left\lceil \frac{t}{2} \right\rceil = \left\lfloor \frac{t}{2} \right\rfloor , \tag{3.12}$$

where the final equality holds as $\lceil \frac{t}{2} \rceil + \lfloor \frac{t}{2} \rfloor = t$. Substituting 3.12 into 3.11 completes the derivation of the following 2 Matching inequality:

Definition 3.4.2 ((Edmonds 1965), 2-Matching Inequality)**.** Given a 2-Matching with handle H and associated teeth $T_j, j = 1, \ldots, t$, the *2-Matching Inequality* is given by

$$\sum_{e \in E(H)} x_e + \sum_{j=1}^{t} \sum_{e \in E(T_j)} x_e \leq |H| + \left\lfloor \frac{t}{2} \right\rfloor .$$

Separation Procedure

This subsection details an effective heuristic separation procedure for violated 2-Matching Inequalities from a given fractional solution during the branch and cut algorithm. There exist exact separation procedures (see A. Letchford, Reinelt, and Theis 2004), however we elect to detail and later implement the following heuristic procedure due to its simplicity and low computational overhead.

Given a fractional solution $(x_e)_{e\in E}$ to a TSP instance on a graph G, we define the fractional support graph:

Definition 3.4.3 (Fractional Support Graph)**.** G_F is the graph obtained from G by including only those edges for which the corresponding variable $x_e \in (0,1)$:

$$G_F = (V, E_F = \{e \in E \mid x_e \in (0,1)\}).$$

Notice that this excludes all edges $e \in E$ for which $x_e = 1$.

Within the Fractional Support Graph G_F, a DFS Algorithm can be applied to identify all connected components. Each set of vertices S which makes up a connected component in G_F is then considered as a potential handle for a 2-Matching structure. For each potential handle S the edges within the set $\delta(S)$ for which $x_e = 1$ in the solution $(x_e)_{e\in E}$ are identified as potential teeth for S. If the number of potential teeth for S found in this way is odd and greater than or equal to 3 then we proceed to determine if any of these teeth are incident to the same vertex outside of S. We refer to such pairs of teeth as 'false teeth'.

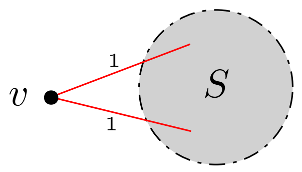


Figure 3.10: A false tooth for the set S.

For every 'false tooth', the two edges which constitute it are removed from the set of potential teeth, and the vertex to which both edges were adjacent is added to the potential handle set S. After this process has taken place for all false teeth - if the number of potential teeth is still greater than or equal to 3 - then the structure that remains is exactly a 2-Matching which violates the 2-Matching inequality as we will now show.

For any 2-Matching obtained in this way, we consider the sum $\sum_{e\in E(H)} x_e$, or specifically the sum of all edge weights between vertices within the handle. Notice that every vertex in the handle is adjacent to only other vertices within the handle, unless it is one of the t vertices to which a tooth is incident. Furthermore since the solution $(x_e)_{e\in E}$ must satisfy the degree constraints, we deduce that every vertex in the handle to which a tooth is not incident must contribute exactly 2 to the sum of the degrees within the 2-Matching. Additionally the t vertices to which teeth are incident must contribute exactly 1 to this sum of degrees as the edge within any tooth is of weight 1. In this way the sum of degrees within the handle is exactly:

$$2(|H| - t) + t = 2|H| - t,$$

and thus by the handshaking lemma 2.2.3 the sum of the edge weights within the handle is given by:

$$\sum_{e\in E(H)} x_e = |H| - \frac{t}{2}. \tag{3.13}$$

Now notice that every tooth has exactly 1 edge of weight 1 and thus:

$$\sum_{j=1}^{t} \sum_{e\in E(T_j)} x_e = \sum_{j=1}^{t} 1 = t. \tag{3.14}$$

Thus bringing together 3.13 and 3.14, the sum of the edge weights in the 2-Matching is given by:

$$\sum_{e\in E(H)} + \sum_{j=1}^{t} \sum_{e\in E(T_j)} x_e = |H| - \frac{t}{2} + t = |H| + \frac{t}{2},$$

which since t is odd, results in a violation of the 2-Matching Inequality given by Definition 3.4.2 as:

$$|H| + \frac{t}{2} \geq |H| + \left\lfloor \frac{t}{2} \right\rfloor .$$

Chapter 4

Implementation

This section details our implementation of the branch and cut algorithm outlined in Section 3.3.2. The implementation is written in Python 3.10 using IBM's CPLEX version 22.1.1.0 to handle the core branch and cut algorithm including the LP solving. We use the DOcplex API as the communication interface between our code and CPLEX, specifically utilizing *callbacks* to communicate with CPLEX during the execution of the branch and cut algorithm.

Definition 4.0.1. A **callback** in computer programming refers to executable code that is passed as an argument to another piece of code.

These Callbacks allow us to implement the SEC and cut separation procedures outlined in Section 3.4, as well as to attempt incumbent updating through the use of a selection of the heuristic procedures given in Section 3.2.

Definition 4.0.2. The *Integer Programming (IP) Gap* is a measure of the difference between the incumbent value (z^*) and lower bound (LB) on the objective value of an ILP at any point in a solution process. Specifically assuming $z^* \neq 0$:

$$\text{IP Gap} = \frac{z^* - LB}{z^*}.$$

Informally this IP gap serves as a measure of how close the solution procedure is to the optimal solution, a crucial metric watched to understand when to terminate a solution process.

4.1 Warm-start

In order to provide an initial incumbent solution to the branch and bound algorithm we begin processing by implementing the Nearest Neighbour Algorithm from Section 3.2.2 to construct a feasible solution to a given TSP instance. We then use the two-opt improvement algorithm of section 3.2.7 in an attempt to improve this feasible solution. The obtained solution may then be given to CPLEX when the branch and bound algorithm is initialised so as to facilitate the pruning of subproblems (via their objective value being greater than this incumbent) at very early stages in the process. This allows us to potentially prevent the exploration of subproblems which may blossom into large trees having wasted significant computation both to create and then further to prune completely.

4.2 Branch and Cut Implementation

We now unpack CPLEX's execution of the branch and cut algorithm, along with our implemented SEC, cutting plane, and heuristic augmentations. The following flow chart gives a conceptual overview of the order of executions within the integration of our code and CPLEX's branch and cut algorithm.

