Solving the LP Relaxation of Distance-Constrained Vehicle Routing Problem Using Column Generation

by

Seyyed Sina Dezfuli

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Seyyed Sina Dezfuli, 2020

Abstract

The distance-constrained vehicle routing problem (DVRP) is one of the less studied variants of vehicle routing problems. Here, the objective is to deliver packages from a depot to clients with as few delivery vehicles as possible within a given time frame.

In this thesis, we tackle larger instances of DVRP with around 200 nodes using column generation. We utilize a recent 3-approximation algorithm for the rooted orienteering problem to provide a practical algorithm for generating columns to approximately solve an exponentially large LP-relaxation of the DVRP. We also provide a proof-of-concept implementation, analyze its performance, extract the practical bounds and compare them against the theoretical ones. Finally, we measure how much improvement our algorithm provides on top of more naive heuristics that can be used for generating columns.

Acknowledgements

Special thanks to Professor Zachary Friggstad without whom this thesis would not have been possible.

Table of Contents

1 Introduction **1**
1.1 Preliminaries and Notations 1
1.2 Motivation and Literature Review 5

2 Column Generation for Linear Programs **9**
2.1 An Example of Column Generation: Solving the Maximum Multi-Commodity Flow Problem 15
2.2 Column Generation Using Approximation Algorithms 22

3 Approximately Solving The Orienteering Subproblem **24**
3.1 From Prize-Collecting Arborescence to Orienteering Path . . . 25
3.2 Pseudocode 40
3.3 Post-Processing 43
3.4 Time Complexity 43

4 Putting It All Together: Approximately Solving The LP Relaxation of DVRP **45**
4.1 Hardness of The LP Relaxation 45
4.2 Heuristics 46
4.3 Algorithm 48
4.4 Bounds for DVRP 50

5 Experiments **51**
5.1 Datasets 51
5.2 Results 52
5.2.1 Improvement 52
5.2.2 Approximation Factor 53
5.2.3 Running Time 61
5.3 Possible Optimizations 63

6 Conclusion **64**

References **65**

Appendix A Solving DVRP Integer Program **67**
A.1 Integrality Gap 67
A.2 Objective Function Values and Lower Bounds 68
A.3 Path Visualizations 74

Appendix B Google Maps Visualizations **85**

List of Tables

5.1 Statistics for Improvement 53
5.2 Statistics for the approximation factor (U) 54

A.1 Statistics for Integrality Gap 68

List of Figures

1.1 An example of an arborescence 4

2.1 Feasible region of (2.3) 11
2.2 The edge weights denote capacities 17
2.3 The edges numbered 17
2.4 The initial dual variables 18
2.5 The shortest path between s_1 and t_1 19
2.6 The dual variables after generating the sixth column 20
2.7 The shortest path between s_2 and t_2 20
2.8 The final dual variables after generating the seventh column . 22

3.1 A sample arborescence. Note that in this graph, t is the furthest node from the root (r) 28
3.2 The solid arcs of the arborescence (T) in blue and the dashed arcs of A in red 29
3.3 One of the possible walks of the graph in fig. 3.2 30
3.4 Path extracted by shortcutting past repeated nodes 30
3.5 The dashed and solid lines indicate paths and edges, respectively. 31
3.6 Segmentation of a path 35

5.1 The Distribution of Improvement 53
5.2 The Distribution of the approximation factor 54
5.3 Distribution of the final lower bound (LB) versus the final objective function value (V_f). Each dot represents a dataset . . . 55
5.4 How the objective function value and the lower bound change over the iterations of column generation 61
5.5 Average orienteering runtime versus the number of clients. Each dot represent a data set 62

A.1 The Distribution of Integrality Gap 68
A.2 How the objective function value of (1.8) and the lower bound change over the course of column generation iterations. The green dots denote the integer solution to (1.7) 73
A.3 Visualizations of paths generated by Integer Program Solver. The depot is marked by a red x 84

B.1 Visualization of paths generated by the algorithm on real Google Maps data. 87

List of Algorithms

2.1 Simplex Algorithm 13
3.1 Extract path from arborescence 31
3.2 Greedy Splitting 34
3.3 Binary Search Orienteering 41
3.4 Binary Search 42
3.5 Pruning Algorithm 43
3.6 Post processing step 43
4.1 First heuristic 47
4.2 Second heuristic 47
4.3 Third heuristic 48
4.4 Column Generation for Distance-Constrained Vehicle Routing Problem 49

Symbols

U
: DVRP Approximation Factor

T
: Arborescence

D
: Distance limit

W
: Eulerian walk

LB
: DVRP Lower Bound

P^*
: Optimal Orienteering Path

π^*
: Optimal Orienteering Reward

r
: Root node

P
: Path

Abbreviations

CVRP
Capacitated Vehicle Routing Problem

DP
Dynamic Programming

DVRP
Distance-Constrained Vehicle Routing Problem

LP
Linear Program

PCAP
Prize-Collecting Arborescence Problem

TSP
Travelling Salesman Problem

VRP
Vehicle Routing Problem

VRPMT
Vehicle Routing Problem with Multiple Trips

VRPP
Vehicle Routing Problem with Profits

VRPPD
Vehicle Routing Problem with Pickup and Delivery

VRPTW
Vehicle Routing Problem with Time Windows

Chapter 1

Introduction

Vehicle routing problems (VRPs) are a broad class of combinatorial optimization problems with a wide range of applications. The general objective in this class of problems is to find the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of clients under several constraints. There are several variants of vehicle routing problems, each addressing different measurements of a feasible vehicle routing solution, including:

- CVRP (Capacitated Vehicle Routing Problem)
- VRPP (Vehicle Routing Problem with Profits)
- VRPPD (Vehicle Routing Problem with Pickup and Delivery)
- VRPTW (Vehicle Routing Problem with Time Windows)
- VRPMT (Vehicle Routing Problem with Multiple Trips)

The problem we will be trying to tackle in this thesis is the less-studied distance-constrained vehicle routing problem (DVRP). As an example of DVRP, suppose we are to deliver packages from a depot to all clients today, but each client must receive the package by 5pm. In this model, we are looking to minimize the number of trucks we deploy to serve the clients.

1.1 Preliminaries and Notations

Definition 1 (Linear Program (LP))**.** A linear program $(\boldsymbol{A}_{m\times n}, \boldsymbol{b}_{m\times 1}, \boldsymbol{c}_{n\times 1})$ is an optimization problem with a linear objective function and linear constraints.

Equation (1.1) shows the *canonical form* of an LP.

$$\begin{aligned} \max \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned} \tag{1.1}$$

Equation (1.2) shows the same LP in scalar format. It is important to note that the columns and rows of $\boldsymbol{A}$ correspond to the variables and constraints of the LP, respectively.

$$\begin{aligned} \max \sum_j c_j x_j & \\ \text{s.t.} \sum_j a_{ij} x_j \leq b_i & \qquad \forall 1 \leq i \leq m \\ x_j \geq 0 & \qquad \forall 1 \leq j \leq n \end{aligned} \tag{1.2}$$

The simplex algorithm works with equality constraints which are obtained by introducing *slack variables.*

$$s_i := b_i - \sum_{j=1}^{n} a_{ij} x_j \quad \forall 1 \leq i \leq m \tag{1.3}$$

(1.4) is the *slack form* of the same LP.

$$\begin{aligned} \max \sum_{j=1}^{n} c_j x_j & \\ \text{s.t.} \quad \sum_{j=1}^{n} a_{ij} x_j + s_i = b_i & \qquad \forall 1 \leq i \leq m \\ x_j \geq 0 & \qquad \forall 1 \leq j \leq n \\ s_i \geq 0 & \qquad \forall 1 \leq i \leq m \end{aligned} \tag{1.4}$$

In order to rewrite (1.4) using vectors and matrices, we need to augment $\boldsymbol{A}$, $\boldsymbol{c}$, and $\boldsymbol{x}$ as follows:

$$\begin{aligned} \max \quad & \left(\boldsymbol{c}^T | \boldsymbol{0}\right) \hat{\boldsymbol{x}} \\ \text{s.t.} \quad & \left(\boldsymbol{A} | \boldsymbol{I}_m\right) \hat{\boldsymbol{x}} = \boldsymbol{b} \\ & \hat{\boldsymbol{x}} \geq \boldsymbol{0} \end{aligned} \tag{1.5}$$

where $\hat{\boldsymbol{x}}$ denotes the column vector of slack variables appended to x_i variables. We will briefly discuss the simplex algorithm in chapter 2.

Definition 2 (Covering LP)**.** A minimization LP($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c}$) is called a *covering LP* if and only if all elements of $\boldsymbol{A}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are non-negative.[1]

Definition 3 (Metric)**.** A distance function $d : S \times S \to [0, +\infty)$ on set S is called metric if and only if for all $x, y, z \in S$ the following three conditions hold.

Identity of indiscernibles

$$d(x, y) = 0 \iff x = y$$

Symmetry

$$d(x, y) = d(y, x)$$

Triangle inequality

$$d(x, y) \leq d(x, z) + d(z, y)$$

Definition 4 (α-approximation)**.** Let P be an optimization problem with optimal value OPT. An algorithm is an α-approximation for P if for every instance of P, it provides value V such that:

$$1 \leq \max\left\{\frac{V}{OPT}, \frac{OPT}{V}\right\} \leq \alpha \tag{1.6}$$

where $\alpha \geq 1$ is called an approximation factor. For a minimization problem, (1.6) translates to

$$\frac{V}{\alpha} \leq OPT \leq V$$

and for a maximization problem, it translates to

$$V \leq OPT \leq \alpha \cdot V$$

Definition 5 (Arborescence)**.** An arborescence rooted at node r is a directed graph in which there is exactly one walk from r to any other node. Equivalently, if the edge directions are ignored, then an arborescence becomes a tree. Figure 1.1 demonstrates an example of an arborescence.

[1]Examples include minimum set cover, minimum vertex cover, and minimum edge cover

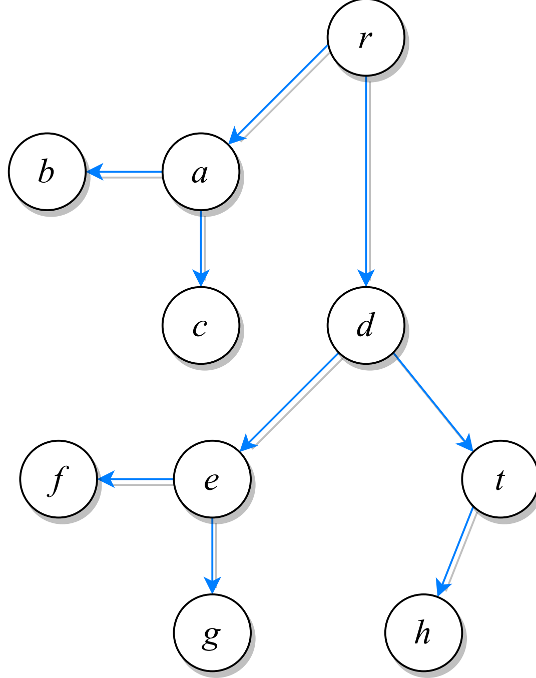


Figure 1.1: An example of an arborescence

The following conditions hold for an arborescence rooted at r:

$$\begin{cases} \delta_{in}(r) = 0 \\ \delta_{in}(v) = 1 \quad v \in V - \{r\} \end{cases}$$

where δ_{in} denotes the number of incoming edges or indegree of a node.

Definition 6 (Distance Cost)**.** For graph $G(V, E)$, we define the *distance cost* of G as the sum of distances of its edges, *i.e.*

$$d(G) := \sum_{e \in E} d_e$$

Definition 7 (Reward)**.** For graph $G(V, E)$, we define the reward of G as the sum of the rewards of its vertices[2], *i.e.*

$$\pi(G) := \sum_{v \in V} \pi_v$$

Also, we define the reward of vertex set S as the sum of the rewards of its elements, *i.e.*

$$\pi(S) := \sum_{v \in S} \pi_v$$

[2]This notation is mostly used for paths and arborescences in this thesis

Definition 8 (Prize-Collecting Cost Function)**.** We define the prize-collecting cost or simply cost function of S, a subgraph of $G(V, E)$, for a positive λ as

$$c(S, \lambda) \coloneqq \lambda \cdot \pi(V - V(S)) + d(S)$$

where $V(S)$ denotes the vertex set of S. In other words, the prize-collecting cost of S is the distance cost of all edges of S plus the rewards of vertices that are not included in S.

Definition 9 (Rooted Hamiltonian path problem)**.** Given a metric graph G with distance matrix $\left[d_{uv}\right]$, a root node r, and a distance bound D, determine whether there exists a path starting at r and with distance cost of at most D, which visits every node of G exactly once. The rooted Hamiltonian path problem is **NP**-hard (Garey & Johnson, 1990).

1.2 Motivation and Literature Review

Definition 10 (Distance-Constrained Vehicle Routing Problem (DVRP))**.** Given a graph $G(V \cup \{r\}, E)$ and its associated distance matrix $\left[d_{uv}\right]$ with rational distances, a root node (r), and a distance limit $D \geq 0$, the objective is to cover all the other nodes using the least number of paths starting at r whose length is at most D. The distance-constrained vehicle routing problem can be mathematically formulated by the integer linear program demonstrated in (1.7).

$$\begin{aligned} \min \quad & \sum_{P \in \mathcal{P}_D^r} x_P \\ \text{s.t.} \quad & \sum_{P : v \in P} x_P \geq 1 \qquad \forall v \in V \\ & x_P \in \{0, 1\} \qquad \forall P \in \mathcal{P}_D^r \end{aligned} \tag{1.7}$$

where $\mathcal{P}_D^r$ is the set of all paths that start at the root node (r) and whose distance cost is less than or equal to D, and x_P is a decision variable indicating whether path P is selected or not.

A common approach when dealing with integer programs is relaxing the constraints[3] to obtain a linear program (LP). In case of (1.7), this can be done

[3]Known as *LP relaxation*

by assuming that x_P variables can take any non-negative real value, which gives us (1.8).

$$\begin{aligned} \min \quad & \sum_{P \in \mathcal{P}^r_D} x_P \\ \text{s.t.} \quad & \sum_{P : v \in P} x_P \geq 1 \qquad \forall v \in V \\ & x_P \geq 0 \qquad\qquad \forall P \in \mathcal{P}^r_D \end{aligned} \tag{1.8}$$

Several approximation algorithms have been suggested to solve VRPs (Arkin et al., 2006; Bansal et al., 2004; Toth & Vigo, 2001) and more specifically DVRP (Nagarajan & Ravi, 2012). Nagarajan and Ravi (2012) have proposed a 2-approximation algorithm if the distance metric is the shortest path in a tree. They also proposed an $O(\log D)$-approximation algorithm for DVRP with general metrics and integer distances. Their algorithm can also provide a $(\log \frac{1}{\epsilon})$-approximation if the distance bound is allowed to be violated by a factor of $(1+\epsilon)$. (Nagarajan & Ravi, 2012)

Friggstad and Swamy (2014) were able to achieve an $O(\frac{\log D}{\log \log D})$-approximation for DVRP with integer distances. Their algorithm uses the ellipsoid method (Grötschel et al., 2012) to provide a constant factor approximation to (1.8), which makes their algorithm impractical.

A plausible conjecture is that DVRP itself admits a constant-factor approximation. If so, it seems possible that such an approximation could be obtained by somehow rounding a near-optimal solution to (1.8). Also in practice, an LP solution could be rounded to an integer solution using integer programming methods (Barnhart et al., 1998).

The challenge of solving (1.8) with simplex is that the number of variables (paths) grows exponentially with the number of nodes as $|\mathcal{P}^r_D| = \Omega(2^{|V|})$. To overcome this challenge, we use an approach called *column generation*[4]. Column generation has been used in the literature to solve problems such as maximum multi-commodity flow problem (Ford & Fulkerson, 2004), the cutting stock problem (Desrosiers & Lübbecke, 2005; Gilmore & Gomory, 1961, 1963), and even VRPs (Desrosiers et al., 1984).

[4]Also known as *delayed column generation*

When using column generation, the problem is split into a main problem and a subproblem. The main problem is the original one with only a small subset of variables. The subproblem is then used to identify a new variable at each iteration.

As we will explain in more detail in chapter 2, in order to generate a column that can potentially improve the objective function of (1.8), we have to find a path which violates the constraints of its dual LP described by (1.9), *i.e.* a path of bounded length where sum of the dual variables (π_v) along that path is more than 1.

$$\begin{aligned} \max \quad & \sum_{v \in V} \pi_v \\ \text{s.t.} \quad & \sum_{v \in P} \pi_v \leq 1 \quad \forall P \in \mathcal{P}_D^r \\ & \pi_v \geq 0 \qquad\quad \forall v \in V \end{aligned} \tag{1.9}$$

To that end, we try to find a bounded path starting at the root node (r) with maximum sum of the dual variables. This subproblem is known as the *rooted orienteering problem* (Blum et al., 2003; Friggstad & Swamy, 2017; Nagarajan & Ravi, 2011). We then check to see if the sum is more than 1 or not.

Definition 11 (Rooted Orienteering Problem)**.** Given a complete bidirected metric graph $G(V, E)$, integer distance matrix $\left[d_{uv}\right]$, a root node r, an integer distance bound ($D \geq 0$), and non-negative node rewards ($\pi_v \geq 0$), the objective is to find a rooted path, *i.e.* a path starting at node r, with distance cost of at most D that maximizes the reward along that path.

Blum et al. (2003) and Bansal et al. (2004) have proposed 4- and 3-approximation algorithms for solving orienteering, respectively. Their approximations, however, do not seem practical due to the fact that their dynamic programming algorithm has multiple parameters, which results in a high-degree polynomial time complexity.

A $(2 + \epsilon)$-approximation for orienteering in undirected graphs has been proposed by Chekuri et al. (2012). Their algorithm has a time complexity of $O(|V|^{\frac{c}{\epsilon}})$ for some constant c, which means that their algorithm runs in

polynomial time for a constant $\epsilon > 0$. However, this algorithm is impractical for small enough ϵ, as well.

Friggstad and Swamy (2017) introduced the first LP-based 3-approximation algorithm for rooted orienteering, which was based on solving an LP relaxation and then rounding its solution to an integral one.

Post and Swamy (2017) used insights from the aforementioned paper (Friggstad & Swamy, 2017) and designed a combinatorial $(3+\epsilon)$-approximation algorithm for orienteering, which is the first approximation that stands a chance of running fast enough in practice. We will elaborate on their algorithm in chapter 3.

Our main contributions in this work include:

- Designing and implementing a more practical algorithm for solving (1.8)[5].
- Calculating the approximation factor and lower bound for every individual instance

The rest of this work is organized as follows:

In chapter 2, we will briefly go over the simplex algorithm and also explain column generation and why it works. Furthermore, we solve an instance of the maximum multi-commodity problem as an example in section 2.1 to explain the process of generating columns. We will also discuss how using approximation algorithms for solving the column generation subproblem can lead to an approximation for the main problem in section 2.2.

In chapters 3 and 4, we explain the orienteering algorithm proposed by Post and Swamy (2017) and combine it with column generation to come up with our own practical $(3+\epsilon)$-approximation algorithm for (1.8). We also analyze the asymptotic time complexity of the algorithm.

In chapter 5, we desrcibe the datasets we used in our experiments and present the results. We will show that the approximation factor is far better than $3+\epsilon$ in practice, at least in our experiments. Finally, we conclude by suggesting possible optimizations to the algorithm.

[5]Solving the integer program (1.7) was not one of the main goals of this work. However, we used one of the integer program solvers commonly used in the industry, to solve the integer program. You can see our results in appendix A.

Chapter 2

Column Generation for Linear Programs

We begin by briefly summarizing the basic workings of the simplex algorithm, which are essential to understanding the column generation approach, without reproducing all the details of the algorithm and then go on to describe column generation itself. Next, in section 2.1, we use column generation to solve an instance of the maximum multi-commodity flow problem as an example. Finally in section 2.2, we explain how approximate column generation yields an approximate solution to the LP.

Definition 12. Consider the LP in (2.1).

$$\begin{aligned} \max \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned} \tag{2.1}$$

A basis (B) is a subset of column indices where $\boldsymbol{A}_B$ is a square and invertible matrix. $\boldsymbol{A}_B$ denotes the matrix formed by columns in B. Also, let N be the set of column indices that are not in B. $\overline{\boldsymbol{x}}$ is a *basic solution* if and only if:

$$\begin{cases} \boldsymbol{A}_B \overline{\boldsymbol{x}}_B = \boldsymbol{b} \implies \overline{\boldsymbol{x}}_B = \boldsymbol{A}_B^{-1} \boldsymbol{b} \\ \overline{\boldsymbol{x}}_N = \boldsymbol{0} \end{cases} \tag{2.2}$$

if $\overline{\boldsymbol{x}} \geq \boldsymbol{0}$, then $\overline{\boldsymbol{x}}$ is a basic feasible solution.

From a geometric point of view, the space of feasible solutions forms a convex polyhedron, and the basic feasible solutions are its vertices (corners). See example 2.1.

Example 2.1. Consider the LP in (2.3).

$$\begin{aligned} \max \quad & c_1x_1 + c_2x_2 & & \\ x_1 + x_2 &\leq 4 & \quad (1) & \\ x_1 &\leq 3 & \quad (2) & \\ x_2 &\leq 3 & \quad (3) & \\ x_1, x_2 &\geq 0 & & \end{aligned} \tag{2.3}$$

We add slack variables s_1, s_2, and s_3 to convert inequalities (1),(2), and (3) to equalities as demonstrated in (2.4).

$$\begin{aligned} \max \quad & c_1x_1 + c_2x_2 & & \\ x_1 + x_2 + s_1 &= 4 & \quad (1) & \\ x_1 + s_2 &= 3 & \quad (2) & \\ x_2 + s_3 &= 3 & \quad (3) & \\ x_1, x_2, s_1, s_2, s_3 &\geq 0 & & \end{aligned} \tag{2.4}$$

We can rewrite (2.4) using vectors and matrices as demonstrated in (2.5).

$$\begin{aligned} \max \; & \begin{bmatrix} c_1 \\ c_2 \\ 0 \\ 0 \\ 0 \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \\ \text{s.t.} \; & \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 3 \end{bmatrix} \end{aligned} \tag{2.5}$$

$B = \{1, 2, 4\}$ is a possible basis for (2.4) as $\boldsymbol{A}_B = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ is an invertible matrix. If we set the non-basic variables, s_1 and s_3, to 0, then the basic

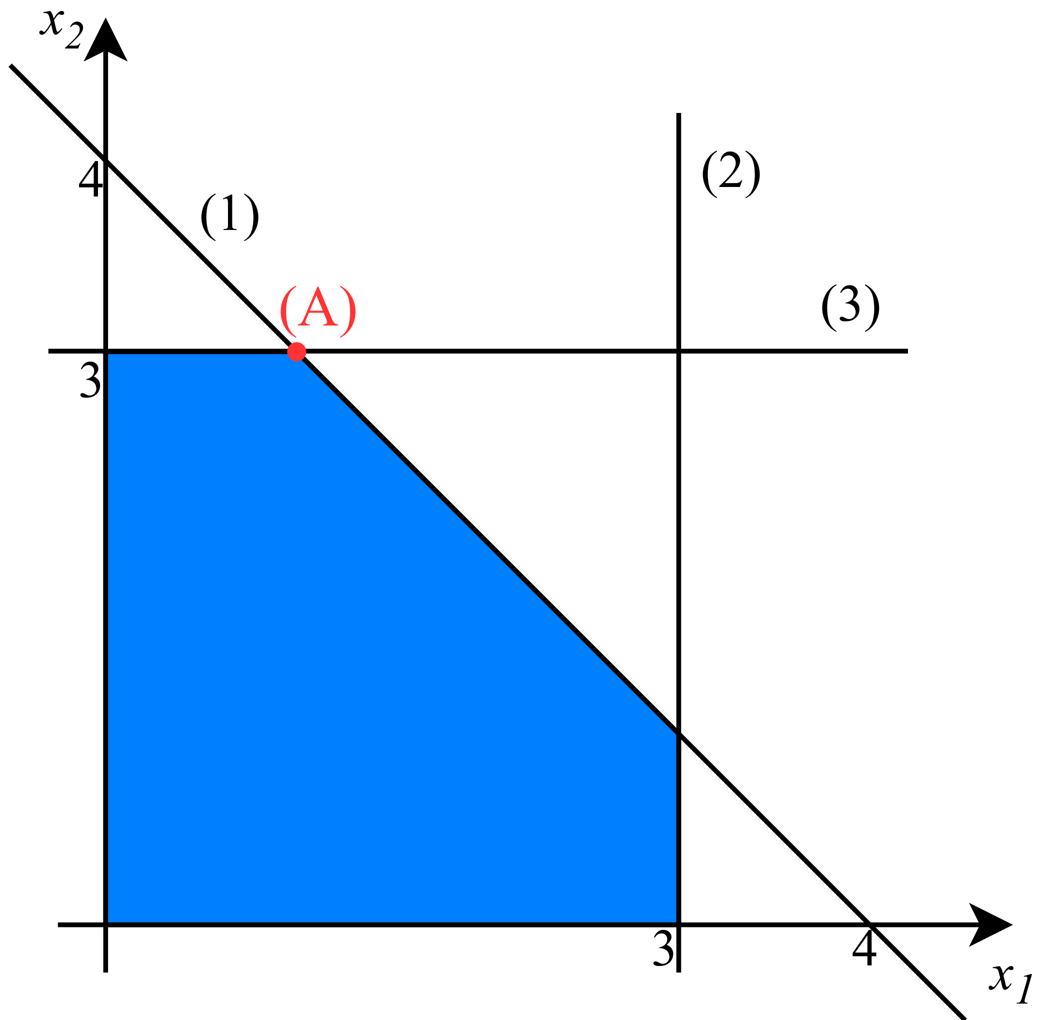


Figure 2.1: Feasible region of (2.3)

variables are determined by the system of equations in (2.2).

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ s_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 3 \end{bmatrix}$$
$$\implies \overline{\boldsymbol{x}}_B = \begin{bmatrix} x_1 \\ x_2 \\ s_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}$$
$$\implies \overline{\boldsymbol{x}} = \begin{bmatrix} x_1 \\ x_2 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 0 \\ 2 \\ 0 \end{bmatrix} \tag{2.6}$$

$\overline{\boldsymbol{x}}$ is a basic solution to (2.4), and since all the elements in $\overline{\boldsymbol{x}}$ are non-negative, it is also a basic feasible solution corresponding to the corner (A) in fig. 2.1.

In addition to including slack variables, we also add a constant term $\bar{z}$ to the objective function as part of the simplex algorithm. (2.7) is the final representation of the LP with which simplex works.

$$\begin{aligned} \max \quad & \boldsymbol{c}^T\boldsymbol{x} + \bar{z} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned} \tag{2.7}$$

From a high level point of view, the simplex algorithm searches for an optimal basis by performing a *pivot operation* at each iteration. During each pivot operation, a new column enters, and an old column exits the basis. The new column (k) should be carefully chosen, *e.g.* by using Bland's rule(Guenin et al., 2014), to prevent infinite loops.

Algorithm 2.1 Simplex Algorithm

Input: Linear Program $P(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ and feasible basis B
Output: An optimal solution of P or a certificate of unboundedness

1: $\bar{z} \leftarrow \boldsymbol{c}_B^T \boldsymbol{A}_B^{-1} \boldsymbol{b}$
2: Rewrite($\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$, B)
3: **while** $\exists k \in N : c_k > 0$ **do** ▷ While there exists a column that can improve the objective function.
4: **if** $\boldsymbol{A}_k \leq \boldsymbol{0}$ **then** ▷ P is unbounded
5: **return** Certificate of unboundedness
6: **end if**
7: $r \leftarrow \arg\min_i \{\frac{b_i}{a_{ik}} : a_{ik} > 0\}$
8: $\ell \leftarrow$ the rth basis element
9: $B \leftarrow (B \cup \{k\}) - \{\ell\}$ ▷ Add new column and remove old column from the basis
10: $t \leftarrow \frac{b_r}{a_{rk}}$
11: $\bar{z} \leftarrow \bar{z} + c_k t$
12: Rewrite($\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$, B)
13: **end while**
14: **return** $\boldsymbol{x}, \bar{z}$

1: **procedure** Rewrite($\boldsymbol{A}$, $\boldsymbol{b}$, $\boldsymbol{c}$, B)
2: $\boldsymbol{x}_B \leftarrow \boldsymbol{A}_B^{-1} \boldsymbol{b}$
3: $\boldsymbol{x}_N \leftarrow \boldsymbol{0}$
4: $\boldsymbol{y} \leftarrow \boldsymbol{A}_B^{-T} \boldsymbol{c}_B$ ▷ The dual variables
5: $\boldsymbol{c} \leftarrow \boldsymbol{c} - \boldsymbol{A}^T \boldsymbol{y}$ ▷ Reduced cost
6: $\boldsymbol{b} \leftarrow \boldsymbol{A}_B^{-1} \boldsymbol{b}$
7: $\boldsymbol{A} \leftarrow \boldsymbol{A}_B^{-1} \boldsymbol{A}$
8: **end procedure**

After each iteration of the simplex algorithm, the new objective function value would increase by $c_k t$. Since c_k is positive and t is non-negative, the new value be would be no smaller than the old one. If $c_k t$ is 0, then the pivot is a *degenerate pivot* in which only the basis changes but not the objective function value.

From a geometric perspective, the simplex algorithm jumps from one vertex to an adjacent one at each iteration until it finds an optimal vertex.

Note: The $\boldsymbol{y}$ variables on line 4 of the REWRITE procedure are called dual variables as they correspond to the variables in the dual LP. The dual of (1.2) is defined as:

$$\begin{aligned} \min & \sum_{i=1}^{m} b_i y_i \\ \text{s.t.} & \sum_{i=1}^{m} a_{ij} y_i \geq c_j \quad \forall 1 \leq j \leq n \\ & y_i \geq 0 \qquad\qquad\quad \forall 1 \leq i \leq m \end{aligned} \tag{2.8}$$

If the number of constraints is far less than the number of variables, *e.g.* when $n = \Omega(2^m)$, the solution would be very sparse. Moreover, if we examine the simplex algorithm and the REWRITE procedure more closely, we can see that, for most of the computations in each iteration, we just need to know a basis (B), its corresponding columns of $\boldsymbol{A}$ ($\boldsymbol{A}_B$), and its corresponding objective function coefficients ($\boldsymbol{c}_B$). It is, therefore, possible to start with a few columns and generate new ones as needed.

The process of column generation consists of the following steps:

1. Start with m columns.

2. Solve the LP and calculate the dual variables.

3. Solve the subproblem to find a column that violates the dual constraints.

4. If a column is found, add it to the LP and go to Step 2.

When no new column is found, then, according to theorem 2.1, the solution is optimal.

Theorem 2.1. *Let $\overline{\boldsymbol{x}}$ be a basic feasible solution corresponding to basis B and $\boldsymbol{y} = \boldsymbol{A}_B^{-T}\boldsymbol{c}_B$. If $\boldsymbol{y}$ is a feasible dual solution, then $\overline{\boldsymbol{x}}$ is an optimal solution.*

Proof by contradiction. Let $\boldsymbol{x'}$ be a feasible solution whose value is larger than $\overline{\boldsymbol{x}}$, *i.e.*

$$\boldsymbol{c}^T\boldsymbol{x'} > \boldsymbol{c}^T\overline{\boldsymbol{x}} \tag{2.9}$$

Since $\boldsymbol{y}$ is feasible, $\boldsymbol{A}^T\boldsymbol{y} \geq \boldsymbol{c}$, therefore,

$$\begin{aligned} \forall j, (\boldsymbol{y}^T\boldsymbol{A})_j \geq c_j \xRightarrow{\boldsymbol{x'}\geq\boldsymbol{0}} \forall j, (\boldsymbol{y}^T\boldsymbol{A})_j x'_j \geq c_j x'_j \implies \sum_j (\boldsymbol{y}^T\boldsymbol{A})_j x'_j \geq \sum_j c_j x'_j \\ \implies \boldsymbol{c}^T\boldsymbol{x'} \leq \boldsymbol{y}^T\boldsymbol{A}\boldsymbol{x'} \end{aligned} \tag{2.10}$$

$$\begin{aligned} \boldsymbol{y}^T(\boldsymbol{A}\boldsymbol{x'}) &= \boldsymbol{y}^T\boldsymbol{b} && \text{by feasibility of } \boldsymbol{x'} \\ &= \boldsymbol{c}_B^T\boldsymbol{A}_B^{-1}\boldsymbol{b} \\ &= \boldsymbol{c}_B^T\overline{\boldsymbol{x}}_B && \text{by (2.2)} \\ &= \boldsymbol{c}^T\overline{\boldsymbol{x}} \\ \implies \boldsymbol{y}^T(\boldsymbol{A}\boldsymbol{x'}) &= \boldsymbol{c}^T\overline{\boldsymbol{x}} \end{aligned} \tag{2.11}$$

$$(2.10), (2.11) \implies \boldsymbol{c}^T\boldsymbol{x'} \leq \boldsymbol{c}^T\overline{\boldsymbol{x}} \tag{2.12}$$

which contradicts (2.9). □

In section 2.1, we provide an example for column generation by using it to solve the maximum multi-commodity flow problem similar to (Ford & Fulkerson, 2004).