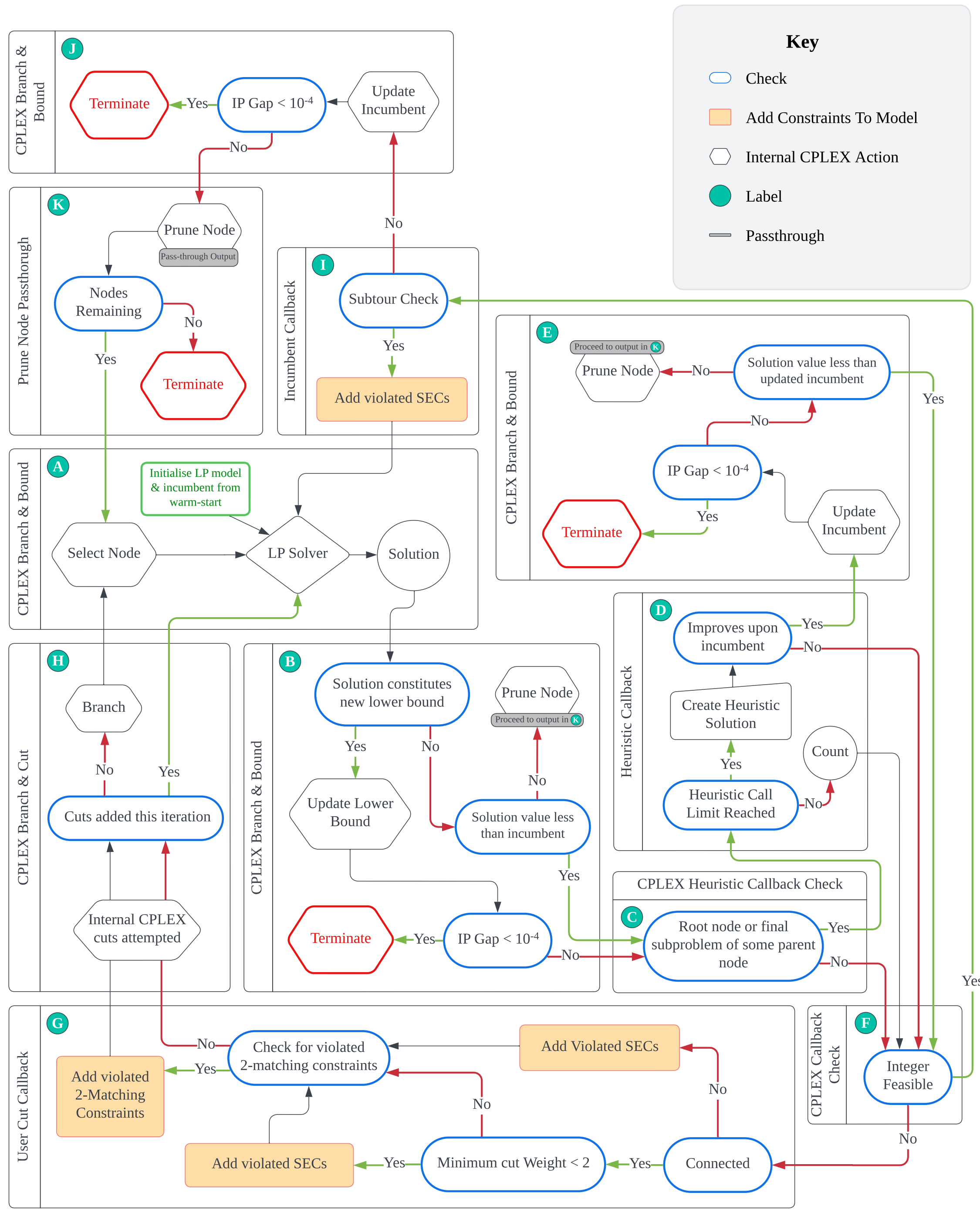


Figure 4.1: The order of processing in our Branch and Cut TSP solution implementation.

When referencing this figure we will refer to the containers labeled *A*-*K* as *cells*. This grouping provides an easier method of reference throughout the explanation as well as providing a high level segmentation of processes.

4.2.1 Initial Branch and Bound

Initialisation (Cell A)

CPLEX is initialized to solve the LP relaxation given by LP in Section 3.3.2 with the initial solution value z^* obtained from the warmstart process. This initialisation is represented by the green process in cell A, from which CPLEX proceeds to solve this LP relaxation obtaining a solution: $(x_e)_{e\in E}$, with objective value: v.

Branch and Bound Checks (Cell B)

Cell B details CPLEX's internal branch and bound comparisons. Firstly a check is performed to deduce whether the solution constitutes a new lower bound for the problem. That is specifically if v is the new minimum amongst the minimum solution values to all feasible, active (non-pruned) regions of LP relaxation search space. Crucially, as the TSP is a minimisation problem, these minimums can never decrease as cuts are added, translating into a monotonically increasing lower bound. The process continues according to:

i. **If $(x_e)_{e\in E}$ constitutes a new lower bound**: the lower bound is updated to v and CPLEX proceeds to check if the new IP gap $< 10^{-4}$. This is the default termination tolerance within CPLEX's branch and bound algorithm.

   – *If IP gap* $\leq 10^{-4}$: the standing incumbent is within $10^{-4}\%$ of the optimal solution and thus the algorithm terminates providing the standing incumbent as it's ultimate solution.

   – *If IP gap* $> 10^{-4}$: execution proceeds to the Heuristic Callback Check in cell C. (The significant IP gap implies this solution (LB) is some positive amount less than the incumbent; meaning it can skip the comparison detailed next.)

ii. **If $(x_e)_{e\in E}$ does not constitute a new lower bound**: CPLEX proceeds to compare the solution value v with the standing incumbent value z^*:

   – *If* $v \geq z^*$: CPLEX prunes this node, represented by the pass-through to cell K (progression from cell K will be detailed in Section 4.2.6).

   – *If* $v < z^*$: CPLEX proceeds to the Heuristic Callback Checks in cell C.

4.2.2 Heuristic Callback

The *heuristic callback* allows a combination of the heuristic solutions detailed in Section 3.2 to be used to construct feasible solutions through some adjustment of a current solution. This is done in the hope of improving upon the incumbent solution at points throughout the branch and bound process. The following details our implementation of the Integrating Heuristics topic of Section 3.3.2.

CPLEX Activation Criteria (Cell C)

Initially, CPLEX evaluates a solution's position in the branch and bound tree to determine whether to initiate the `Heuristic Callback`. This evaluation, specified in cell C, triggers the callback (proceeding into cell D) under either of the following conditions:

i. The solution emanates from the root LP problem, existing before any branching has occurred.

ii. The solution represents the final subproblem (branched child) to be addressed under a specific parent node.

The rationale behind each of these criteria respectively is:

i. Similar to the warmstart, at this early stage, improved incumbents can facilitate the pruning of branches that could otherwise branch into large trees, wasting significant computation.

ii. Updated incumbents at this juncture, may facilitate the pruning of a portion of these child problems, limiting the number of new branches stemming from this subset of nodes.

If neither condition is met CPLEX proceeds to cell F (Section 4.2.3).

Custom Activation Criteria (Cell D)

Within the `Heuristic Callback`, we have introduced a call limiter to afford more nuanced control over the frequency of heuristic callback executions. Specifically, we implement this limiting as a function of the IP Gap so as to reduce the frequency of execution as a solution approaches optimality. This was done as a result of observed high failure rates (thus wasted computation) of the heuristic procedures for solutions with low IP gaps. Specifically we define a function:

$$f(\text{IP Gap}) = \left\lfloor \frac{b}{(\text{IP Gap} * 100)^a + 10^{-6}} \right\rfloor ,$$

where a and b are adjustable parameters used to fine-tune the function's behavior, and 10^{-6} is added to the denominator to prevent division by 0 (if called for a very small IP gaps this may overflow into 0 when raised to the power of a).

Thus cell D initially compares a count (initialised to 0) to a limit (initialised to 1), executing according to:

i. **If count $<$ limit**: Increment the count by 1 and proceed to cell F (Section 4.2.3).

ii. **If count $=$ limit**: reset the count to 0 and calculate a new limit using $f(\text{IP Gap})$. proceed to construct a heuristic solution.

Constructing Heuristic Solutions (Cell D)

In the construction of a heuristic the goal is to obtain a feasible solution through some modification of the current solution. The initial construction is as follows:

1. Sort the edges in the current solution by their weights (descending),

2. Iterate through every edge sequentially, constructing a graph $\bar{G}$ wherein an edge is added to $\bar{G}$ if and only if it meets the following criteria:

   i. *It does not lead to any vertex in $\bar{G}$ having 3 incident edges.* This is checked by keeping a count of the number of edges incident to every vertex and ensuring this count is < 2 for both vertices to which the edge under consideration is incident.

   ii. *It does not form a subtour in $\bar{G}$.* This is done by checking that the two vertices to which the edge under consideration is incident are not connected in the current graph (using a DFS algorithm).

At this stage $\bar{G}$ is typically some disconnected set of edges which will be used as the base from which to construct a heuristic solution. Continuing an altered version of the nearest neighbour heuristic (Algorithm 4) is applied as pere:

3. Create a list of all vertices in $\bar{G}$ with less than 2 incident edges (these are exactly the vertices which require incident edges),
4. Select a vertex v_i from this list and identify all the other vertices in the list to which it is not connected (i.e the vertices in the list with all vertices in connected component within which v_i lies removed),
5. From this set of candidate vertices select the one that is closest to v_i and add the edge between v_i and this vertex to $\bar{G}$,
6. If either of these vertices now have two edges incident to them remove them from the list and repeat from step 4 until all vertices have been removed.

Upon completion $\bar{G}$ contains a Hamiltonian cycle. The two-opt algorithm is then applied exactly as detailed in Algorithm 9 in an attempt to improve the solution value. We then preform a check on the length of this final Hamiltonian cycle and compare it's value $\bar{v}$ to the standing incumbent value z^* as follows:

i. *If* $\bar{v} \geq z^*$: we have failed to find an improved incumbent and execution proceeds to cell F.

ii. *If* $\bar{v} < z^*$: we have found an improved incumbent and we instruct CPLEX to update the current incumbent solution - proceeding to cell E.