2.1 An Example of Column Generation: Solving the Maximum Multi-Commodity Flow Problem

Definition 13 (Maximum Multi-Commodity Flow Problem)**.** Let $G(V, E)$ be a directed graph where every edge $e \in E$ has an associated capacity c_e. Also, let $K_1, K_2, \ldots, K_k$ be k commodities such that each commodity K_i is defined as the ordered pair (s_i, t_i) where s_i and t_i are the source and sink nodes of

the commodity K_i, respectively. The objective of the problem is to maximize the total flow going from the sources to the sinks provided that for every commodity K_i, the amount of flow going out of s_i has to be equal to the amount of flow going into t_i .

One way to formulate this problem is to imagine an amount of flow f_p for each path p from s_i to t_i. The LP (2.13) represents this formulation mathematically:

$$
\begin{aligned}
\max \quad & \sum_{i=1}^{k} \sum_{p \in \mathcal{P}_i} f_p & \forall s_i - t_i \text{ pair} \\
\text{s.t.} \quad & \sum_{i=1}^{k} \sum_{p \in \mathcal{P}_i : e \in p} f_p \le c_e & \forall e \in E \\
& f_p \ge 0 & \forall 1 \le i \le k \forall p \in \mathcal{P}_i
\end{aligned}
\tag{2.13}
$$

where $\mathcal{P}_i$ is the set of all paths from s_i to t_i.

The constraint indicates that for every edge (e), the total amount of flow going through that edge has to be less than or equal to its capacity (c_e). The dual of (2.13) is as follows:

$$
\begin{aligned}
\min \quad & \sum_{e \in E} c_e y_e \\
\text{s.t.} \quad & \sum_{e \in p} y_e \ge 1 \quad \forall 1 \le i \le k \forall p \in \mathcal{P}_i \\
& y_e \ge 0 \qquad \forall e \in E
\end{aligned}
$$

If there exists a path which violates the constraints of the dual LP, that is, the sum of dual variables along that path is less than 1, then that path has the potential to increase the objective function value in the primal LP.

In order to find such a path, we construct a new graph G' with the same nodes and edges and set the weight of every edge (e) equal to its corresponding dual variable (y_e). Since all the weights are non-negative, we can run Dijkstra's algorithm for each of the $s_i - t_i$ pairs to find a path of length less than 1 and add it to the LP as our new column. When no such path is found, the algorithm terminates.

Example 2.2. Consider the graph in fig. 2.2.

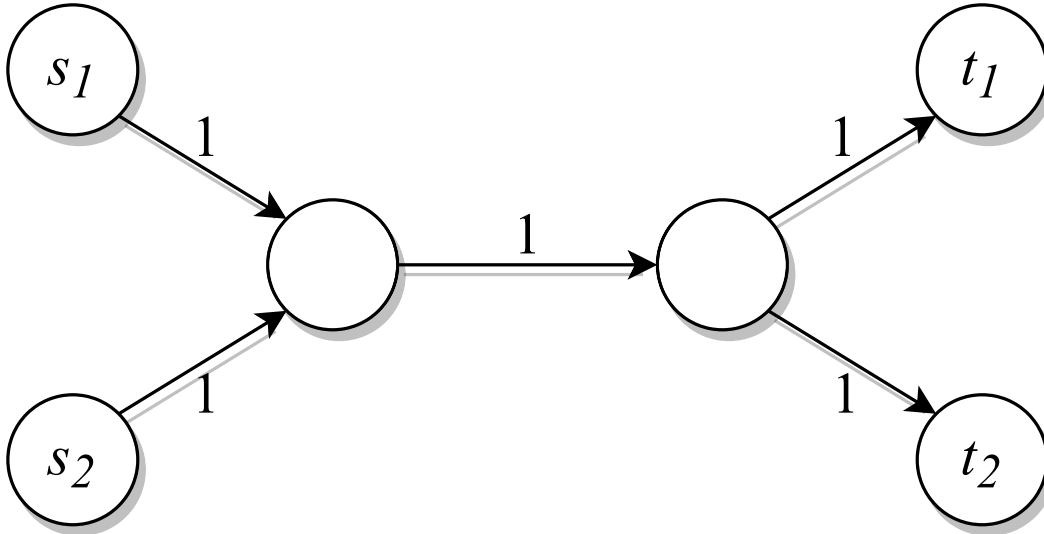


Figure 2.2: The edge weights denote capacities

In order to solve this problem with column generation, we number all the edges according to fig. 2.3 in order to map the rows of $\boldsymbol{A}$ and $\boldsymbol{b}$ to the edges.

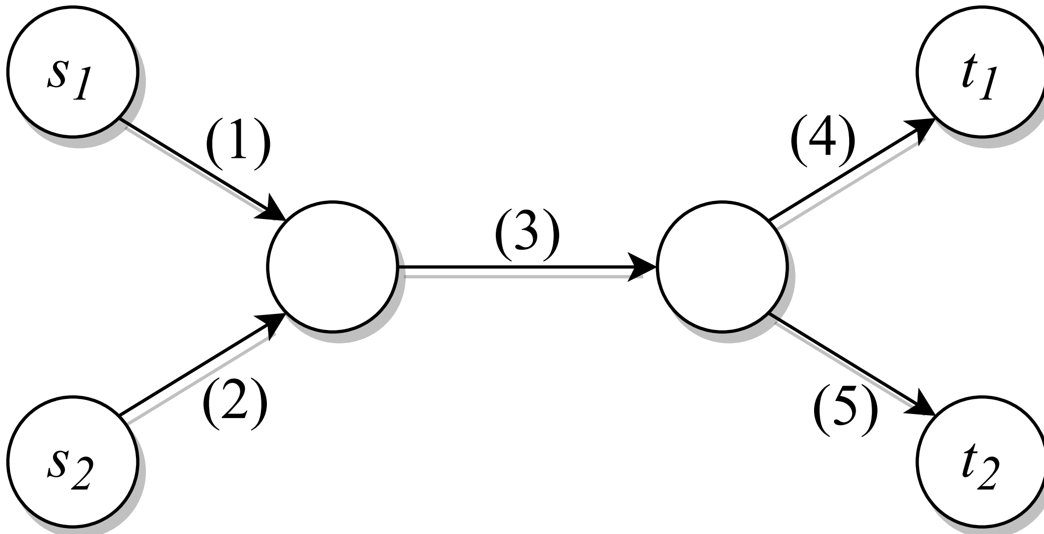


Figure 2.3: The edges numbered

Since there are no paths initially, we only have the slack variables. (2.14)

shows the initial LP for this example.

$$\begin{aligned} \max & \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}^T \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix} \\ \text{s.t.} & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{aligned} \tag{2.14}$$

We start with $B = \{1, 2, 3, 4, 5\}$ as the basis and calculate the dual variables and map them to the edges as demonstrated in fig. 2.4.

$$\boldsymbol{y} = \boldsymbol{A}_B^{-T} \boldsymbol{c}_B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

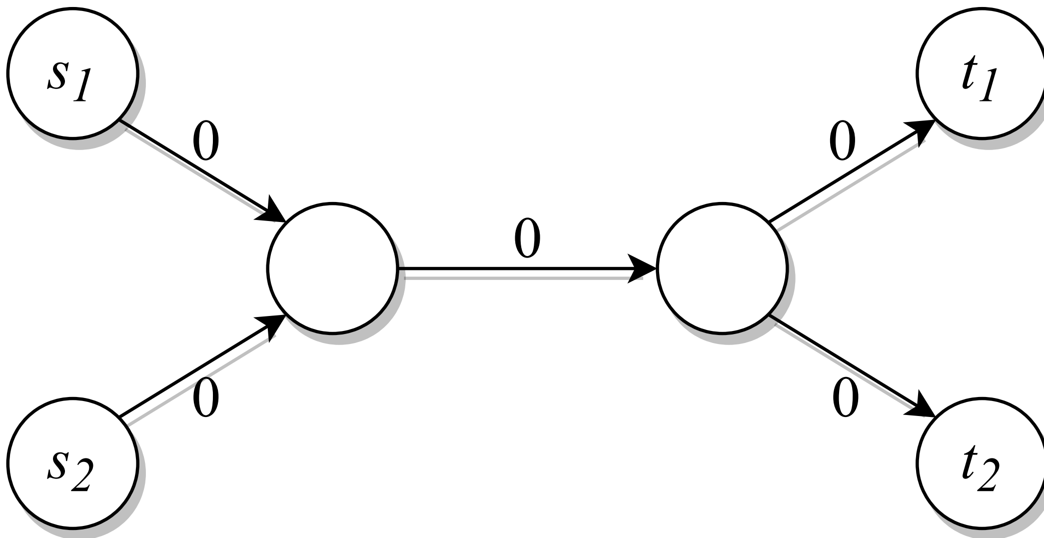


Figure 2.4: The initial dual variables

Using Dijkstra's algorithm, we find the shortest path between s_1 and t_1 shown in fig. 2.5.

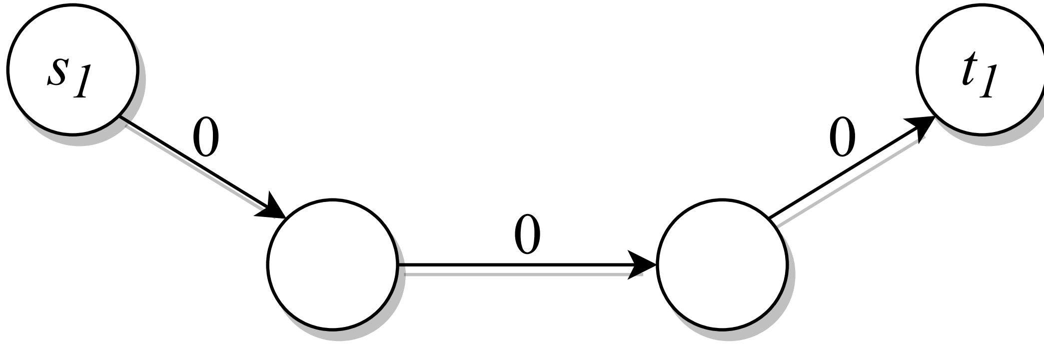


Figure 2.5: The shortest path between s_1 and t_1

As the length of the path in fig. 2.5 is less than 1, we can add this path as a column to the LP with the corresponding variable f_1.

$$\max \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}^T \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ f_1 \end{bmatrix}$$

$$\text{s.t.} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ f_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Using the simplex algorithm, we choose to evict the first column and add the new column. The new basis would then be $B = \{2, 3, 4, 5, 6\}$, and the new basic variables $s_2 = 1$, $s_3 = 0$, $s_4 = 0$, $s_5 = 1$, and $f_1 = 1$.

We recalculate the dual variables and map them to edges.

$$\boldsymbol{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

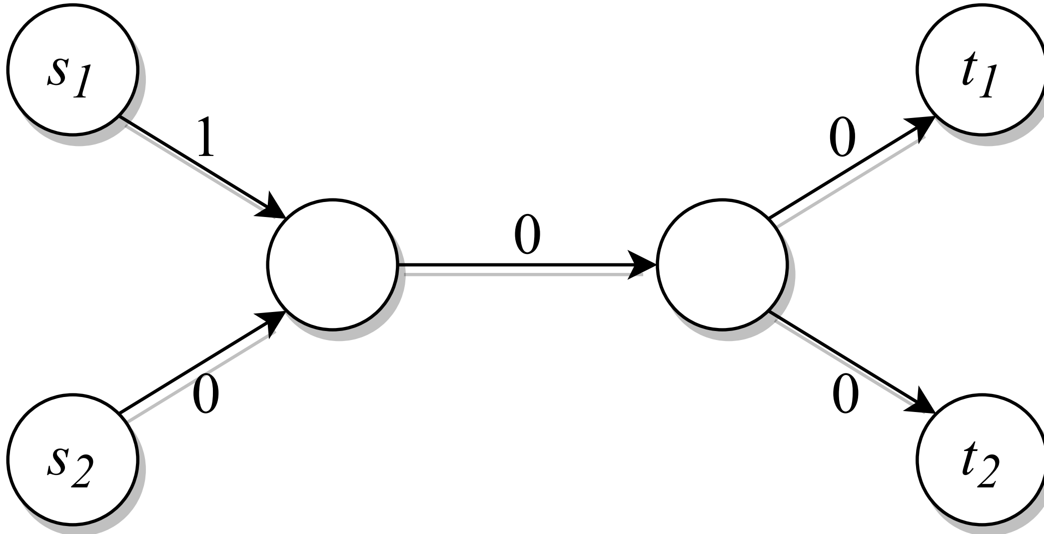


Figure 2.6: The dual variables after generating the sixth column

The shortest path between s_1 and t_1 now has a length of 1 so it satisfies the dual constraint. Using Dijkstra's algorithm again, we find the path in fig. 2.7 and add it to the LP with the corresponding variable f_2.

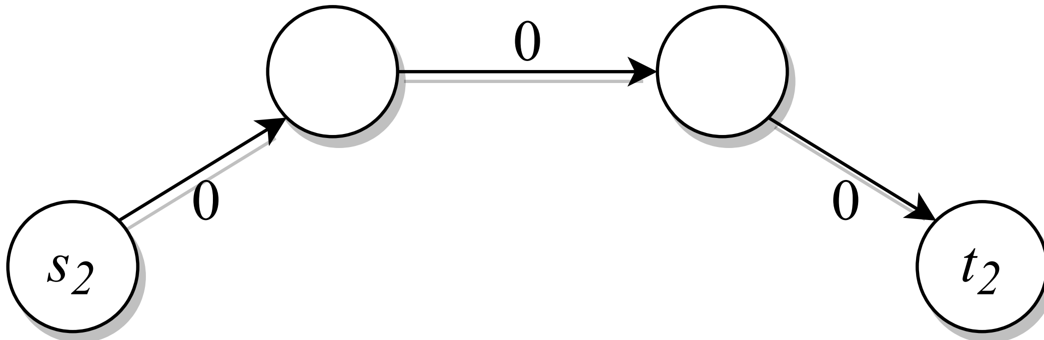


Figure 2.7: The shortest path between s_2 and t_2

$$\max \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}^T \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ f_1 \\ f_2 \end{bmatrix}$$

$$\text{s.t.} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Using the simplex algorithm, we choose to evict the third column and add the new column. As a result, the new basis becomes $B = \{2, 4, 5, 6, 7\}$, and the new basic variables will be $s_2 = 1$, $s_4 = 0$, $s_5 = 1$, $f_1 = 1$, and $f_2 = 0$.

Once again, we recalculate the dual variables and map them to edges.

$$\boldsymbol{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

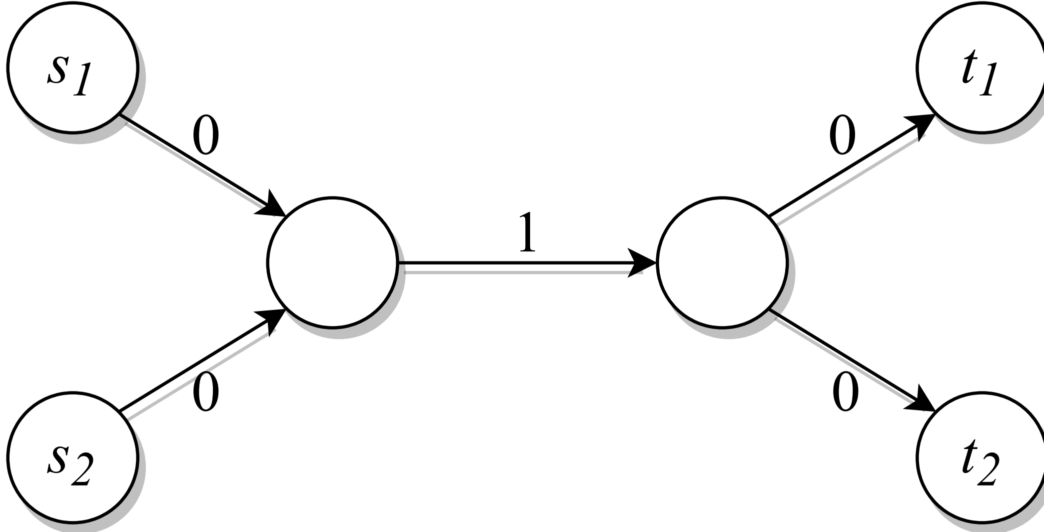


Figure 2.8: The final dual variables after generating the seventh column

Now that all the paths between $s - t$ pairs have a length of at least 1, in accordance with theorem 2.1, we have reached the optimal solution.

From this point forward, we only focus on *minimization LPs* and more specifically *covering LPs*.