Standing Incumbent Updating (Cell E)

After updating the incumbent solution CPLEX calculates the new IP gap and proceeds according to:

i. *If IP gap* $< 10^{-4}$: the new incumbent is within $10^{-4}\%$ of the optimal solution and the algorithm terminates providing the updated incumbent as it's ultimate solution.

ii. *If IP gap* $> 10^{-4}$: CPLEX proceeds to ensure that the current solution value v is below below updated incumbent value proceeding according to:

   – **If** $v \geq$ **new incumbent**: CPLEX prunes this node, represented by the pass-through to cell K.
   – **If** $v <$ **new incumbent**: the current solution needs to be explored further and execution proceeds to cell F.

4.2.3 Solution Classification

Cell F

After ensuring a solution is viable to be explored (cell B) and possibly preforming a `Heuristic callback`, CPLEX evaluates the solution's integer feasibility to determine the appropriate callback continuation:

i. **If the solution is integer feasible**: specifically if $x_e \in 0, 1$ for all $e \in E$, CPLEX triggers the `Incumbent Callback` in cell I.

ii. **If solution is not integer feasible**: i.e. contains any fractional values $x_e \in (0, 1)$, CPLEX initiates the `User Cut Callback` in cell G.

4.2.4 Incumbent Callback

The `Incumbent Callback`, the most crucial component of our implementation, ensures that integer solutions violating some number of SECs (thus infeasible solutions to the TSP) are not accepted as incumbent solutions, thereby maintaining the integrity of our approach to solving the TSP. It is responsible for the exact separation of violated SECs from a given integer feasible solution.

Subtour Check & Constraint Addition (Cell I)

Upon identifying an integer feasible solution, CPLEX calls the `Incumbent Callback`, proceeding into cell I. Initially we implement the subtour identification routine for integer solutions detailed in Section 3.4.1. This generates a list wherein each entry corresponds to a unique subtour in the solution. Thus the subtour check proceeds according to:

i. **If the solution contains subtours**: i.e. if the length of the subtour list > 1; each with element in the list corresponds to a set of vertices S containing a subtour of length $|S| - 1$. Thus each element provides a set which constitutes a violated SEC. We add the SEC for each set of vertices in this list to the LP model. CPLEX then solves this *same subproblem* with the new constraints added, generating a solution which then proceeds to Cell B with processing continuing from Section 4.2.1.

ii. **If the solution contains no subtours**: i.e. if the length of the subtour list $= 1$, the `Incumbent Callback` ends and CPLEX proceeds to update the standing incumbent in cell J.

Standing Incumbent Updating (Cell J)

If no subtours are identified the solution is not only feasible but must also be an improvement over the current incumbent. To clarify this we notice that the solution is both integer feasible (as this precisely when the `Incumbent Callback` is executed) and violates no SECs as it contains no subtours. Furthermore the solution value must be lower than the current incumbent, i.e. $v < z^*$, as otherwise the node would have been pruned by the value comparison in cell B (or in cell E in the event that a `Heuristic Callback` was executed). Thus CPLEX will update the standing incumbent to the current solution and proceed to recalculate the IP Gap:

i. *If IP gap* $< 10^{-4}$: the new incumbent is within $10^{-4}\%$ of the optimal solution and the algorithm terminates providing $(x_e)_{e \in E}$ with value v as it's ultimate solution.

ii. *If IP gap* $\geq 10^{-4}$: CPLEX proceeds to cell K to prune the current node as it has yielded a feasible solution.

4.2.5 User Cut Callback

The `User Cut Callback` facilitates the addition of cutting planes at fractional solutions in the branch and bound tree. This is precisely the cutting-plane augmentation which transforms the branch and bound into a branch and cut algorithm. This callback implements the conditional SEC separation techniques for fractional solutions, as well as the separation of violated 2-matching inequalities.

Initially a check is preformed to identify all connected components of the fractional solution. This is done using a simple DFS algorithm returning a list wherein each element is a set of vertices constituting a connected component.

i. **If the solution is disconnected**: i.e. if this list has more than one component, we proceed to execute the Connected Component Separation procedure.

ii. **If the solution is connected**: i.e. if this list has exactly one element; we proceed to attempt a minimum cut separation procedure.

Connected Component (Cell G)

If the solution is disconnected we implement the connected component separation heuristic to identify and add violated SECs at low computational expense. The implementation is precisely as detailed in Section 3.4.1, wherein we use the set of vertices that constitute each connected component as a set over which to add an SEC. Given that each element of the list returned from the DFS algorithm is exactly such a set of vertices this entails simply adding the SEC over every set in the list. After adding each of these constraints to the LP relaxation execution proceeds to to check for violated 2-Matching constraints.

Minimum Cut (Cell G)

If the solution is connected we attempt a minimum cut separation routine. Specifically, we use the Stoer-Wagner (Algorithm 1) to identify the minimum cut value.

i. **If this cut has a value less than** 2: we use one of the vertex sets which the cut partitions to formulate a violated SEC. Specifically, we use the edges spanning the partition as those whose sum we constrain to be greater than or equal to 2 as these are exactly the edges in $\delta(S)$, wherein S is either of the sides of the partition. Once this constraint has been added execution proceeds to check for violated 2-Matching constraints.

ii. **If this cut has a value greater than** 2: there exist no violated SECs in this solution and we proceed to check for violated 2-Matching constraints.

2-Matchings (Cell G)

Here we implement the heuristic 2-Matching separation detailed in 3.4.2. Specifically, we filter the edges into fractional and integer sets proceeding to identify all connected components in the fractional support graph using the DFS algorithm. Then we identify potential teeth from the list of integer edges; swallowing any 'false teeth' into the handle set. If this process identifies any 2-Matchings with an odd number of teeth greater than 3 we add the 2-Matching constraint over each of these 2-Matchings. Execution then proceeds into cell H.

4.2.6 CPLEX Branch and Cut progression

Pruned Node progression (Cell K)

Upon pruning a node CPLEX preforms a check to discern if there are unsolved nodes remaining in the branch and bound tree:

i. **If there are unsolved nodes remaining:** CPLEX proceeds to select one of these unsolved nodes and solve its LP problem; generating a solution which then proceeds into the branch and bound checks of cell B as detailed in Section 4.2.1.

ii. **If there are no unsolved nodes remaining**: The search space has been exhaustively explored; meaning that the optimal solution must have been found, promoting CPLEX to terminate the algorithm and provide the standing incumbent as the ultimate solution.

Fractional solution progression (Cell H)

After completing a `User Cut Callback` CPLEX attempts to identify a set of violated cuts internally. If successful it adds these to LP model. A check is then preformed to identify if any constraints have been added in this iteration. That is if any cuts were added within the `User Cut Callback`, or by CPLEX internally:

i. **If any cuts were added**: CPLEX proceeds solves this *same subproblem* with the new constraints added, generating a solution which then proceeds to Cell B with processing continuing as detailed in Section 4.2.1.

ii. **If no cuts were added**: we have a fractional solution for which no cuts could be identified; thus necessitating a branch. CPLEX identifies a fractional variable x_e upon which to branch. This creates two new subproblems adding the constraint $x_e = 0$ to the first and the constraint $x_e = 1$ to the second. CPLEX then proceeds to select some unsolved subproblem (of which there must be at least 2) to solve; generating a solution which then proceeds to Cell B.

In this way the process described continues until some termination condition is met.

Chapter 5

Computational Results

This section outlines the computational results from applying our implementation in Chapter 4 to a library of test problems known as the TSPLIB. Created by Reinelt 1991 the TSPLIB, the premier collection of TSP test instances, provides a testing ground for evaluating algorithmic performance.

We initially compute 20 permutations of our implemented augmentations to CPLEX's core branch and cut algorithm for 3 distinctly sized TSPLIB instances. This provides an idea of how the different augmentations are affecting the solution process for differently sized problems. We then execute our full implementation on a set of 83 TSPLIB instances to observe performance on a wider range of problem sizes and topologies.

In every case, for brevity and feasibility, we enforced a one-hour time limit on all experiments, and set the values in the custom heuristic callback limiter to $a = 3$ and $b = 200$ (Section 4.2.2). Each test was conducted as the sole operation running on an Apple M3 MacBook Pro (16-inch, Nov 2023).

The tests recorded data in the form of the following set of metrics.

Table 5.1: Recorded Metrics

Metric		**Description**
IP Gap		The IP gap upon termination of the algorithm.
Opt. Gap (Optimality Gap)		The gap between the algorithm's ultimate upper bound and best-known solution to the TSPLIB instance.
Solve Time		Time (in seconds) spent executing the algorithm.
UB (Upper Bound)		The best upper bound (incumbent solution) found.
LB (Lower Bound)		The best lower bound found.
Branches		The number of branching operations performed by the algorithm.
Br. Gap (Branch Gap)		The IP gap when the the algorithm first branched. Empty if algorithm didn't branch or had no incumbent at first branch.
Incumbent	Cuts	The number of violated SECs added by the integer solution SEC separation in the `Incumbent Callback`.
	Time	Time (in seconds) spent identifying violated SECs within the `Incumbent Callback`.

Continued on next page

Table 5.1: Recorded Metrics (Continued)

Warmstart		Value	The objective value of the heuristic solution created in the warmstart process.
		Time	Time (in seconds) spent calculating the warmstart heuristic solution.
Heuristic			x/y: where x is the number of successful and y the total number of `Heuristic Callback` executions.
User Cut	Connected	Cuts	The number of violated SECs added using connected components separation in the `User Cut Callback`.
		Time	Time (in seconds) spent identifying violated SECs using connected component separation.
	Min-cut	Cuts	The number of violated SECs added using minimum cut separation in the `User Cut Callback`.
		Time	Time (in seconds) spent identifying violated SECs using the minimum cut SEC separation.
	2-Matching	Cuts	The number of violated 2-Matching constraints added in `User Cut Callback`.
		Time	Time (in seconds) spent identifying violated 2-Matching constraints in the.

5.1 Comparison of B&C Augmentations

We select `ch130`, `ali535`, and `u724` to represent small, medium, and large cases, respectively, within the context of our implementation's capabilities. The number in the name of each TSPLIB file represents the number of nodes (vertices) this instance is defined on.

In each case, we perform a test on a set of 20 permutations of the Connected Component SEC separation, Minimum Cut SEC separation, and 2-matching augmentations within the `User Cut Callback`, as well as the Warmstart and `Heuristic Callback` augmentations. In the tables in this section, each row represents a unique permutation of these augmentations. Specifically, a cell within the column of each augmentation is populated with data only if that augmentation was applied in that test.

We have omitted the Warmstart column from the tables due to redundancy, as all obtained warm start values are identical. Instead, we organize the entries such that the bottom 10 entries in each table represent those permutations including the Warmstart augmentation. It is important to note that the order of permutations in the bottom 10 rows is identical to that of the top 10 rows, with the addition of Warmstart in each iteration. We provide, for each TSPLIB instance, the warm start objective value and calculation time in the text, as the objective values were constant for all tests of an instance and the calculation times were within 0.1% of one another across all tests.

We then present the data for each of these instances and analyze the patterns within them thereafter.

`ch130`: Small Instance

In this case every instance solved to optimality, and thus we leave off both the IP Gap, and Opt. gap as these where both 0 for all instances. Similarly the upper and lower bounds where both exactly 6110 for every instance, which agrees with the best known solution, and thus we do not include either of these columns.

The warmstart values for this test are given by:

Objective value: 6856 **Calculation time:** 0.035 seconds.

Table 5.2: Comparison of augmentations for `ch130`.

Solve Time	Branches	Br. Gap	Incumbent		Heuristic	User Cut					
						Connected		Min-cut		2-Matching	
			Cuts	Time		Cuts	Time	Cuts	Time	Cuts	Time
0.509	33	3.126	46	0.176							
0.387	16	4.219	3	0.042		37	0.045				
0.7	4	26.36	14	0.108		37	0.008			27	0.043
0.644	4	0.844	4	0.055		37	0.031	7	0.242		
0.611	0		4	0.037		38	0.012	5	0.141	27	0.077
1.154	16	0.343	45	0.135	6/15						
0.723	4	0.642	4	0.039	5/9	37	0.027				
0.852	3	0.583	5	0.068	6/9	37	0.008			23	0.081
1.438	16	0.599	10	0.088	5/12	37	0.053	11	0.451		
1.116	4	0.373	1	0.027	7/11	39	0.012	8	0.234	21	0.032
Instances below used warmstart.											
0.707	5	1.666	51	0.184							
0.441	6	0.609	5	0.098		37	0.031				
0.589	8	3.469	8	0.114		38	0.011			31	0.065
0.658	4	0.844	4	0.063		37	0.03	7	0.247		
0.733	4	0.4	7	0.086		38	0.011	6	0.2	27	0.089
1.165	16	0.343	45	0.142	6/15						
0.72	4	0.642	4	0.051	5/9	37	0.03				
0.846	3	0.583	5	0.079	6/9	37	0.007			23	0.079
1.438	16	0.599	10	0.095	5/12	37	0.05	11	0.435		
1.145	4	0.373	1	0.039	7/11	39	0.011	8	0.248	21	0.032

`ali535`: Medium Instance

The best known solution in this instance is given by: 202339; which we identified in the 16 cases which solved to optimally. The warmstart values for this instance are:

Objective value: 231934 **Calculation time:** 1.167 seconds.

`u724`: Large Instance

The best known solution in this instance is given by: 41910; which we failed to reach in all cases. The warmstart values for this instance are:

Objective value: 45698 **Calculation time:** 2.81 seconds.

The data for each of these instances is presented on the following 2 pages in Tables 5.3 & 5.4 respectively. For compactness the warmstart indicator row is collapsed to a double line break.

Table 5.3: Comparison of augmentations for `ali535`.

IP Gap	Opt. Gap	Solve Time	UB	LB	Branches	Br. Gap	Incumbent		Heuristic	User Cut					
										Connected		Min-cut		2-Matching	
							Cuts	Time		Cuts	Time	Cuts	Time	Cuts	Time
0	0	1198.25	202339	202318.79	130434		493	27.11							
0	0	481.03	202339	202318.83	2967		65	7.83		271	441.64				
0.33	0.17	3611.79	202690	202020.18	150		43	4.04		185	125.93			2285	3382.84
0	0	526.77	202339	202319.51	1249		31	3.62		176	68.58	167	433.59		
0	0	418.11	202339	202330.63	348		33	3.85		167	109.95	153	219.81	429	60.08
0	0	2584.97	202339	202318.8	228224	7.37	347	17.49	11/45						
0	0	400.27	202339	202319	2976	1.35	2	1	5/22	235	302.25				
0.7	0.58	3606.39	203513	202079.09	54	1.76	7	0.22	8/24	177	47.69			965	3484.57
0	0	781.34	202339	202319.86	1692	1.32	3	1.97	4/15	172	153.75	196	565.01		
0	0	184.81	202339	202318.97	178	1.21	5	1.73	5/18	157	0.75	85	103	237	17.67
0	0	759.45	202339	202318.92	101660	13.77	343	18.02							
0	0	696.99	202339	202319	7240	13.17	29	4.37		299	638.25				
13.15	12.76	3600.52	231934	201438.05	0		0	0.17		144	1.51			2077	991.47
0	0	1290.36	202339	202318.83	3552	13.08	29	4.28		175	243.73	269	1006.56		
0	0	434.74	202339	202335.25	420	12.94	12	3.91		158	43.09	221	276.79	651	80.85
0	0	2561.98	202339	202318.8	228224	7.37	347	17.52	11/45						
0	0	369.84	202339	202319	2976	1.35	2	1.16	5/22	235	274.5				
0.7	0.58	3604.93	203513	202079.09	54	1.76	7	0.38	8/24	177	42.37			960	3489.51
0	0	773.97	202339	202319.86	1692	1.32	3	2.15	4/15	172	148.88	196	562.13		
0	0	184.78	202339	202318.97	178	1.21	5	1.92	5/18	157	0.74	85	102.5	237	17.45

Table 5.4: Comparison of augmentations for `u724`.