2.2 Column Generation Using Approximation Algorithms

In some problems, *e.g.* set partitioning, cutting stock, bin packing, vehicle routing[1], and edge partitioning, even the column generation subproblem itself can be **NP**-hard. Therefore, we try to solve the subproblem using approximation algorithms. When our algorithms fail to generate a column, then according to theorem 2.2, guarantees that the solution is near-optimal for the primal LP. We provide a simple proof for theorem 2.2 as we are not aware of it appearing in the literature.

Theorem 2.2. *Let $\overline{\boldsymbol{x}}$ be a vertex corresponding to basis B for a covering linear program $(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ with optimum solution value OPT, and let $\boldsymbol{y} = \boldsymbol{A}_B^{-T} \boldsymbol{c}_B$. If $\boldsymbol{y}/\alpha$ is a feasible dual solution for some $\alpha \geq 1$, then $\frac{\boldsymbol{c}^T \overline{\boldsymbol{x}}}{\alpha}$ is a lower bound for*

[1]Discussed in this thesis

the optimal value OPT, i.e.

$$\frac{\boldsymbol{c}^T\overline{\boldsymbol{x}}}{\alpha} \leq OPT \tag{2.15}$$

Proof.

$$\boldsymbol{c}^T\overline{\boldsymbol{x}} = \overline{\boldsymbol{x}}^T\boldsymbol{c} = \overline{\boldsymbol{x}}_B^T\boldsymbol{c}_B + \overline{\boldsymbol{x}}_N^T\boldsymbol{c}_N = \overline{\boldsymbol{x}}_B^T\boldsymbol{c}_B = \boldsymbol{b}^T\boldsymbol{A}_B^{-T}\boldsymbol{c}_B = \boldsymbol{b}^T\boldsymbol{y} \tag{2.16}$$

Since $\boldsymbol{y}/\alpha$ is a feasible dual solution,

$$\boldsymbol{b}^T(\frac{\boldsymbol{y}}{\alpha}) \leq OPT \implies \frac{\boldsymbol{b}^T\boldsymbol{y}}{\alpha} \leq OPT \xRightarrow{(2.16)} \frac{\boldsymbol{c}^T\overline{\boldsymbol{x}}}{\alpha} \leq OPT$$

□

Chapter 3

Approximately Solving The Orienteering Subproblem

The rooted orienteering problem is an **NP**-hard problem as every instance of rooted Hamiltonian path problem, defined in definition 9, can be reduced to an instance of the orienteering problem by saying that a Hamiltonian path exists if and only if the optimum orienteering solution spans all nodes. As a result, we will need to use heuristic or approximation algorithms to solve the orienteering subproblem.

We use a $(3+\epsilon)$-approximation algorithm suggested by Post and Swamy (2017) to solve the rooted orienteering problem. Their approximation uses an algorithm called IterPCA (Post & Swamy, 2017) internally combined with bisection method to find two arborescences from which a path can be extracted.

In order to understand the approximation algorithm for orienteering, we first need to discuss PCAP and IterPCA.

Definition 14 (Prize-Collecting Arborescence Problem (PCAP))**.** Given a graph $G(V, E)$, edge weights d_e, and node rewards π_v, the objective is to find an arborescence (T) with the minimum prize-collecting cost, *i.e.*

$$\arg\min_{T} c(T, 1) = \arg\min_{T} \left[\pi(V - V(T)) + d(T)\right] \tag{3.1}$$

PCAP can be regarded as a generalization of the Steiner tree problem (Charikar et al., 1999). PCAP is also **NP**-hard and difficult to approximate (Halperin & Krauthgamer, 2003).

The IterPCA algorithm (Post & Swamy, 2017) does not address PCAP directly. Instead, it finds an arborescence T_λ whose cost is at most the prize-collecting cost of any rooted walk in the graph. It also finds a value θ_λ as an upper bound for the arborescence cost and a lower bound for the cost of all rooted walks, *i.e.*

$$c(T_\lambda, \lambda) \leq \theta_\lambda \leq c(P, \lambda) \tag{3.2}$$

for any rooted path P and $\lambda > 0$. We will later on use θ_λ to measure the quality of the solution for DVRP. The IterPCA algorithm has a time complexity of $O(|V|^3)$.

3.1 From Prize-Collecting Arborescence to Orienteering Path

Orienteering can be formulated as the problem of finding the path minimizing the reward of nodes, which are not included in the path, *i.e.*

$$\arg\min_P \pi(V - V(P)) \quad \forall P \in \mathcal{P}^r_D \tag{3.3}$$

We use Lagrange multipliers to relax the distance bound and incorporate it into the cost function.

$$\arg\min_P \pi(V - V(P)) + \frac{1}{\lambda}(d(P) - D) \quad \forall P \in \mathcal{P}^r$$

or equivalently

$$\arg\min_P \lambda \cdot \pi(V - V(P)) + d(P) - D \quad \forall P \in \mathcal{P}^r$$

for any arbitrary $\lambda > 0$. $\mathcal{P}^r$ is the set of all paths starting at the root.

Since D is a constant value, it can be dropped, further simplifying the objective function to (3.4).

$$\arg\min_P \lambda \cdot \pi(V - V(P)) + d(P) = \arg\min_P c(P, \lambda) \quad \forall P \in \mathcal{P}^r \tag{3.4}$$

In other words, this Lagrangian relaxation turns the rooted orienteering problem into the problem of finding a rooted path with cheapest prize-collecting value, which we can address with IterPCA according to (3.2).

Note: For small enough values of λ, the output arborescence will only include the root node, and for large enough values, it will include all the nodes. Also, for small enough λ, the distance cost of the arborescence is at most D. See lemma 3.1.

Lemma 3.1. *Let $\pi_{\max} := \max_{v \in V} \pi_v$ and n be the number of nodes (including the root node). If $\lambda \leq \frac{1}{n\pi_{\max}}$, then $d(T_\lambda) \leq D$.*

Proof by contradiction. Suppose $d(T_\lambda) > D$. Also, let P be an arbitrary rooted path with distance cost at most D. Since all distances including the distance bound are integers, then $d(T_\lambda) \geq D + 1$.

$$
\begin{aligned}
& D + 1 \leq d(T_\lambda) \leq d(T_\lambda) + \lambda \cdot \pi(V - V(T_\lambda)) \\
& \leq d(P) + \lambda \cdot \pi(V - V(P)) \qquad \text{according to (3.2)} \\
& \leq D + \lambda \cdot \pi(V - V(P)) \\
& \leq D + \frac{\pi(V - V(P))}{n\pi_{\max}} < D + 1 \implies D + 1 < D + 1
\end{aligned}
$$

□

The remaining problem is determining λ. Previous works show that a desirable λ results in $d(T_\lambda)$ to be equal to D. We elaborate on this case in section 3.1.1. We also discuss the lower bounds of our solution for the other cases in sections 3.1.2 and 3.1.3.

3.1.1 Case #1

The first case is when, for some λ, the distance cost of the corresponding arborescence (T_λ) is exactly equal to the distance limit (D), *i.e.*

$$\exists \lambda > 0, d(T_\lambda) = D$$

Lemma 3.2. *If $d(T_\lambda)$ is greater than or equal to D for some positive λ, then the reward of T_λ is at least the optimal orienteering reward,* i.e.

$$d(T_\lambda) \geq D \implies \pi(T_\lambda) \geq \pi^* \tag{3.5}$$

where $\pi^* := \pi(P^*)$.

Proof. From (3.2),

$$
\begin{aligned}
& c(T_\lambda, \lambda) \leq c(P^*, \lambda) \\
\implies & d(T_\lambda) + \lambda \cdot \pi(V - V(T_\lambda)) \leq d(P^*) + \lambda \cdot \pi(V - V(P^*)) \\
\implies & D + \lambda \cdot \pi(V - V(T_\lambda)) \leq d(P^*) + \lambda \cdot \pi(V - V(P^*)) \leq D + \lambda \cdot \pi(V - V(P^*)) \\
\implies & \lambda \cdot \pi(V - V(T_\lambda)) \leq \lambda \cdot \pi(V - V(P^*)) \\
\implies & \pi(V - V(T_\lambda)) \leq \pi(V - V(P^*)) \\
\implies & \pi(T_\lambda) \geq \pi^*
\end{aligned}
$$

□

Let $t \in V(T)$ be the node on the arborescence furthest from the root, *i.e.*

$$t := \arg\max_{v \in V(T)} d_{rv} \tag{3.6}$$

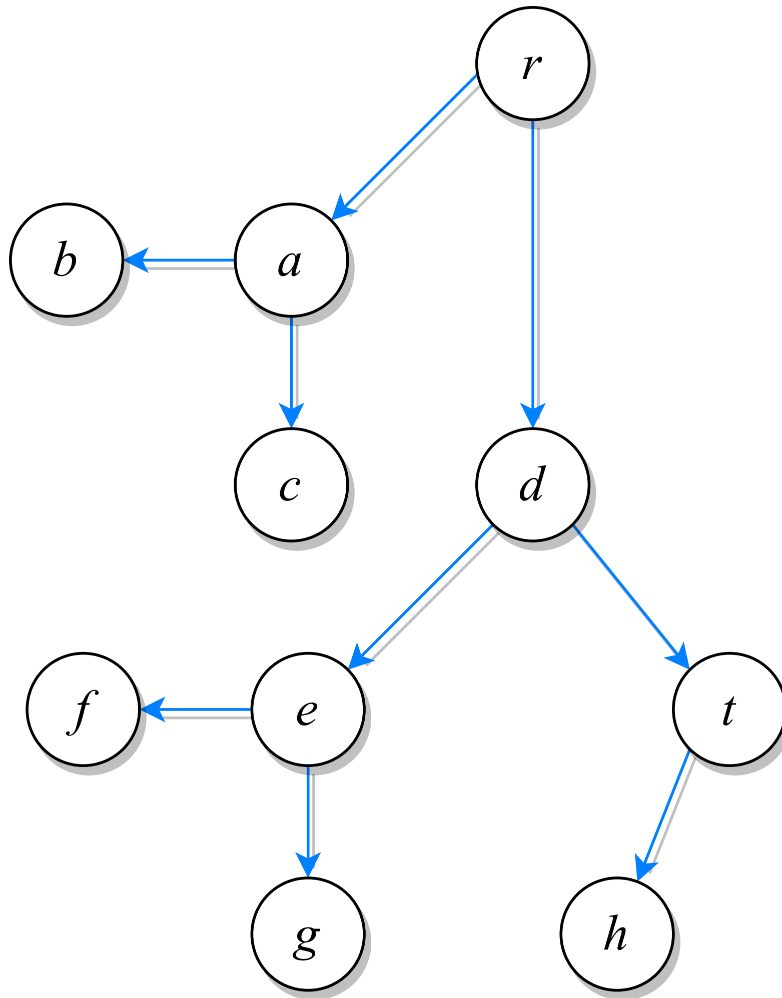


Figure 3.1: A sample arborescence. Note that in this graph, t is the furthest node from the root (r)

We can extract a path from the arborescence using the following steps:

1. Let A be the set of reverse of the arcs in T except those on the r-t path. We construct a new graph G' using the same vertices as T and edges of T and A.

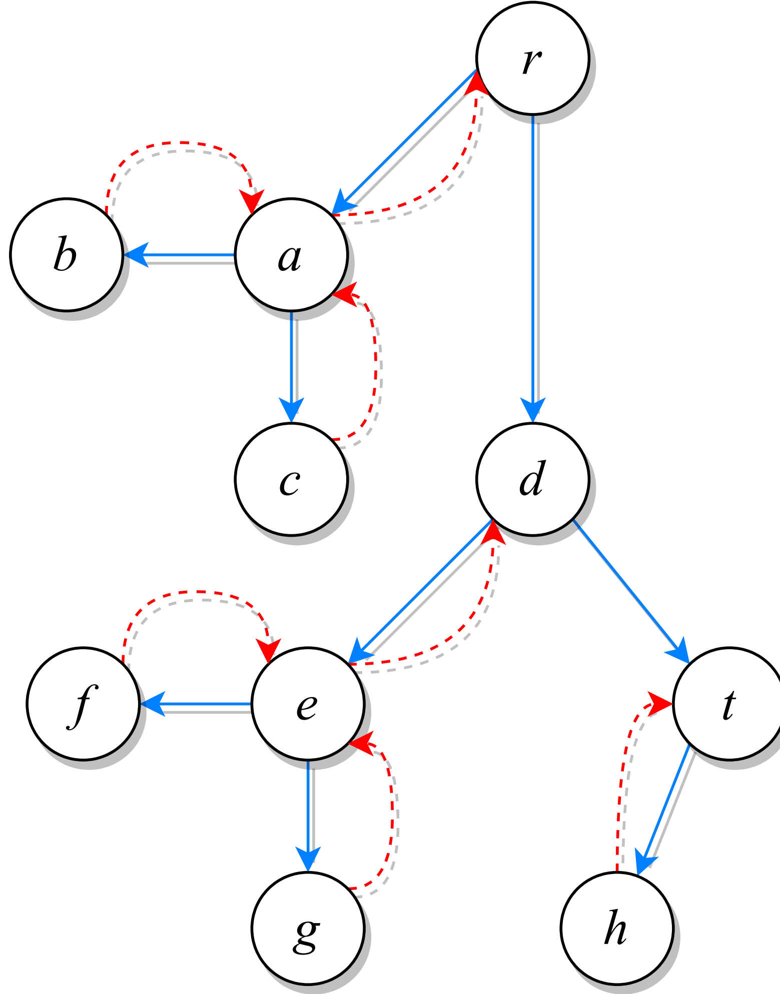


Figure 3.2: The solid arcs of the arborescence (T) in blue and the dashed arcs of A in red

The following conditions hold in G':

$$\begin{cases} \delta_{out}(r) = \delta_{in}(r) + 1 \\ \delta_{out}(t) = \delta_{in}(t) - 1 \\ \delta_{out}(v) = \delta_{in}(v) \qquad \forall v \in V(T) - \{r, t\} \end{cases} \tag{3.7}$$

2. Since all the nodes are connected to the root through T, and conditions in (3.7) hold, then Hierholzer's algorithm guarantees lemma 3.3 to be true.

Lemma 3.3. *There exists an Eulerian walk which uses every edge in G' exactly once.*

We extract one such Eulerian walk and call it W.

$$r \to a \to b \to a \to c \to a \to r \to d \to e \to f \to e \to g \to e \to d \to t \to h \to t$$

Figure 3.3: One of the possible walks of the graph in fig. 3.2

3. We then create a path P' by shortcutting past repeated nodes in W except for the last occurrence of the furthest node (t).

$$r \to a \to b \cancel{\to a \to} c \cancel{\to a \to r \to} d \to e \to f \cancel{\to e \to} g \cancel{\to e \to d \to t \to} h \to t$$
$$r \to a \to b \to c \to d \to e \to f \to g \to h \to t$$

Figure 3.4: Path extracted by shortcutting past repeated nodes

Lemma 3.4.

$$d(P') \leq 2d(T) - d_{rt} \tag{3.8}$$

Proof. Since the only edges not included in A are those on the r-t path, and by the triangle inequality the distance cost of the r-t path is at least d_{rt}:

$$d(A) \leq d(T) - d_{rt}$$
$$\implies d(T) + d(A) \leq 2d(T) - d_{rt}$$
$$\implies d(P') \leq d(W) = d(T) + d(A) \leq 2d(T) - d_{rt}$$

□

Algorithm 3.1 summarizes the steps covered in this section so far.

Algorithm 3.1 Extract path from arborescence

```
function Arb2Path(T, r, t)
    A ← Reverse of the edges of T except those on r-t path
    G' ← (V(T), T ∪ A)
    W ← Extract Eulerian walk from G'                    ▷ Step 2
    P' ← Shortcut past repeated nodes on W               ▷ Step 3
    return P'
end function
```

4. Let a and b be consecutive nodes on the path such that the section of P' starting at r and ending at a, denoted by $P'_{r \rightsquigarrow a}$, has distance cost at most D, and $P'_{r \rightsquigarrow b}$ has a distance greater than D, *i.e.*

$$d(P'_{r \rightsquigarrow a}) \leq D \tag{3.9}$$

$$d(P'_{r \rightsquigarrow b}) > D \tag{3.10}$$

Also, let P_1 be $P'_{r \rightsquigarrow a}$ and P_2 be the path created by concatenating the edge rb and $P'_{b \rightsquigarrow t}$ as shown in fig. 3.5.

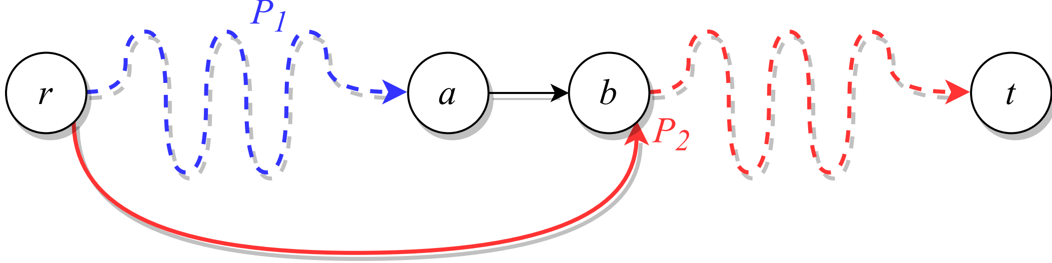


Figure 3.5: The dashed and solid lines indicate paths and edges, respectively.

Note that P_1 and P_2 together span all nodes of T.

Theorem 3.5. *P_2 is a feasible path,* i.e.

$$d(P_2) \leq D$$

Proof.

$$\begin{aligned} d(P_2) &= d_{rb} + d(P'_{b \rightsquigarrow t}) \\ &= d_{rb} + d(P') - d(P'_{r \rightsquigarrow b}) \\ &\leq d_{rb} - d_{rt} + 2D - D && \text{by (3.8) and (3.10)} \\ &= d_{rb} - d_{rt} + D \\ &\leq D && \text{by (3.6)} \end{aligned}$$

□

Finally, we return the path with the highest reward between P_1 and P_2 as our orienteering solution.

Theorem 3.6. *The maximum reward of P_1 and P_2 is at least half the reward of the optimum orienteering path.*

Proof.

$$\max\{\pi(P_1), \pi(P_2)\} \geq \frac{\pi(P_1) + \pi(P_2)}{2} = \frac{\pi(T)}{2} \tag{3.11}$$

$$\text{Lemma 3.2 and (3.11)} \implies \max\{\pi(P_1), \pi(P_2)\} \geq \frac{\pi^*}{2} \tag{3.12}$$

□

Theorem 3.6 proves that, for this case, the solution is a 2-approximation for the rooted orienteering problem.

3.1.2 Case #2

The second case is when, for any λ, the distance cost of the tree is less than the distance bound, *i.e.*

$$\forall \lambda > 0, d(T_\lambda) < D$$

In that case, we can set λ high enough so that the corresponding arborescence spans all the nodes and maximizes the collected reward. We can then use steps 1-3 in section 3.1.1 to extract P'. If the distance cost of P' is less than or

equal to the distance limit, *i.e.* $d(P') \leq D$, then we return P' as the solution. Otherwise, we use step 4 to split P' and return the path with the highest reward as our orienteering solution.

Theorem 3.7. *If, for every positive λ, the distance cost is at most D, then the orienteering solution is a 2-approximation.*

3.1.3 Case #3

It may not be immediately clear how to find a λ that leads to a distance cost of exactly D. Furthermore, there may not always exist such a value for λ. If neither of the conditions in cases 1 and 2 hold, then we can use the bisection method to find a small enough range $[\lambda_1, \lambda_2]$ and end up with two arborescences, T_1 and T_2 such that $d(T_1) < D$ and $d(T_2) > D$.

In this section, we calculate a lower bound for the reward of the paths extracted from T_1 and T_2. Our analysis here is similar to the ones in previous works such as (Friggstad & Swamy, 2017; Post & Swamy, 2017). We also use the θ values to calculate a dynamic lower bound for every instance.

Similar to the analysis done by Friggstad and Swamy (2017), our analysis requires that the furthest node from the root (t) be the same on both arborescences. For this assumption to hold, we add an extra step to our algorithm in which we iterate over all nodes and in each iteration, set that node as a guess for furthest node and discard all nodes further than this furthest node. We also have to set this node's reward to $+\infty$ to ensure that it lies on both arborescences.

Let $\pi_{\min}$ be $\min_{v:\pi_v>0} \pi_v$. Also, let $\lambda_1 = 0$ and $\lambda_2 = \frac{d_{rt}}{\pi_{\min}} + 1$, initially. For $\lambda_1 = 0$, we return the edge rt as T_1. Lemma 3.1 guarantees that the distance cost of T_1 is less than or equal to D.

At each iteration of the binary search, we calculate $\lambda_{mid} = \frac{1}{2}(\lambda_1 + \lambda_2)$ and use IterPCA to get its corresponding arborescence. If the sum of edges of this arborescence is greater than D, then we repeat the same procedure for the lower half of the interval, *i.e.* $[\lambda_1, \lambda_{mid}]$. Otherwise, we binary search the other half of the interval, *i.e.* $[\lambda_{mid}, \lambda_2]$. When the length of the interval is

less than or equal to ϵ_λ, the search terminates, and the following assumptions hold:

$$d(T_1) < D \tag{3.13}$$

$$d(T_2) > D \tag{3.14}$$

$$\lambda_2 - \lambda_1 \leq \epsilon_\lambda \tag{3.15}$$

We use a similar procedure as section 3.1.1 to extract Eulerian walks W_1 and W_2, and then we get P'_1 and P'_2 by shortcutting past repeated nodes. Next, we use algorithm 3.2 to split and extract rooted subpaths from each of these paths.

Algorithm 3.2 Greedy Splitting

Input: Path P'

Output: Feasible rooted paths $S_1, S_2, \ldots, S_k$ satisfying $\bigcup_i V(S_i) = V(P')$

1: $u_1 \leftarrow r$
2: $k \leftarrow 1$
3: $v_k \leftarrow$ last node along P' after u_k such that $d_{ru_k} + d(P'_{u_k \rightsquigarrow v_k}) - d_{rv_k} \leq D - d_{rt}$ ▷ This could be t
4: $S_k \leftarrow$ edge ru_k appended by $P'_{u_k \rightsquigarrow v_k}$
5: **while** $v_k \neq t$ **do**
6: $\quad u_{k+1} \leftarrow$ node after v_k
7: $\quad k \leftarrow k + 1$
8: $\quad v_k \leftarrow$ last node along P' after u_k such that $d_{ru_k} + d(P'_{u_k \rightsquigarrow v_k}) - d_{rv_k} \leq D - d_{rt}$
9: $\quad S_k \leftarrow$ edge ru_k appended by $P'_{u_k \rightsquigarrow v_k}$
10: **end while**

Note: Algorithm 3.2 is presented here for the sole purpose of proving the lower bound. In practice, however, we use the GetBestPath function in section 4.3 to brute-force search all acceptable subpaths and extract the path with highest reward.

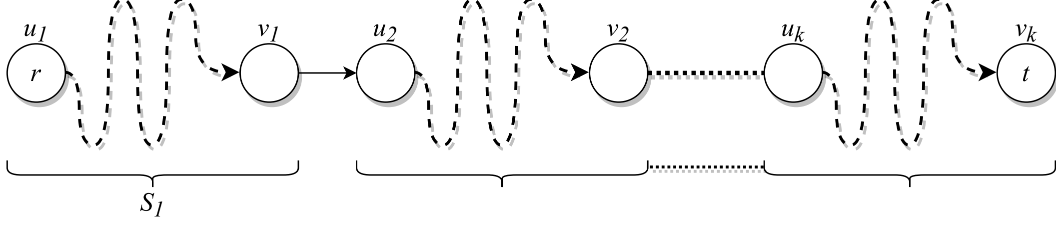


Figure 3.6: Segmentation of a path

Corollary 3.7.1. *All subpaths $(S_1, S_2, \ldots, S_k)$ are feasible,* i.e.

$$\forall 1 \leq i \leq k, d(S_i) \leq D$$

Proof.

$$d_{ru_i} + d(P'_{u_i \rightsquigarrow v_i}) - d_{rv_i} \leq D - d_{rt} \tag{3.16}$$
$$\implies d(S_i) = d_{ru_i} + d(P'_{u_i \rightsquigarrow v_i}) \leq D + d_{rv_i} - d_{rt} \leq D$$

□

Lemma 3.8. *Let k be the number of extracted segments, then*

$$k < \frac{2d(T) - 2d_{rt}}{D - d_{rt}} + 1 \tag{3.17}$$

Proof.

$$\sum_{j=1}^{k-1} [d_{ru_j} + d(P'_{u_j \rightsquigarrow v_j}) + d_{v_j u_{j+1}} - d_{ru_{j+1}}] > (k-1)(D - d_{rt})$$
$$\implies d(P'_{r \rightsquigarrow u_k}) - d_{ru_k} > (k-1)(D - d_{rt})$$
$$\implies d(S_k) + d_{ru_k} - d_{rt} + d(P'_{r \rightsquigarrow u_k}) - d_{ru_k} > (k-1)(D - d_{rt}) + d(S_k) + d_{ru_k} - d_{rt}$$
$$\implies d(P') - d_{rt} > (k-1)(D - d_{rt}) + d(S_k) + d_{ru_k} - d_{rt}$$
$$\geq (k-1)(D - d_{rt}) \qquad \text{by the triangle inequality}$$
$$\implies (k-1)(D - d_{rt}) < 2d(T) - 2d_{rt}$$
$$\implies k - 1 < \frac{2d(T) - 2d_{rt}}{D - d_{rt}}$$
$$\implies k < \frac{2d(T) - 2d_{rt}}{D - d_{rt}} + 1$$

□

For the rest of this section, we will argue that the best path among all those produced by splitting both P'_1 and P'_2 will be a $(3+\epsilon)$-approximation for the orienteering problem. Let α_1 and α_2 be two non-negative numbers such that:

$$\alpha_1, \alpha_2 \geq 0$$
$$\alpha_1 + \alpha_2 = 1$$
$$\alpha_1 \cdot d(T_1) + \alpha_2 \cdot d(T_2) = D$$

Note that α_1 and α_2 exist because, according to (3.13) and (3.14),

$$d(T_1) < D$$
$$d(T_2) > D$$

Also, let k_1 and k_2 be the number of segments extracted from P'_1 and P'_2, respectively.

We define the weighted average reward of T_1 and T_2 ($\overline{\pi}$) and the weighted average number of extracted segments ($\overline{k}$) as follows.

$$\overline{\pi} \coloneqq \alpha_1 \cdot \pi(T_1) + \alpha_2 \cdot \pi(T_2)$$
$$\overline{k} \coloneqq \alpha_1 k_1 + \alpha_2 k_2$$

We will show that $\overline{\pi}$ is nearly the optimum path reward (π^*), and $\overline{k}$ is always less than 3.

Lemma 3.9.

$$\overline{\pi} + \delta \geq U \geq \pi^* \tag{3.18}$$

where $\delta \coloneqq \frac{\alpha_1 \epsilon_\lambda}{\lambda_2} \cdot \pi(V - V(T_1))$ *and* $U \coloneqq \pi(V) - \frac{\alpha_1\theta_1 + \alpha_2\theta_2 - D}{\lambda_2}$.

Proof. Since T_1 and T_2 were generated using IterPCA algorithm, (3.2) holds for both of them, *i.e.*

$$c(T_1, \lambda_1) \leq \theta_1 \leq c(P^*, \lambda_1)$$
$$c(T_2, \lambda_2) \leq \theta_2 \leq c(P^*, \lambda_2)$$

If we expand the above equations for both $i = 1$ and $i = 2$,

$$\begin{aligned} c(T_i, \lambda_i) \leq \theta_i \leq c(P^*, \lambda_i) \\ \implies d(T_i) + \lambda_i \cdot \pi(V - V(T_i)) \leq \theta_i \leq d(P^*) + \lambda_i \cdot \pi(V - V(P^*)) \\ \implies \alpha_i \cdot d(T_i) + \alpha_i\lambda_i \cdot \pi(V - V(T_i)) \leq \alpha_i\theta_i \leq \alpha_i \cdot d(P^*) + \alpha_i\lambda_i \cdot \pi(V - V(P^*)) \end{aligned} \tag{3.19}$$

By summing (3.19) over $i = 1, 2$, we get:

$$\begin{aligned} \sum_{i=1}^{2}[\alpha_i \cdot d(T_i) + \alpha_i\lambda_i \cdot \pi(V - V(T_i))] \leq \sum_{i=1}^{2}\alpha_i\theta_i \leq \sum_{i=1}^{2}[\alpha_i \cdot d(P^*) + \alpha_i\lambda_i \cdot \pi(V - V(P^*))] \\ \implies D + \alpha_1\lambda_1 \cdot \pi(V - V(T_1)) + \alpha_2\lambda_2 \cdot \pi(V - V(T_2)) \leq \alpha_1\theta_1 + \alpha_2\theta_2 \\ \leq d(P^*) + [\alpha_1\lambda_1 + \alpha_2\lambda_2] \cdot \pi(V - V(P^*)) \leq D + [\alpha_1\lambda_1 + \alpha_2\lambda_2] \cdot \pi(V - V(P^*)) \\ \implies \alpha_1\lambda_1 \cdot \pi(V - V(T_1)) + \alpha_2\lambda_2 \cdot \pi(V - V(T_2)) \leq \alpha_1\theta_1 + \alpha_2\theta_2 - D \\ \leq [\alpha_1\lambda_1 + \alpha_2\lambda_2] \cdot \pi(V - V(P^*)) \end{aligned} \tag{3.20}$$

After the binary search terminates, we would have $\lambda_2 - \epsilon_\lambda \leq \lambda_1 < \lambda_2$. Therefore,

$$\begin{aligned} \alpha_1(\lambda_2 - \epsilon_\lambda) \cdot \pi(V - V(T_1)) + \alpha_2\lambda_2 \cdot \pi(V - V(T_2)) \\ \leq \alpha_1\lambda_1 \cdot \pi(V - V(T_1)) + \alpha_2\lambda_2 \cdot \pi(V - V(T_2)) \leq \alpha_1\theta_1 + \alpha_2\theta_2 - D \\ \leq [\alpha_1\lambda_1 + \alpha_2\lambda_2] \cdot \pi(V - V(P^*)) \leq [\alpha_1\lambda_2 + \alpha_2\lambda_2] \cdot \pi(V - V(P^*)) \\ \implies \alpha_1(\lambda_2 - \epsilon_\lambda) \cdot \pi(V - V(T_1)) + \alpha_2\lambda_2 \cdot \pi(V - V(T_2)) \leq \alpha_1\theta_1 + \alpha_2\theta_2 - D \\ \leq [\alpha_1\lambda_2 + \alpha_2\lambda_2] \cdot \pi(V - V(P^*)) \\ \implies \alpha_1(\lambda_2 - \epsilon_\lambda) \cdot \pi(V - V(T_1)) + \alpha_2\lambda_2 \cdot \pi(V - V(T_2)) \leq \alpha_1\theta_1 + \alpha_2\theta_2 - D \\ \leq \lambda_2 \cdot \pi(V - V(P^*)) \\ \implies \alpha_1 \cdot \pi(V - V(T_1)) - \frac{\alpha_1\epsilon_\lambda}{\lambda_2} \cdot \pi(V - V(T_1)) + \alpha_2 \cdot \pi(V - V(T_2)) \leq \frac{\alpha_1\theta_1 + \alpha_2\theta_2 - D}{\lambda_2} \\ \leq \pi(V - V(P^*)) \\ \implies \pi(V) - \alpha_1 \cdot \pi(T_1) - \alpha_2 \cdot \pi(T_2) - \delta \leq \frac{\alpha_1\theta_1 + \alpha_2\theta_2 - D}{\lambda_2} \leq \pi(V) - \pi^* \\ \implies \alpha_1 \cdot \pi(T_1) + \alpha_2 \cdot \pi(T_2) + \delta \geq \pi(V) - \frac{\alpha_1\theta_1 + \alpha_2\theta_2 - D}{\lambda_2} \geq \pi^* \\ \implies \alpha_1 \cdot \pi(T_1) + \alpha_2 \cdot \pi(T_2) + \delta \geq U \geq \pi^* \\ \implies \overline{\pi} + \delta \geq U \geq \pi^* \end{aligned}$$

□

Lemma 3.9 is a very important inequality as not only does it provide a theoretical guarantee for the solution, but as we will show in section 4.4, it also gives us a way to calculate a lower bound for every DVRP instance solution dynamically.

Corollary 3.9.1. *The δ value from* (3.18) *can be made arbitrarily small with a small choice of ϵ_λ because:*

$$\delta \leq \epsilon_\lambda \cdot [n\pi_{\max}]^2 \tag{3.21}$$

where n is the number of nodes, and $\pi_{\max} \coloneqq \max_v \pi_v$.