IP Gap	Opt. Gap	Solve Time	UB	LB	Branches	Br. Gap	Incumbent		Heuristic	User Cut					
										Connected		Min-cut		2-Matching	
							Cuts	Time		Cuts	Time	Cuts	Time	Cuts	Time
0.32	0.07	3600.02	41941	41807.08	293103		709	45.16							
0.53	0.35	3600.1	42059	41837.93	11120		95	16.04		696	3303.63				
3.75	3.57	3600.18	43461	41830.3	138		34	3.62		190	475.77			972	2990.12
1.26	1.05	3611.08	42356	41821.71	938		35	6.8		184	507.15	467	3022.18		
0.2	0.1	3614.02	41954	41870.43	1297		19	3.97		161	2.27	484	2911.09	776	612.58
0.96	0.59	3600.06	42158	41752.83	39800	6.84	446	21.82	12/1896						
0.93	0.8	3600.02	42247	41852.22	7026	2.85	12	1.11	11/352	568	2915.68				
1.31	1.11	3609.69	42381	41826.17	103	3.16	7	0.42	9/26	202	141.2			986	3357.33
0.5	0.31	3620.29	42042	41832.79	1429	2.88	11	0.97	12/89	160	227.08	458	3204.12		
0.11	0.04	3614.87	41926	41879.55	1894	2.44	0	1.63	9/92	147	9.02	484	2904.76	808	476.4
0.24	0.06	3600.1	41936	41835.75	313700	9.17	563	42.3							
0.47	0.21	3600.39	41998	41801.76	9806	8.68	73	13.49		627	3340.05				
1.08	0.83	3614.45	42262	41807.66	169	8.56	32	2.79		181	955.91			658	2611.93
1.2	0.94	3600.76	42308	41799.41	1122	8.6	11	3.35		175	155.52	449	3390.21		
0.73	0.55	3601.29	42140	41830.27	658	8.52	1	1.71		140	1.44	549	3195.09	774	306.01
0.97	0.59	3600.5	42158	41749.83	37830	6.84	441	22.27	12/1804						
0.94	0.8	3624.46	42247	41850.37	6048	2.85	12	1.49	11/306	541	3012.87				
1.31	1.11	3606.59	42381	41826.17	103	3.16	7	0.76	9/26	202	159.12			977	3331.29
0.5	0.31	3600.23	42042	41832.78	1298	2.88	11	1.39	12/83	159	230.25	430	3182.28		
0.12	0.04	3614.23	41926	41877.32	1615	2.44	0	2.07	9/80	147	8.84	467	2903.22	770	491.93

5.1.1 Analysis

The data collected from our experiments presents an encouraging perspective on the effectiveness of the augmentations implemented in our study. To grasp the impact of these enhancements at a high level, we recommend prioritizing the examination of solve times, particularly for the instances `ch130` and `ali535`. These instances offer the most straightforward insights into the efficacy of different augmentations on the solution process.

For the larger instance, `u724`, given that none of these instances reached optimality within the allotted time, solve times offer limited value for comparative analysis. Instead, focusing on the IP gap or the Optimality gap provides more meaningful information. These metrics shed light on the relative performance of augmentation permutations, offering insights into the algorithm's efficiency in narrowing down the search for an optimal solution under time constraints.

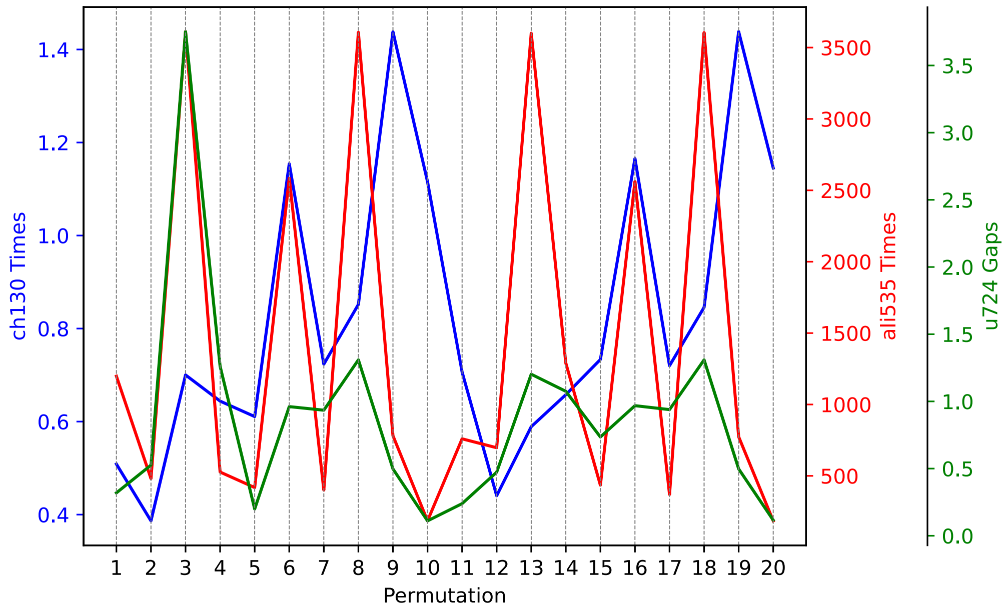


Figure 5.1: A comparison of key metrics for each TSPLIB instance across all permutations.

Selected key metrics for each instance across all 20 permutations tested. The number of the permutations here corresponds canonically to the order of permutations given by the rows in each of the tables above.

We notice crucially that for both the medium `ali5353` and large `u724` the optimum permutations seem to be both 10 and 20, corresponding to the permutations wherein all augmentations (except warmstart in permutation 10) were included. This at the very least verifies that for somewhat challenging TSP instances our augmentation are indeed beneficial.

Warmstart

We begin by noticing that efficacy of the warmstart seems to disappear when the **`heuristic callback`** is implemented especially for larger TSP instances.

Figure 5.2 displays the percentage benefit in key metrics from applying the warmstart augmentation on top of the base 10 permutations. Most notably the latter 5 permutations show near zero or negative improvements across all instances - suggesting that in any permutation wherein the heuristic callback is applied; the warmstart augmentation offers no benefits. This can viewed as a result of our implemented warmstart strategy building

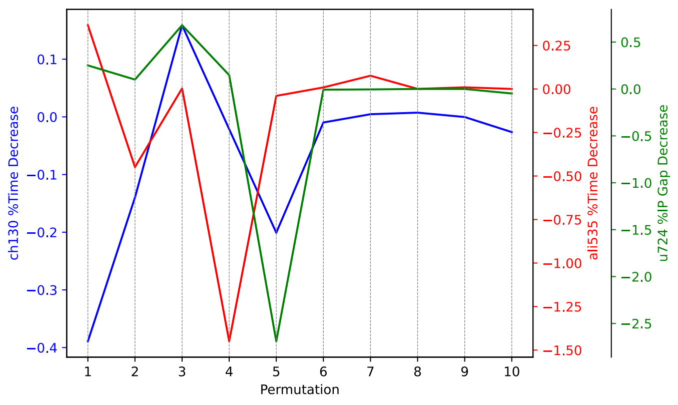


Figure 5.2: Percentage decrease in key metrics for each TSPLIB instance between permutations with and without warmstart.

heuristic solutions in a very similar manner to the heuristic callback. Since the heuristic callback is typically executed at the root node; this implies that the quality of solution provided by the warmstart is quickly matched if not improved by the heuristic callback - rendering this method of warmstart practically useless in these cases.

2-Matchings

We observe that the inclusion of 2-matching and connected components augmentations, when not complemented by the minimum cut augmentation, leads to significantly less effective solution strategies for the more challenging instances. Specifically, the permutations corresponding to entries 3, 8, 13, and 18 in Tables 5.3 and 5.4 demonstrate inferior performance, exhibiting wider IP Gaps and longer solution times compared to practically all other permutations examined. A detailed examination reveals that the 2-Matching separation primarily contributes to these shortcomings, with computation times reaching 3000 seconds in several instances. This observation indicates that our implemented 2-matching separation, when applied in isolation, stands as a markedly inefficient augmentation. However we do notice significant improvements when the 2-matching are paired with the minimum cut separation; suggesting that in tandem these augmentations complement each other ultimately serving as an effective cutting plane in the branch and cut solution process

5.2 Large subset of TSP Data

For completeness and reference we ran our full implementation on a large selection of instances of the TSPLIB subject to the same restrictions and parameters as the previous section. In this data we include each instances name as well as the specific warmstart values for all attempts. The data for the test is given in the Appendix in Section 6.2.

Chapter 6

Conclusion

6.1 Key findings

In this paper we have explored an exact solution strategy to the TSP with a set of supplementary augmentations to this solution process. Ultimately our s affirm that this solution strategy constitutes a viable exact strategy for the TSP and that when applied in conjunction, each of our augmentations do indeed aid the solution process - as is expected given the widespread attention these augmentations have received. We hope to have accurately detailed the complexities of the TSP, and put forward a viable solution strategy.

6.2 Future Work

Initially we recognise room for a much deeper and more insightful analysis of results. This would potentially unlock the full benefits of the work done in this report.

We further recognize the vast potential for improvements in our implementation and understanding of the TSP. The immediate next step involves implementing parallel processing within our branch and cut solution; allowing multiple subproblems to be explored simultaneously offering huge potential improvements over our current results. Furthermore, the improvement and benchmarking of our custom graph searching algorithms are essential to verify that our findings are indeed true representations of the mathematical challenges of the TSP and not skewed by idiosyncratic inefficiencies. There exists a range of widely used graph traversal algorithms that could be used to rigorously benchmark against.

Additionally, we would like to investigate different classes of cutting-planes and employ new separation procedures. Specifically attempting to implement separations of the general comb inequalities and perhaps even the further generalized clique tree inequalities to improve on our current 2-Matching implementation.

Our current implementation of heuristics is itself very 'heuristic' in the sense that we understand very little about which heuristics should be used at different points in the branch and bound tree and how often these heuristic improvements should be attempted. This is exemplified by the high heuristic failure rates throughout the collected data. Large scale tests on the speed and quality of different heuristics across TSPLIB problems may help provide some more insight as to the optimal manner of heuristic integration.

Looking further, the integration of machine learning presents an exciting frontier. There seems to be potential for machine learning algorithms to arrive at an some sort of understanding of the TSP polytope where they can be reliably used to decide which cuts to add and at what points in the branch and cut algorithm to add these cuts. These implementations have shown promise of optimising the path towards an optimal solution within the branch and cut algorithms (see Tang, Agrawal, and Faenza 2020 and VO et al. 2023).

Beyond the confines of TSP, we would like to explore the realm of routing and matching problems in order to see if any of our insights from the TSP are applicable in these realms and perhaps also to arm us with new tactics with which to tackle the TSP.

Bibliography

Applegate, D. L. et al. (1998). "On the Solution of Traveling Salesman Problems". In: *Documenta Mathematica* Extra Volume ICM III, pp. 645–656.

Applegate, David, Robert Bixby, and Václav Chvátal (2006). *Concorde TSP Solver*. http://www.math.uwaterloo.ca/tsp/concorde/index.html. Accessed: 2024-03-18.

Applegate, David L. et al. (2007). *The Traveling Salesman Problem*. Princeton University Press. ISBN: 9780691129938.

Chvátal, Václav (1973). "Edmonds Polytopes and Weakly Hamiltonian Graphs". In: *Mathematical Programming* 5. Received 23 February 1972; Revised manuscript received 8 March 1973, pp. 29–40.

Cook, Stephen A. (1971). "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

Cornuejols, Gérard, Jean Fonlupt, and Denis Naddef (July 1984). *The Traveling Salesman Problem on a Graph and Some Related Integer Polyhedra*. Tech. rep. MSRR 510. This report was prepared in part as part of the activities of the Management Sciences Research Group Carnegie-Mellon University under Contract No. N00014-82-K-0329 NR 047-048 with the U.S. Office of Naval Research and in part by an NSF grant ECS-8205425. Pittsburgh, Pennsylvania: Carnegie-Mellon University, Graduate School of Industrial Administration.

CPLEX: HeuristicCallback (2020). 20.1.0. IBM. URL: https://www.ibm.com/docs/en/icos/20.1.0?topic=classes-cplexcallbacksheuristiccallback.

Dantzig, George B., Delbert R. Fulkerson, and Selmer M. Johnson (1954). "Solution of a Large-Scale Traveling-Salesman Problem". In: *Journal of the Operations Research Society of America* 2, pp. 393–410.

Edmonds, Jack (1965). "Maximum Matching and a Polyhedron With O1-Vertices". In: *Journal of Research of the National Bureau of Standards-B. Mathematics and Mathematical Physics* 69B.1 and 2. Received December 1, 1964, pp. 125–130.

Fiorini, Samuel et al. (2015). *Exponential Lower Bounds for Polytopes in Combinatorial Optimization*. arXiv: 1111.0837 [math.CO].

Gasarch, William I. (2019). "The Third P =? NP Poll". In: *SIGACT News Complexity Theory Column*. https://blog.computationalcomplexity.org/2018/12/ker-i-ko-1950-2018.html.

Gearhart, Jared Lee et al. (Oct. 2013). *Comparison of Open-Source Linear Programming Solvers*. Tech. rep. Accessed: 2024-03-18. Albuquerque, New Mexico: University of North Texas Libraries, UNT Digital Library. URL: https://digital.library.unt.edu/ark:/67531/metadc868834/.

Grötschel, M. and W. R. Pulleyblank (1986). “Clique Tree Inequalities and the Symmetric Travelling Salesman Problem”. In: *Mathematics of Operations Research* 11.4, pp. 537–569. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3690002 (visited on 03/13/2024).

Grötschel, Martin and Manfred Padberg (1979). “On the Symmetric Travelling Salesman Problem I: Inequalities”. In: *Mathematical Programming* 16.1, pp. 265–280.

Gutekunst, Samuel C. and David P. Williamson (Dec. 2020). “The Circlet Inequalities: A New, Circulant-Based Facet-Defining Inequality for the TSP”. In: *arXiv preprint*. Available at https://arxiv.org/abs/2012.12363. eprint: arXiv:2012.12363.

Hwang, C-P, B Alidaee, and J D Johnson (1999). “A Tour Construction Heuristic for the Travelling Salesman Problem”. In: *Journal of the Operational Research Society* 50.8, pp. 797–809. DOI: 10.1057/palgrave.jors.2600761. URL: https://doi.org/10.1057/palgrave.jors.2600761.

IBM (2022). *ILOG CPLEX Optimization Studio Documentation*. https://www.ibm.com/docs/en/icos. Version: 22.1.1.

Johnson, D.S. and L.A. McGeoch (Nov. 1995). “The Traveling Salesman Problem: A Case Study in Local Optimization”. In: *Not specified*. URL: https://www.cs.ubc.ca/~hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf.

— (2007). “Experimental Analysis of Heuristics for the STSP”. In: *Combinatorial Optimization*, pp. 369–443. DOI: 10.1007/0-306-48213-4_9. URL: https://doi.org/10.1007/0-306-48213-4_9.

Lawler, E. L. et al. (1985). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley.

Letchford, Adam, Gerhard Reinelt, and Dirk Oliver Theis (June 2004). “A Faster Exact Separation Algorithm for Blossom Inequalities”. In: vol. 3064, pp. 196–205. ISBN: 978-3-540-22113-5. DOI: 10.1007/978-3-540-25960-2_15.

Letchford, Adam N. and Andrea Lodi (2002). “Polynomial-Time Separation of Simple Comb Inequalities”. In: *Proceedings of the International Conference on Integer Programming and Combinatorial Optimization (IPCO)*. Ed. by William J. Cook and Andreas S. Schulz. Vol. 2337. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 93–108. URL: https://www.lancaster.ac.uk/staff/letchfoa/other-publications/2002-IPCO-comb.pdf.

Naddef, D. and G. Rinaldi (1992). “The Crown Inequalities for the Symmetric Traveling Salesman Polytope”. In: *Mathematics of Operations Research* 17.2, pp. 308–326. ISSN: 0364765X, 15265471. URL: http://www.jstor.org/stable/3689960 (visited on 03/13/2024).

Padberg, M. W. and G. Rinaldi (1991). “A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Travelling Salesman Problems”. In: *SIAM Review* 33, pp. 60–100.

Pegden, Wesley and Anish Sevekari (Mar. 2023). “Comb Inequalities for Typical Euclidean TSP Instances”. In: *arXiv.org* arXiv:2012.00292. URL: https://arxiv.org/abs/2012.00292.