Proof.

$$\delta = \frac{\epsilon_\lambda \pi(V - V(T_1))}{\lambda_2} \leq \frac{\epsilon_\lambda}{\lambda_2}\pi(V - V(T_1)) \leq \frac{\epsilon_\lambda}{\lambda_2}\pi(V) \leq \frac{\epsilon_\lambda}{\lambda_2} n\pi_{\max} \tag{3.22}$$

$$d(T_2) > D \xRightarrow[\text{Modus Tollens}]{(3.1)} \lambda_2 > \frac{1}{n\pi_{\max}} \implies \frac{1}{\lambda_2} < n\pi_{\max} \tag{3.23}$$

$$(3.22),\ (3.23) \implies \delta \leq \frac{\epsilon_\lambda}{\lambda_2} n\pi_{\max} \leq \epsilon_\lambda [n\pi_{\max}]^2$$

□

Lemma 3.10. *The weighted average number of extracted segments ($\overline{k}$) is less than 3.*

$$\overline{k} < 3 \tag{3.24}$$

Proof. According to (3.17),

$$k_i < \frac{2d(T_i) - 2d_{rt}}{D - d_{rt}} + 1 \implies \alpha_i k_i < 2\alpha_i \cdot \frac{d(T_i) - d_{rt}}{D - d_{rt}} + \alpha_i \tag{3.25}$$

By summing (3.25) for $i = 1$ and $i = 2$, we get

$$\sum_{i=1}^{2} \alpha_i k_i < 2\alpha_1 \cdot \frac{d(T_1) - d_{rt}}{D - d_{rt}} + \alpha_1 + 2\alpha_2 \cdot \frac{d(T_2) - d_{rt}}{D - d_{rt}} + \alpha_2 \tag{3.26}$$

Since the furthest node (t) is the same on both T_1 and T_2, we can further

simplify (3.26) as follows:

$$\sum_{i=1}^{2} \alpha_i k_i < \frac{2}{D - d_{rt}} \sum_{i=1}^{2} \alpha_i d(T_i) + (1 - \frac{2d_{rt}}{D - d_{rt}}) \sum_{i=1}^{2} \alpha_i$$
$$\implies \overline{k} < \frac{2\alpha_1 \cdot d(T_1) + 2\alpha_2 \cdot d(T_2)}{D - d_{rt}} - \frac{2(\alpha_1 + \alpha_2) \cdot d_{rt}}{D - d_{rt}} + \alpha_1 + \alpha_2$$
$$\implies \overline{k} < \frac{2D}{D - d_{rt}} - \frac{2d_{rt}}{D - d_{rt}} + 1 \implies \overline{k} < 3$$

□

Intuitively, since the average reward of the trees is at least $\pi^* - \delta$, and the average number of paths produced when applying pruning to these trees is less than 3, then the best path obtained through pruning has value of at least $\frac{1}{3}[\pi^* - \delta]$. We prove this statement and present a lower bound for the orienteering solution in theorem 3.11.

Theorem 3.11. *The highest reward among the acceptable paths extracted from P'_1 and P'_2 is a $(3+\epsilon)$-approximation for the rooted orienteering problem where $\epsilon = 4n^2\epsilon_\lambda\pi_{\max}$.*

Proof. If we assign a weight of $\alpha_i/3$ to segment S^i_j from path P'_i and calculate the weighted sum, then according to (3.24)

$$\sum_{i=1}^{2} \sum_{j=1}^{k_i} \frac{\alpha_i}{3} = \frac{1}{3} \sum_{i=1}^{2} \alpha_i k_i = \frac{\overline{k}}{3} < 1 \tag{3.27}$$

Also, let $\upsilon \coloneqq \max_{1 \le i \le 2} \max_{1 \le j \le k_i} \pi(S^i_j)$, then

$$\forall 1 \le i \le 2 \forall 1 \le j \le k_i \quad \pi(S^i_j) \le \upsilon \implies \frac{\alpha_i}{3}\pi(S^i_j) \le \frac{\alpha_i \upsilon}{3}$$

If we sum over all values of i and j, by (3.27), we would get:

$$\sum_{i=1}^{2}\sum_{j=1}^{k_i}\frac{\alpha_i}{3}\pi(S_j^i) \leq \sum_{i=1}^{2}\sum_{j=1}^{k_i}\frac{\alpha_i \upsilon}{3} \leq \upsilon\sum_{i=1}^{2}\sum_{j=1}^{k_i}\frac{\alpha_i}{3} < \upsilon \tag{3.28}$$

$$\begin{aligned}
\implies \upsilon > \sum_{i=1}^{2}\sum_{j=1}^{k_i}\frac{\alpha_i}{3}\pi(S_j^i) &= \sum_{i=1}^{2}\frac{\alpha_i}{3}\pi(P_i') \\
&= \sum_{i=1}^{2}\frac{\alpha_i}{3}\pi(T_i) && \text{as } P_i' \text{ and } T_i \text{ visit the same nodes} \\
&= \frac{\overline{\pi}}{3} \\
&\geq \frac{1}{3}(\pi^* - \delta) && \text{by (3.18)} \\
&= \frac{\pi^*}{3}(1 - \frac{\delta}{\pi^*})
\end{aligned}$$

The optimal path is at least as good as the trivial path that connects the root node to the node with maximum reward, *i.e.* $\pi^* \geq \pi_{\max}$, therefore,

$$\begin{aligned}
\upsilon > \frac{\pi^*}{3}(1 - \frac{\delta}{\pi^*}) &\geq \frac{\pi^*}{3}(1 - \frac{\delta}{\pi_{\max}}) \\
&\geq \frac{\pi^*}{3}(1 - \frac{n^2\epsilon_\lambda\pi_{\max}^2}{\pi_{\max}}) && \text{by (3.21)} \\
&= \pi^*(\frac{1 - n^2\epsilon_\lambda\pi_{\max}}{3}) \\
&\geq \pi^*(\frac{1}{3 + 4n^2\epsilon_\lambda\pi_{\max}}) = \pi^*(\frac{1}{3+\epsilon})
\end{aligned}$$

□

3.2 Pseudocode

In this section, we present the summary of what we have presented in chapter 3 so far as pseudocode.

Algorithm 3.3 Binary Search Orienteering

```
1: function BisectionOrienteering(V, r, d, Π, D, ε_λ)
2:     π_min ← min_{v:Π[v]>0} Π[v]
3:     d_max ← max_{v∈V} d_{rv}
4:     λ_max ← d_max/π_min + 1
5:     U_max ← 0
6:     for all v ∈ V do
7:         Π_max[v] ← λ_max · Π[v]
8:     end for
9:     T_max, θ_max ← IterPCA(V, r, d, Π_max)
10:    if d(T_max) ≤ D then                                ▷ . See section 3.1.2
11:        t ← arg max_{v∈V} d_{rv}
12:        P' ← Arb2Path(T_max, r, t)
13:        P ← GetBestPath(P', d, D, Π)
14:        U_max ← π(V)
15:        return P, U_max
16:    end if
17:    P_best ← empty path
18:    for all t ∈ V do                 ▷ t is our guess for the furthest node
19:        V' ← {v ∈ V : d_{rv} ≤ d_{rt}}
20:        λ_1 ← 0
21:        λ_2 ← d_{rt}/π_min + 1
22:        Π_1 ← ScaleRewards(V', Π, λ_1, t)
23:        Π_2 ← ScaleRewards(V', Π, λ_2, t)
24:        T_1, θ_1 ← IterPCA(V', r, d, Π_1)
25:        T_2, θ_2 ← IterPCA(V', r, d, Π_2)
26:        BinarySearch(V', r, d, Π, D, ε_λ)   ▷ This line is the bottleneck
27:        P'_1 ← Arb2Path(T_1, r, t)
28:        P'_2 ← Arb2Path(T_2, r, t)
29:        P_1 ← GetBestPath(P'_1, d, D, Π)
30:        P_2 ← GetBestPath(P'_2, d, D, Π)
31:        if T_1 = T_2 then                            ▷ . See section 3.1.1
32:            α ← 1
33:        else
34:            α ← (d(T_2) − D) / (d(T_2) − d(T_1))
35:        end if
36:        U ← π(V') − (αθ_1 + (1 − α)θ_2 − D) / λ_2
37:        if U < +∞ then
38:            U_max ← max{U_max, U}
39:        end if
40:        P_best ← arg max{π(P_best), π(P_1), π(P_2)}
41:    end for
42:    return P_best, U_max
43: end function
```

Algorithm 3.4 Binary Search

```
1: procedure BinarySearch(V′, r, d, Π, D, ε_λ)
2:     if λ_2 − λ_1 ≤ ε_λ then
3:         return
4:     end if
5:     λ_mid ← ½(λ_1 + λ_2)
6:     Π_mid ←ScaleRewards(V′, Π, λ_mid, t)
7:     T_mid, θ_mid ← IterPCA(V′, r, d, Π_mid)
8:     if d(T_mid) = D then                                        ▷
9:         λ_1 ← λ_mid
10:        λ_2 ← λ_mid
11:        T_1 ← T_mid
12:        T_2 ← T_mid
13:        θ_1 ← θ_mid
14:        θ_2 ← θ_mid
15:        return
16:    else if d(T_mid) < D then
17:        λ_1 ← λ_mid
18:        T_1 ← T_mid
19:        θ_1 ← θ_mid
20:    else
21:        λ_2 ← λ_mid
22:        T_2 ← T_mid
23:        θ_2 ← θ_mid
24:    end if
25:    BinarySearch(V′, r, d, Π, D, ε_λ)
26: end procedure
  function ScaleRewards(V′, Π, λ, t)
      for all v ∈ V′ do
          Π_λ[v] ← λ · Π[v]
      end for
      Π_λ[t] ← +∞                    ▷ So that the arborescence always includes t
      return Π_λ
  end function
```

Algorithm 3.5 Pruning Algorithm

```
function GetBestPath(P', d, D, Π)
    P_best ← empty path
    for node u on P' do
        v ← furthest node along P' and after u such that d_ru + d(P'_{u⇝v}) ≤ D
        S ← edge ru appended by P'_{u⇝v}
        P_best ← arg max{π(P_best), π(S)}
    end for
    return P_best
end function
```

3.3 Post-Processing

We added a post-processing step to further improve an existing path by attempting to add nodes not already on the path. It uses a greedy approach in which it sorts these nodes by their reward. It then finds the first node on the path, after which it can insert them as long as the distance limit is not exceeded.

Algorithm 3.6 Post processing step

```
function PostProcess(P, V, d, D, Π, r)
    for all v ∈ V − V(P) in descending order of Π[v] do
        for all node n on P do
            Insert v right after n
            if d(P) > D then
                Remove v from P
            else
                break
            end if
        end for
    end for
    return P
end function
```

3.4 Time Complexity

The BinarySearch function on line 26 has the highest asymptotic time complexity in our orienteering algorithm. The number of binary search recursions is $O\Big(\log(\frac{d_{\max}}{\pi_{\min}\epsilon_\lambda})\Big)$. At each recursion, IterPCA is run once, there-

fore, the time complexity of BinarySearch is $O\Big(|V|^3 \log(\frac{d_{\max}}{\pi_{\min}\epsilon_\lambda})\Big)$. As we have to guess the furthest node, we would need to call BinarySearch $|V|$ times. Therefore, the total time complexity of the orienteering algorithm is $O\Big(|V|^4 \log(\frac{d_{\max}}{\pi_{\min}\epsilon_\lambda})\Big)$.

Chapter 4

Putting It All Together: Approximately Solving The LP Relaxation of DVRP

Now that we have explained the binary search orienteering algorithm and column generation, we can combine them to get an approximation algorithm for the LP relaxation of DVRP. We begin by establishing the hardness of the LP relaxation itself and then propose several simple but fast heuristics to solve the orienteering subproblem. Next, we present the pseudocode of our algorithm, and finally, we prove that the quantity U introduced in chapter 3 is the approximation factor of the solution to the DVRP relaxation and that U cannot exceed $3 + \epsilon$.

4.1 Hardness of The LP Relaxation

Given a solution to (1.8), we can verify whether its corresponding objective function value ($\sum_P x_P$) equals 1 in polynomial time, as there are at most $|B| = |V|$ positive x_P variables in the basic solution. Therefore, this decision problem is **NP**. It is also **NP**-hard as we can solve any instance of the rooted Hamiltonian path problem by solving (1.8) on the same graph with distance bound D and checking to see if the objective function is 1.

Theorem 4.1. *If the optimal value of* (1.8) *equals 1, then a rooted Hamiltonian path of distance at most D exists.*

Proof by contradiction. Let $P_1, P_2, \ldots, P_N$ be N non-Hamiltonian paths such that

$$\sum_P x_P = \sum_{i=1}^{N} x_{P_i} = 1$$
$$x_{P_i} > 0 \quad \forall 1 \leq i \leq N$$

Since $P_1, P_2, \ldots, P_N$ are non-Hamiltonian, then for each path P_i, there exists a node v which is not on P_i. By writing the corresponding constraint in (1.8) for v, we would have:

$$\sum_{P:v \in P} x_P \leq \sum_P x_P - x_{P_i} = 1 - x_{P_i} < 1 \quad \forall 1 \leq i \leq N$$

which violates the constraint. □

On the other hand, assume that a Hamiltonian path exists. Let us call it P^*. If we set its corresponding x variable (x_{P^*}) to 1 while keeping other variables at 0, we get a feasible solution for (1.8) with objective function value of 1. The objective function value of (1.8) cannot go below 1, otherwise, it would violate the constraints of the LP.

Therefore, every instance of the rooted Hamiltonian path problem is reducible to the decision problem in theorem 4.1, which makes this decision problem **NP**-complete. This means that unless **P** = **NP**, one should not expect an efficient algorithm to find an optimal solution.

4.2 Heuristics

Using column generation to solve (1.8) gives us the flexibility to use any algorithm to solve the orienteering subproblem as long as that algorithm can provide us with a column with negative reduced cost. As a result, we can start column generation with simpler but faster heuristics to improve performance and use more sophisticated but slower algorithms, like BisectionORienteerING, to improve the objective function value whenever the heuristics can not. Keep in mind, however, that the heuristics we used do not provide any guarantees for the LP value.

We implemented the following algorithms:

1. The first heuristic is a greedy algorithm. We start with the root node (r) as the only node on our path and feed it into PostProcess. The output of the function would be our path.

Algorithm 4.1 First heuristic

Input: Set of vertices V, rewards $\Pi[v]$, distance matrix $[d_{uv}]$, distance limit D, root Node r
Output: Path P
$P \leftarrow$ empty path
Add r to P
$P \leftarrow$ PostProcess(P, V, d, D, Π, r)

2. In the second heuristic, we choose two nodes from $V - \{r\}$ and add them as the second and third nodes of the path. We then feed these paths into PostProcess and return the path with highest reward. In practice, this heuristic rarely, if not never, generated a path with reward higher than 1 when the first heuristic failed to do so.

Algorithm 4.2 Second heuristic

Input: Set of vertices V, rewards $\Pi[v]$, distance matrix $[d_{uv}]$, distance limit D
Output: Path P_{best}
$P_{best} \leftarrow$ empty path
for all node $i \in V - \{r\}$ **do**
  for all node $j \in V - \{r, i\}$ **do**
    $P \leftarrow$ empty path
    Add r to P
    Add i to P
    Add j to P
    $P \leftarrow$ PostProcess(P, V, d, D, Π, r)
    $P_{best} \leftarrow \arg\max\{\pi(P), \pi(P_{best})\}$
  end for
end for

3. The third heuristic, apart from minor details, is similar to the one used by Friggstad et al. (2018). In this heuristic, we, once again, start with r as the only node on the path. At each iteration, we pick the node with the highest reward to distance ratio from the last node on the path as

long as it does not violate the distance limit and append this new node to the path. We repeat this procedure until there are no more nodes to be added.

Algorithm 4.3 Third heuristic

Input: Set of vertices V, rewards $\Pi[v]$, distance matrix $[d_{uv}]$, distance limit D
Output: Path P
  $P \leftarrow$ empty path
  Add r to P
  $c \leftarrow r$
  repeat
    $done \leftarrow true$
    $V' \leftarrow \{v \in V - V(P) : d(P) + d_{cv} \leq D\}$
    if $V' \neq \emptyset$ **then**
      $n \leftarrow \arg\max_{v \in V'} \Pi[v]/d_{cv}$
      $done \leftarrow false$
      Add n to P
      $c \leftarrow n$
    end if
  until $done$ is $true$

Note: We made the observation that for very large values of D, the heuristics would keep generating paths for thousands of iterations without improving the objective function value, similar to degenerate pivots in the simplex algorithm. BisectionORienteerING was seldom called in these cases.

4.3 Algorithm

Algorithm 4.4 Column Generation for Distance-Constrained Vehicle Routing Problem

Input: Set of vertices V, root node r, distance matrix $\left[d_{uv}\right]$, distance limit D, ϵ
Output: Set of acceptable rooted paths that together cover all nodes in V, Lower bound LB

1: $n \leftarrow |V| - 1$
2: **for all** $v \in V - \{r\}$ **do** ▷ This for loop is equivalent to $\boldsymbol{A} \leftarrow I_n$
3: Add the edge rv as a path to $\boldsymbol{A}$
4: **end for**
5: $\boldsymbol{b} \leftarrow [1]_{n\times 1}$
6: $\boldsymbol{c} \leftarrow [1]_{n\times 1}$
7: $L \leftarrow LP(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$ ▷ minimize $\boldsymbol{c}^T\boldsymbol{x}$ such that $\boldsymbol{Ax} \geq \boldsymbol{b}$
8: $LB \leftarrow 1$ ▷ The lower bound for the solution of (1.8). See section 5.2.2
9: $done \leftarrow false$
10: **repeat**
11: $done \leftarrow true$
12: Solve L
13: **for all** $v \in V$ **do**
14: $\Pi[v] \leftarrow$ Dual variable corresponding to vertex v
15: **end for**
16: $P_H \leftarrow$ Rooted path generated by heuristics ▷ See section 4.2
17: **if** $\pi(P_H) > 1$ **then**
18: Add P_H as a column to $\boldsymbol{A}$
19: $c_{P_H} \leftarrow 1$
20: $L \leftarrow LP(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$
21: $done \leftarrow false$
22: **else** ▷ if heuristics are not sufficient
23: $\pi_{\max} \leftarrow \max_{v\in V} \Pi[v]$
24: $\epsilon_\lambda \leftarrow \frac{\epsilon}{4(n+1)^2\pi_{\max}}$
25: $P_O, U \leftarrow$ BisectionOrienteerING$(V, r, d, \Pi, D, \epsilon_\lambda)$
26: $P_O \leftarrow$ PostProcess(P_O, V, d, D, Π, r) ▷ See section 3.3
27: **if** $\pi(P_O) > 1$ **then**
28: $LB \leftarrow \max\{LB, \frac{\pi(V)}{U}\}$
29: Add P_O as a column to $\boldsymbol{A}$
30: $c_{P_O} \leftarrow 1$
31: $L \leftarrow LP(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{c})$
32: $done \leftarrow false$
33: **end if**
34: **end if**
35: **until** $done$ is $true$
36: $B \leftarrow$ Basis corresponding to the solution of L
37: **return** Paths corresponding to B, LB

4.4 Bounds for DVRP

As we saw in (3.18), U is an upper bound for the reward of the optimal orienteering path. As a result, if we divide the rewards of all nodes by U, then the reward of any path would be at most 1, which means the scaled dual solution ($\frac{\Pi}{U}$) is feasible. Therefore, according to theorem 2.2, the corresponding primal solution would be a U-approximation to (1.8). Equivalently, if we divide the value of (1.8) by U, the resulting quantity would be a lower bound for OPT, indicated by LB in algorithm 4.4.

Theorem 4.2. *When algorithm 3.3 can not find a path with reward greater than 1, then* U *would be at most* $3 + \epsilon$.

Proof. If algorithm 3.3 can not find a path with reward greater than 1, then

$$1 \geq \max_{1 \leq i \leq 2} \max_{1 \leq j \leq k_i} \pi(S_j^i) > \sum_{i=1}^{2} \sum_{j=1}^{k_i} \frac{\alpha_i}{3} \pi(S_j^i) = \frac{\overline{\pi}}{3} \geq \frac{U - \delta}{3} \geq \frac{U}{3}(1 - \frac{\delta}{U})$$

Since $U \geq \pi^* \geq \pi_{\max}$,

$$\begin{aligned} 1 \geq \frac{U}{3}(1 - \frac{\delta}{U}) &\geq \frac{U}{3}(1 - \frac{\delta}{\pi_{\max}}) \\ &\geq \frac{U}{3}(1 - \frac{n^2 \epsilon_\lambda \pi_{\max}^2}{\pi_{\max}}) && \text{by (3.21)} \\ &= U(\frac{1 - n^2 \epsilon_\lambda \pi_{\max}}{3}) \\ &\geq U(\frac{1}{3 + \epsilon}) && \text{similar to theorem 3.11} \end{aligned}$$

Therefore,

$$U(\frac{1}{3 + \epsilon}) \leq 1 \implies U \leq 3 + \epsilon$$

□

Keep in mind that theorem 4.2 provides a theoretical guarantee on the value of U. In chapter 5, we see that in practice, U is much closer to 1, at least in our experiments.

Chapter 5

Experiments

In this chapter, we explain the details of the experiments that we run on our implementation[1] of algorithm 4.4. In section 5.1, we describe the datasets which we used in our experiments. Next, we discuss the results including approximation factor, improvement, and running time in section 5.2. Finally, we suggest some possible optimizations in section 5.3, which can be applied to improve the implementation even further.

5.1 Datasets

We are not aware of any benchmark datasets specific to DVRP, so we generated some. We also slightly modified the datasets for related problems. We used the following datasets:

Euclidean For these datasets, we generated a number of pseudorandom points uniformly distributed over the unit interval $[0, 1]^2$ and set the euclidean distance between points as the metric. The expected length of TSP[2] tour in this case is estimated to be $0.7\sqrt{n}$ where n is the number of points (Beardwood et al., 1959; Moscato & Norman, 1994). We use this knowledge and set $n = 25, 50, 100, 200$ and the distance bound (D) to $\max\{\frac{0.7\sqrt{n}}{5}, \sqrt{2}\}$.

TSP We chose 3 datasets with known optimum TSP tour lengths from *TSPLIB*, Gerhard Reinelt's library of 110 instances of the traveling salesman prob-

[1] You can find the code at `https://github.com/ssinad/dcvr-thesis`

[2] Travelling Salesman Problem

lem.[3] We, then, set the distance bound (D) to one-fifth of the optimum TSP tour length of each dataset.

CVRP We also chose two sets of capacitated vehicle routing problem datasets: *Fisher 1994 - Set F*[4] and *Christofides and Eilon 1969 - Set E*[5]. We swept the distance bounds for these datasets as follows:

$$D = 50, 100, 200, 500, 1000$$

5.2 Results

In this section, we share some of the insights we extracted from the experiments' results. You can find the raw data at `https://github.com/ssinad/dcvr-thesis/blob/main/result.csv`.

5.2.1 Improvement

Let V_1 be the value of (1.8) right before the first time BisectionORienteerING is called. Also, let V_f be the final value of (1.8). We define improvement as the relative difference of V_1 and V_f, as demonstrated in (5.1).

$$\text{Improvement}(\%) = \frac{V_1 - V_f}{V_1} \times 100\% \tag{5.1}$$

[3] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp`
[4] `http://www.vrp-rep.org/datasets/item/2014-0011.html`
[5] `http://www.vrp-rep.org/datasets/item/2014-0010.html`

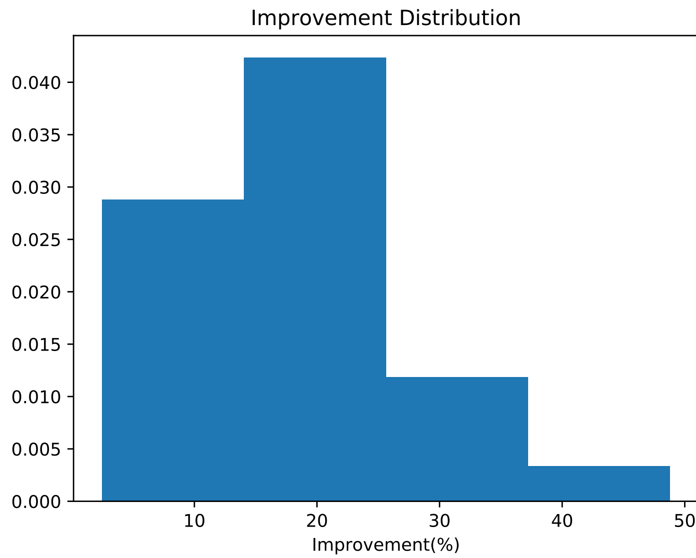


Figure 5.1: The Distribution of Improvement

Table 5.1: Statistics for Improvement

Minimum	Median	Average	Maximum	Standard Deviation
2.46	17.81	17.92	48.77	9.24

On average, BisectionORienteerING improved the objective function value by 18%.

5.2.2 Approximation Factor

The theoretical guarantee that we got for U in section 4.4 was $3+\epsilon$, but as you can see in table 5.2, the empirical results are smaller than half our theoretical guarantee.

$$U = \frac{V_f}{LB} \tag{5.2}$$

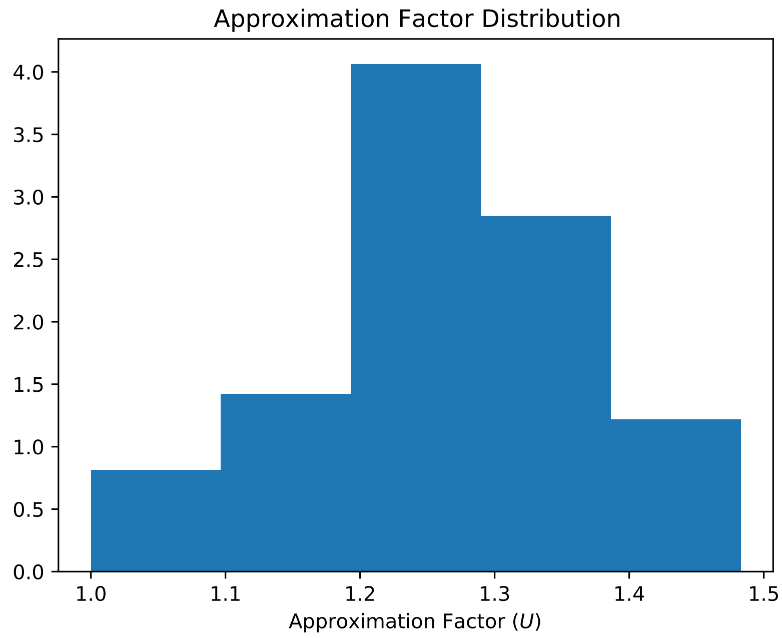
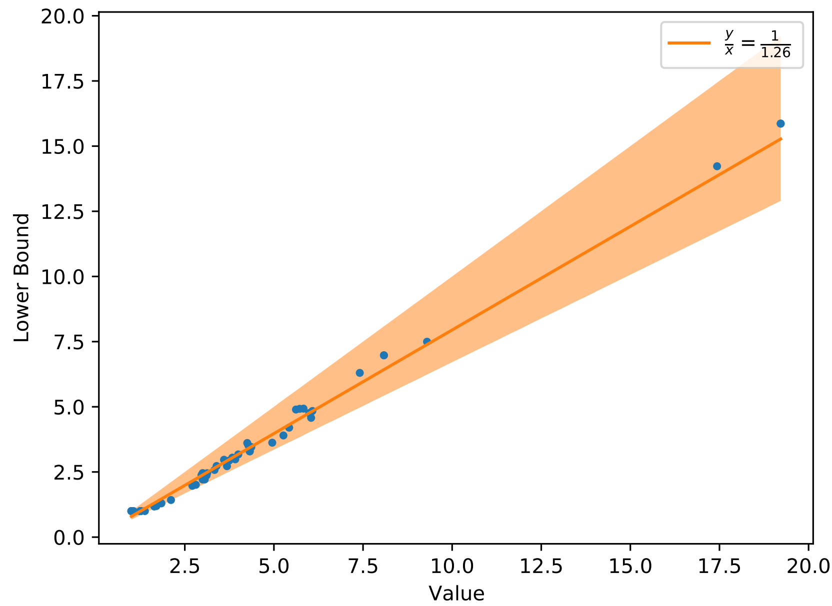


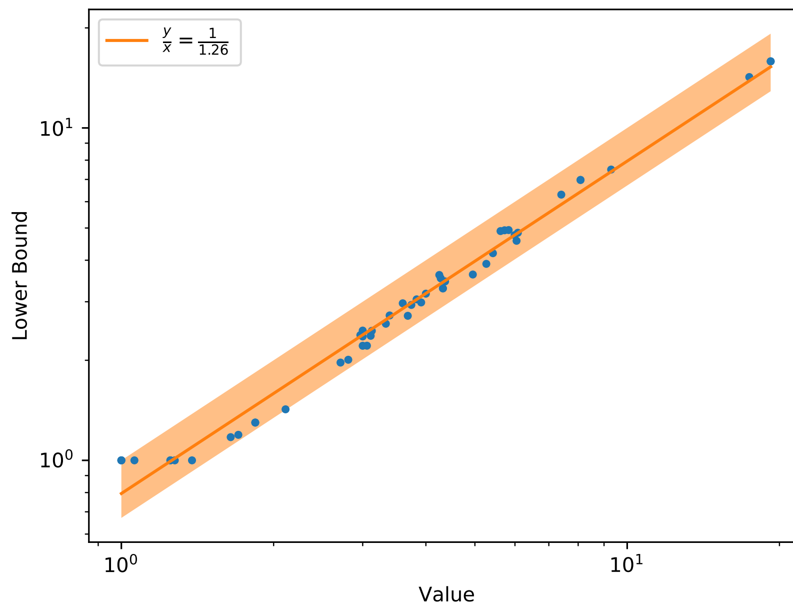
Figure 5.2: The Distribution of the approximation factor

Table 5.2: Statistics for the approximation factor (U)

Minimum	Median	Average	Maximum	Standard Deviation
1.000	1.261	1.263	1.483	0.108



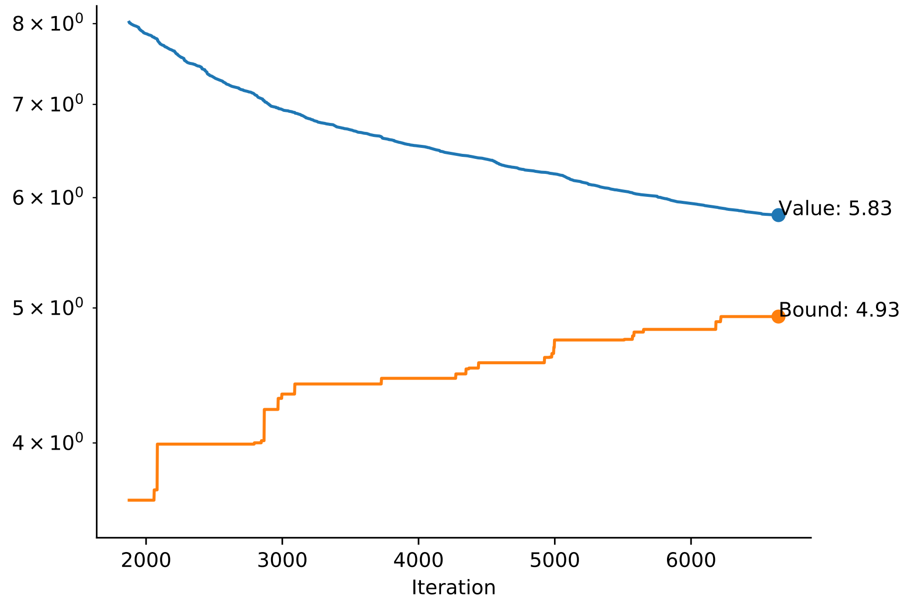
(a) Lower bound (LB) versus objective function value (V_f)



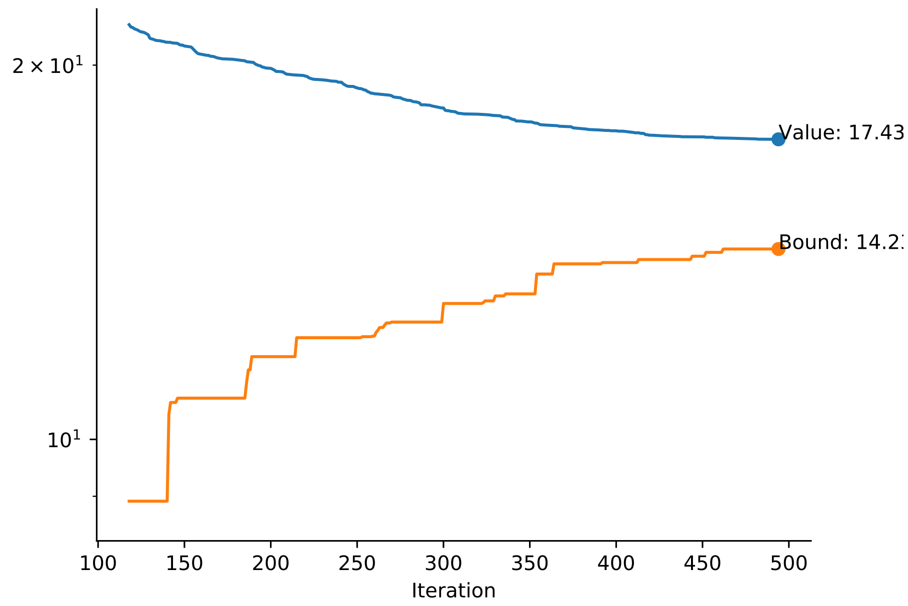
(b) The log-log plot of lower bound (LB) versus objective function value (V_f)

Figure 5.3: Distribution of the final lower bound (LB) versus the final objective function value (V_f). Each dot represents a dataset

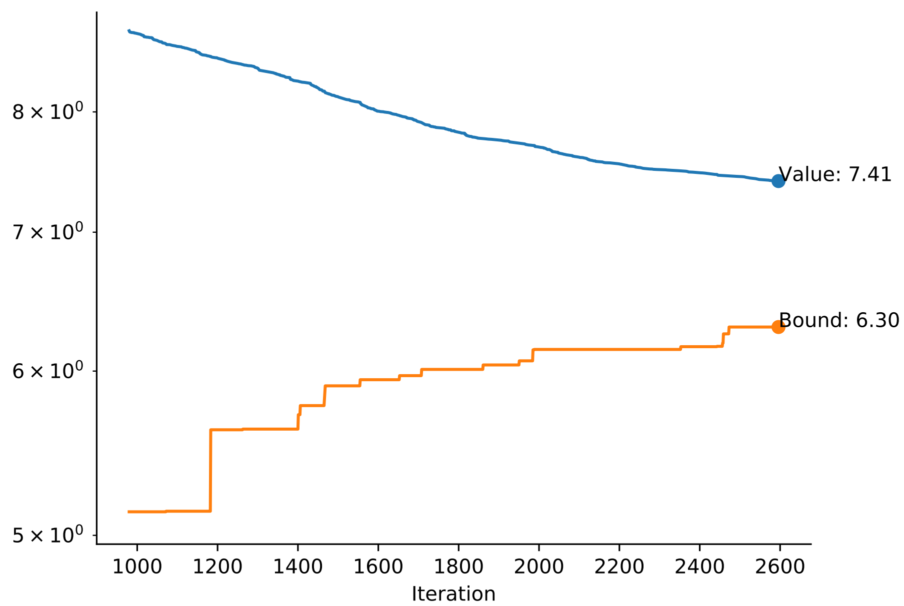
In figs. 5.4a to 5.4j, you can see how the objective function value and the lower bound change over the iterations of column generation. You can find more comprehensive versions of these plots in appendix A.2.