Reinelt, Gerhard (1991). “TSPLIB—A Traveling Salesman Problem Library”. In: *ORSA Journal on Computing* 3.4, pp. 376–384. DOI: 10.1287/ijoc.3.4.376. URL: https://doi.org/10.1287/ijoc.3.4.376.

Rothvoß, Thomas (Nov. 2013). "The Matching Polytope Has Exponential Extension Complexity". In: *Proceedings of the Annual ACM Symposium on Theory of Computing*. DOI: 10.1145/2591796.2591834.

Stoer, Mechthild and Frank Wagner (July 1997). "A Simple Min-Cut Algorithm". In: *J. ACM* 44.4, pp. 585–591. ISSN: 0004-5411. DOI: 10.1145/263867.263872. URL: https://doi.org/10.1145/263867.263872.

Tang, Yunhao, Shipra Agrawal, and Yuri Faenza (2020). *Reinforcement Learning for Integer Programming: Learning to Cut*. arXiv: 1906.04859 [cs.LG]. URL: https://arxiv.org/abs/1906.04859.

VO, Thi Quynh Trang et al. (2023). "Improving Subtour Constraints Generation in Branch-and-Cut Algorithms for TSP with Machine Learning". In: *ROADEF2023*. LIMOS, Université Clermont Auvergne and UM-SJTU Joint Institute, Shanghai Jiao Tong University. Clermont-Ferrand, France.

Wilson, Robin J. (2012). *Introduction to Graph Theory*. 5th. Prentice Hall. ISBN: 9780273728894.

Appendix A

Implementation Code

For those interested in the practical implementation of our findings, the complete source code is available on GitHub. This repository includes all the scripts, TSPLIB files and detailed instructions necessary to replicate our experiments and explore the application of the branch-and-cut algorithm to solve instances of the TSP.

GitHub Repository: https://github.com/RubenMitchell/TSP_rebuild

Please refer to the README file within the repository for installation and execution guidelines.

TSPLIB data

Table 6.1: TSPLIB data.

Name	Optimal	Op Gap	MIP Gap	Solve Time	UB	LB	Nodes	Gap*	Incumbent		Warmstart		Heuristic	User Cut					
														Connected		Min-cut		2-Matching	
									Cuts	Time	Value	Time		Cuts	Time	Cuts	Time	Cuts	Time
a280	2579	0	0	9.8	2579	2579	0		0	0.1	2819	0.25	4/6	48	5.72	3	0.17	48	0.09
ali535	202339	0	0	221.6	202339	202326.42	75	1.42	1	1.25	231934	1.26	5/8	168	75.41	87	92.91	301	26.52
att48	10628	0	0	0.09	10628	10628	0		9	0.01	11918	0	3/4	6	0	4	0.01	7	0
att532	27686	0	0	1116.8	27686	27683.24	714	3.25	8	0.95	29876	1.46	10/24	118	174.53	275	560.45	578	304
berlin52	7542	0	0	0.02	7542	7542	0		0	0	8113	0	1/1	3	0	0	0	1	0
bier127	118282	0	0	0.53	118282	118282	0		1	0.03	124858	0.04	3/4	25	0.08	2	0.02	16	0.01
burma14	3323	0	0	0.01	3323	3323	0		2	0	3749	0	0/0	0	0	0	0	0	0
ch130	6110	0	0	1.11	6110	6110	3	2.25	1	0.03	6856	0.04	5/6	39	0.01	9	0.23	21	0.03
ch150	6528	0	0	2.99	6528	6528	4	1.85	5	0.05	6974	0.06	1/5	32	0.35	17	0.41	27	0.09
d1291	50801	2.51	3.21	3677.66	52107	50432.29	27	3.25	0	1.11	55462	14.27	7/25	117	1.9	274	2794.54	401	262.38
d1655	62128	4.77	5.33	4721.75	65241	61764.71	0		0	1.86	67811	25.81	6/11	271	4416.52	2	17.6	276	3.24
d198	15780	0	0	11.67	15780	15780	37	1.64	3	0.24	16260	0.12	2/11	94	4.36	30	3.17	79	0.41
d2103	80450	1.85	3.23	15344.22	81967	79318.32	0		0	2.85	82424	56.19	1/4	93	15185.81	0	0	72	2.54
d493	35002	0	0	448.01	35004	35000.5	594	2.18	16	1.59	37545	1.08	6/11	187	21.91	407	308.27	657	79.3
d657	48912	0.3	0.55	3361.54	49059	48788.54	1222	2.33	1	2	53796	2.23	7/17	156	511.69	659	2371.54	950	349.11
dsj1000	18659688	1.5	1.68	3600.81	18943223	18625466	107	2.33	0	0.67	20743178	8.39	10/25	296	17.75	251	2859.01	568	369.29
eil101	629	0	0	0.22	629	629	0		3	0.03	699	0.02	1/2	6	0	2	0.04	8	0.01
eil51	426	0	0	0.06	426	426	0		0	0	486	0	1/3	3	0	4	0.01	7	0
eil76	538	0	0	0.08	538	538	0		3	0.02	602	0.01	2/2	1	0	0	0	6	0
fl1400	20127	2.62	4.38	4903.65	20669	19762.87	0		0	1.27	23319	17.17	6/24	463	2325.23	10	2148.79	927	11.02
fl1577	22249	5.96	10.34	3931.94	23660	21214.24	0		16	3.85	24100	23.94	3/11	292	1890.99	1	1828.06	344	4.86
fl3795	28772	3.93	8.08	3606.79	29948	27527.59	0		0	10.05	30568	310.97	2/9	408	2290.2	0	0	399	9.04
fl417	11861	0	0.03	3600.06	11861	11857	141491	2.91	26	1.85	12960	0.69	24/317	305	141.1			1370	2272.48
fnl4461	182566	4.87	5.42	19498.21	191914	181504.67	0		0	14	201638	415.05	2/2	411	18292.45	0	0	340	3.92

Table 6.2: TSPLIB data.

Name	Optimal	Op Gap	MIP Gap	Solve Time	UB	LB	Nodes	Gap*	Incumbent		Warmstart		Heuristic	User Cut					
														Connected		Min-cut		2-Matching	
									Cuts	Time	Value	Time		Cuts	Time	Cuts	Time	Cuts	Time
gil262	2378	0	0	26.41	2378	2378	139	0.93	6	0.3	2696	0.3	7/10	73	0.87	86	17.84	141	2.2
gr137	69853	0	0	2.45	69853	69853	8	0.3	3	0.09	73344	0.05	2/8	40	0.7	12	0.31	28	0.05
gr202	40160	0	0	5.3	40160	40158.12	14	0.17	8	0.33	45456	0.1	6/7	39	0.84	29	2.13	38	0.21
gr229	134602	0	0	38.73	134602	134589.94	546	0.65	5	0.38	141208	0.15	2/8	60	0.14	127	27.49	164	5.04
gr431	171414	0	0	672.34	171414	171396.95	1903	2.14	14	3.18	187301	0.9	9/16	105	31.26	271	489.27	473	97.08
gr666	294358	0	0	723.78	294358	294332.14	194	1.37	1	1.15	327908	2.42	5/10	145	5.66	160	587.31	309	86.48
gr96	55209	0	0	1.59	55209	55204.96	83	1.98	15	0.08	61473	0.04	3/8	19	0.11	26	0.75	49	0.16
kroA100	21282	0	0	0.44	21282	21282	2	0.37	6	0.03	24726	0.02	2/3	21	0	6	0.06	21	0.01
kroA150	26524	0	0	6.14	26524	26524	105	1.66	5	0.16	28532	0.06	2/9	34	0.1	62	3.67	70	0.46
kroA200	29368	0	0	17.01	29368	29365.24	146	1.1	2	0.1	31925	0.11	3/8	61	1.95	81	7.15	156	3.12
kroB100	22141	0	0	1.04	22141	22141	14	2.29	13	0.06	24074	0.02	4/9	29	0.06	18	0.31	39	0.04
kroB150	26130	0	0	6.62	26130	26130	34	2.04	7	0.06	31000	0.05	7/11	35	0.54	45	2.03	84	0.32
kroB200	29437	0	0	3.94	29437	29437	3	0.04	3	0.17	33703	0.11	6/7	54	0.32	11	0.95	36	0.08
kroC100	20749	0	0	0.74	20749	20749	0		3	0.02	22908	0.03	2/8	25	0.16	8	0.06	19	0.01
kroD100	21294	0	0	0.61	21294	21294	0		3	0.04	22560	0.02	5/7	26	0.01	7	0.09	22	0.03
kroE100	22068	0	0	1.2	22068	22068	18	2.9	2	0.03	23676	0.03	5/10	23	0	18	0.25	34	0.07
lin105	14379	0	0	0.24	14379	14379	0		4	0.02	15654	0.03	2/4	26	0	0	0	12	0
lin318	42029	0	0	32.86	42029	42028	52	1.55	14	0.57	45968	0.33	7/12	117	4.12	55	14.07	80	2.1
nrw1379	56638	3.29	3.71	3993.66	58562	56391.36	0		0	1.21	60943	16.81	2/3	152	3936.83	0	0	115	0.61
p654	34643	0	0	134.51	34643	34643	0		2	1.51	36425	2.18	1/11	286	39.16	4	63.16	121	0.97
pcb1173	56892	2.14	2.48	3820.29	58138	56693.57	0		0	0.88	63140	11.29	5/6	136	3556.01	7	171.1	194	2.5
pcb3038	137694	3.03	3.4	3621.56	141990	137164.9	0		0	6.03	150141	152.75	4/6	287	2664.31	2	29.6	416	10.24
pcb442	50778	0	0	127.21	50778	50773.5	167	1.11	37	4.89	57294	0.8	6/7	65	19.94	104	81.11	187	5.83

Table 6.3: TSPLIB data.

Name	Optimal	Op Gap	MIP Gap	Solve Time	UB	LB	Nodes	Gap*	Incumbent		Warmstart		Heuristic	User Cut					
														Connected		Min-cut		2-Matching	
									Cuts	Time	Value	Time		Cuts	Time	Cuts	Time	Cuts	Time
pr1002	259054	4.18	4.71	4038.46	270351	257618.66	0		0	0.66	277453	7.83	4/7	255	3984.58	0	0	190	0.91
pr107	44303	0	0	0.33	44303	44303	0		68	0.1	44625	0.02	0/3	4	0.01	3	0.02	2	0
pr124	59030	0	0	1.54	59030	59030	13	0.43	13	0.11	62437	0.04	3/7	25	0.06	19	0.31	22	0.05
pr136	96772	0	0	1.75	96772	96772	11	2.67	3	0.04	105679	0.05	5/8	31	0.11	16	0.46	80	0.18
pr144	58537	0	0	1.57	58537	58535.33	3	0.47	25	0.13	61244	0.04	2/7	36	0.08	10	0.23	17	0.04
pr152	73682	0	0	3.82	73682	73682	7	0.73	56	0.1	75285	0.06	0/8	80	0.52	13	0.36	53	0.13
pr226	80369	0	0	10.84	80369	80369	0		18	0.35	82211	0.13	2/13	95	1.51	21	1.82	71	0.21
pr2392	378032	5.5	7.39	13141.14	400028	370482.5	0		0	4	411170	72.56	2/3	308	12929.78	0	0	137	1.14
pr264	49135	0	0	6.37	49135	49135	0		1	0.18	52996	0.21	2/4	39	2.24	6	0.74	29	0.08
pr299	48191	0	0	240.74	48191	48186.25	1598	3.33	3	0.95	51412	0.31	8/32	81	18.73	299	138.58	583	49.54
pr439	107217	0	0	625.43	107217	107211.92	2598	2.33	8	1.09	114673	0.77	5/13	97	21.18	313	491.44	498	62.66
pr76	108159	0	0	3.52	108159	108152.54	426	2.75	7	0.05	128999	0.01	5/13	16	0.03	73	1.62	147	0.35
rat195	2323	0	0	19.54	2323	2323	235	2.31	7	0.15	2632	0.12	4/10	10	0.09	98	14.14	69	1.56
rat575	6773	0.01	0.05	3600.77	6774	6770.95	4827	1.71	15	3.27	7418	1.53	5/10	77	31.69	538	3080.24	518	370.46
rat783	8806	0	0	1884.93	8806	8806	186	1.5	7	2.42	9701	3.43	8/10	138	732.97	186	1021.02	220	53.67
rat99	1211	0	0	0.44	1211	1211	0		4	0.04	1356	0.02	3/4	7	0	6	0.08	8	0.01
rd100	7910	0	0	0.4	7910	7910	0		4	0.03	8675	0.02	3/6	26	0	5	0.03	16	0.01
rd400	15281	0	0	356.66	15281	15279.48	1740	2.14	5	0.91	16303	0.64	6/10	89	4.6	209	288.03	244	36.41
rl1304	252948	3.77	4.28	3600.6	262868	251604.19	0		0	1.12	283496	15.18	3/15	284	2612.18	75	557.67	390	101.45
rl1323	270199	5.51	7.63	4092.3	285956	264129.4	0		0	1.19	293747	15.57	4/7	236	3988.13	0	0	96	0.8
rl1889	316536	4.95	8	6018.93	333009	306360.47	0		0	2.45	347944	38.33	4/5	317	5790.32	0	0	143	1.19
rl5915	565530	5.8	8.75	5584.23	600360	547852.58	0		0	25.94	614348	911.31	2/2	411	2821.79	0	0	134	5.37
rl5934	556045	4.76	7.33	7769.83	583827	541023.95	0		0	24.61	601514	965.94	2/3	549	4875.4	0	0	195	6.86

Table 6.4: TSPLIB data.

Name	Optimal	Op Gap	MIP Gap	Solve Time	UB	LB	Nodes	Gap*	Incumbent		Warmstart		Heuristic	User Cut					
														Connected		Min-cut		2-Matching	
									Cuts	Time	Value	Time		Cuts	Time	Cuts	Time	Cuts	Time
st70	675	0	0	0.12	675	675	0		14	0.01	793	0.01	2/3	5	0	3	0.01	8	0
ts225	126643	0.19	2.67	3600.09	126879	123492.33	44819	5.21	2	0.26	133425	0.15	4/241	121	12.46	3274	2261.59	923	328.61
tsp225	3916	0	0	20.04	3916	3916	67	2.66	25	0.47	4149	0.16	3/12	39	2.64	73	10.47	100	2.47
u1060	224094	2.37	2.58	3601.44	229539	223623.73	34	3.06	0	0.73	252661	7.81	8/27	282	1470.19	156	1386.58	531	437.74
u1432	152970	3.97	4.12	4696.02	159300	152729.92	0		0	1.36	166800	18.53	2/5	199	4603.29	0	0	272	1.88
u159	42080	0	0	1.93	42080	42078.29	0		2	0.07	48160	0.07	3/5	30	0.71	5	0.24	17	0.03
u1817	57201	5.42	6.83	7434.87	60476	56343.1	0		0	2.25	64471	36.52	3/3	204	7293.22	0	0	42	1.05
u2152	64253	2.85	3.4	7773.65	66141	63892.39	0		0	3.03	69861	54.08	4/7	329	558.57	1	6830.68	204	14.05
u2319	234256	2.22	2.24	8401.63	239579	234215	0		0	3.5	246193	66.2	2/2	102	8165.29	0	0	77	1.69
u574	36905	0	0	2104.22	36905	36905	1124	1.4	3	3.07	39603	1.56	6/14	158	80.05	520	1261.53	1108	643.95
u724	41910	0.04	0.12	3600.35	41928	41878.22	1100	2.38	11	3.46	45698	2.8	9/16	155	27.05	400	3099.79	568	360.39
ulysses16	6859	0	0	0.01	6859	6859	0		4	0	7580	0	1/1	1	0	0	0	2	0
ulysses22	7013	0	0	0.02	7013	7013	0		6	0	8114	0	1/1	3	0	0	0	0	0
vm1084	239297	4.78	5.75	4283.47	251314	236858.82	0		0	0.77	258292	8.83	3/6	192	4230.19	0	0	148	0.65
vm1748	336556	3.81	4.43	5413.68	349896	334404.84	0		0	2.07	362679	34.2	6/20	305	700.18	36	3956.82	496	166.27