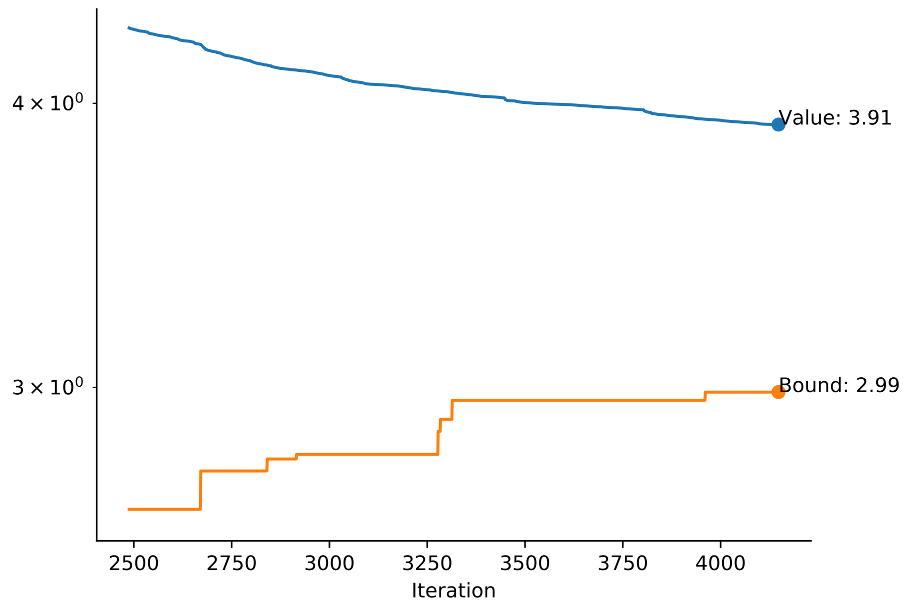
(a) Euclidean dataset with $n = 200$ points



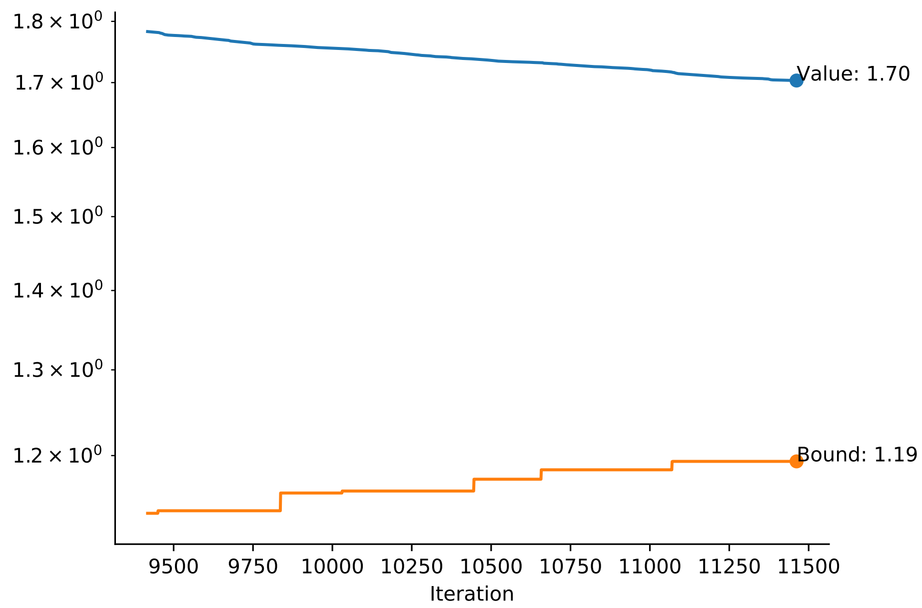
(b) M-n151-k12
$D = 50$



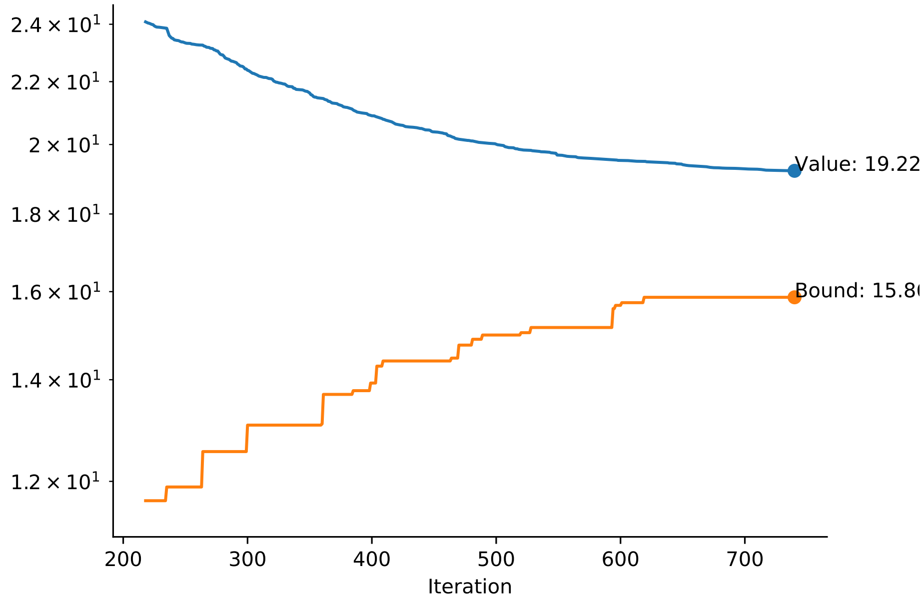
(c) M-n151-k12
$D = 100$



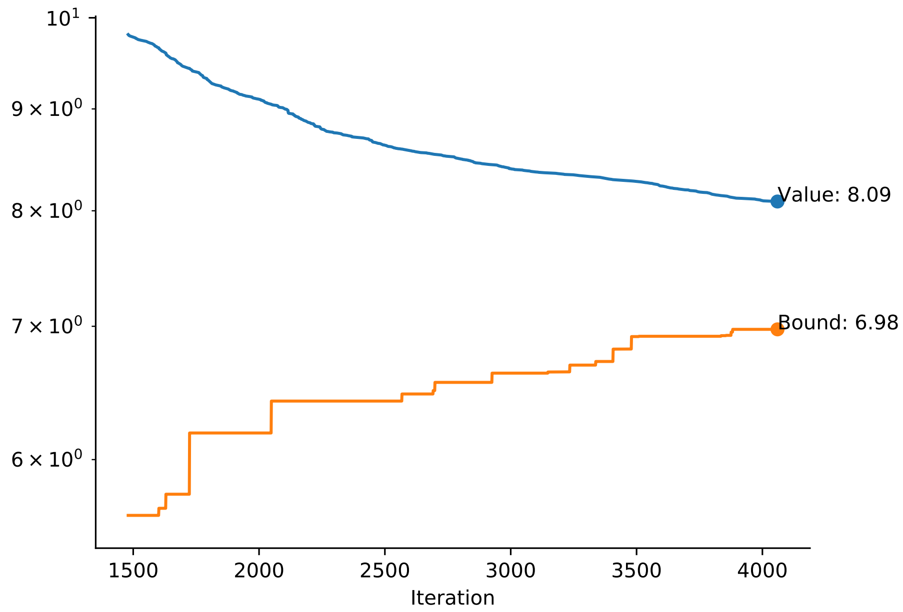
(d) M-n151-k12
$D = 200$



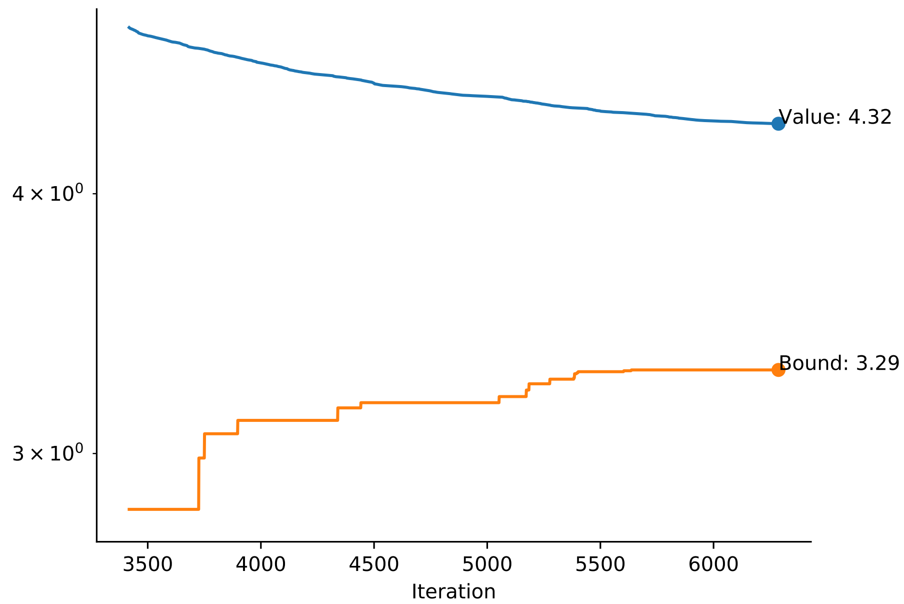
(e) M-n151-k12
$D = 500$



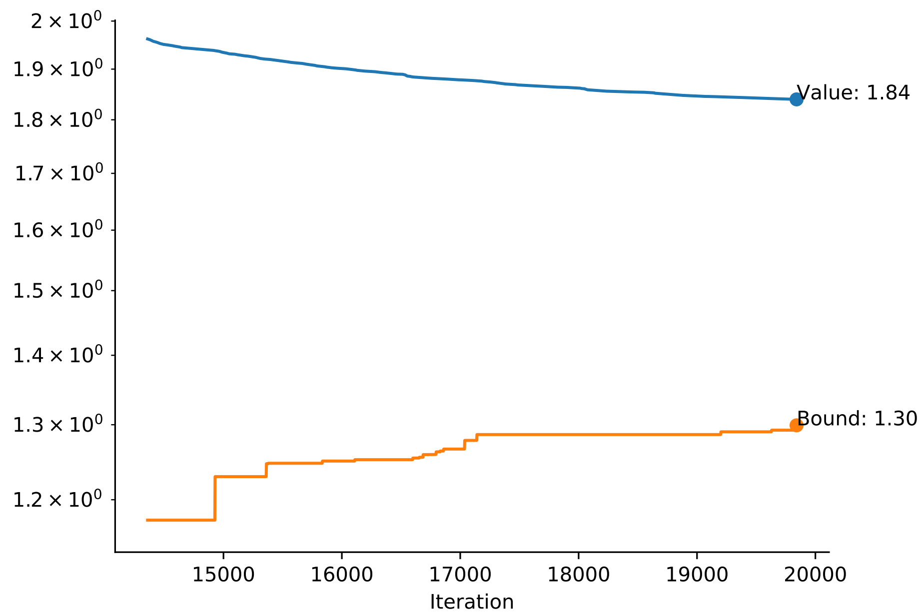
(f) M-n200-k16
$D = 50$



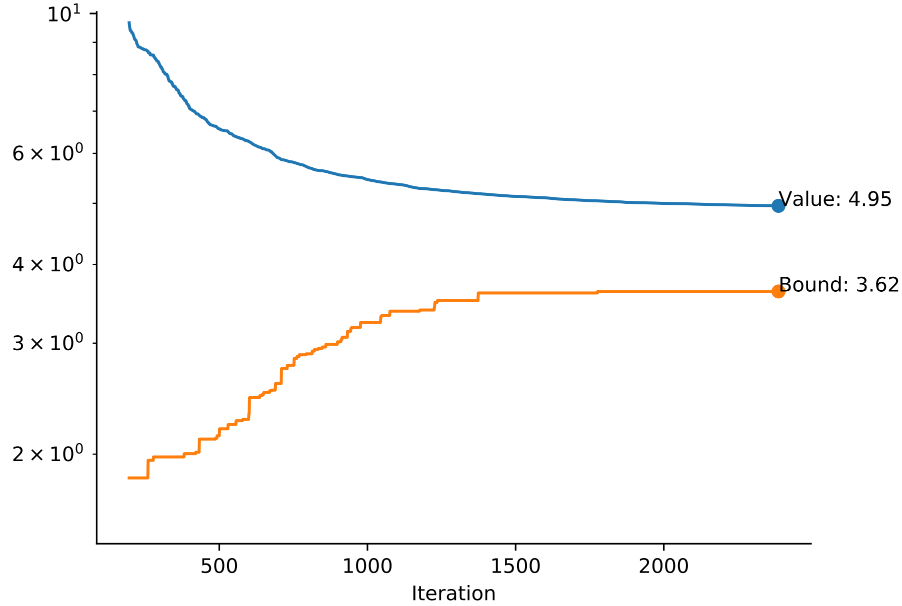
(g) M-n200-k16
$D = 100$



(h) M-n200-k16
$D = 200$



(i) M-n200-k16
$D = 500$

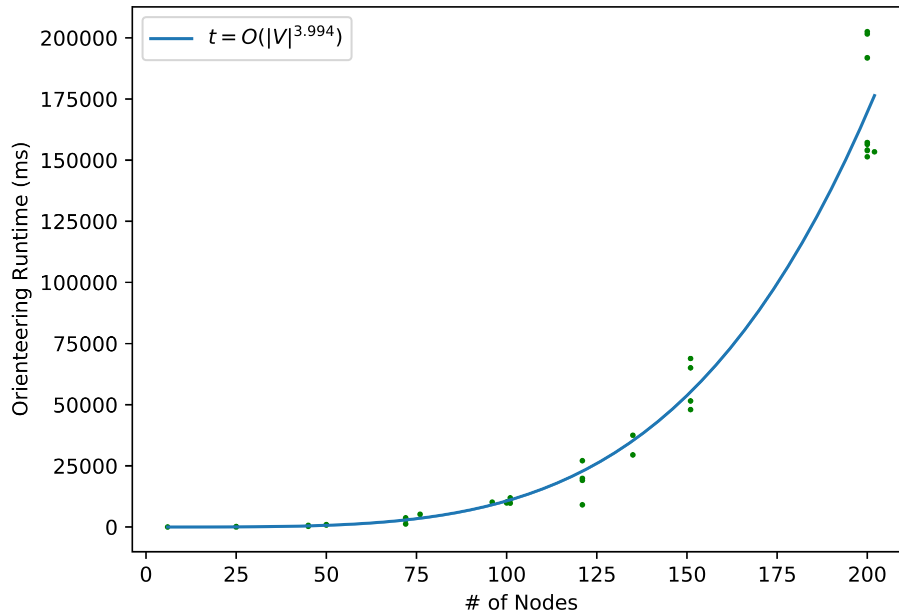


(j) gr202

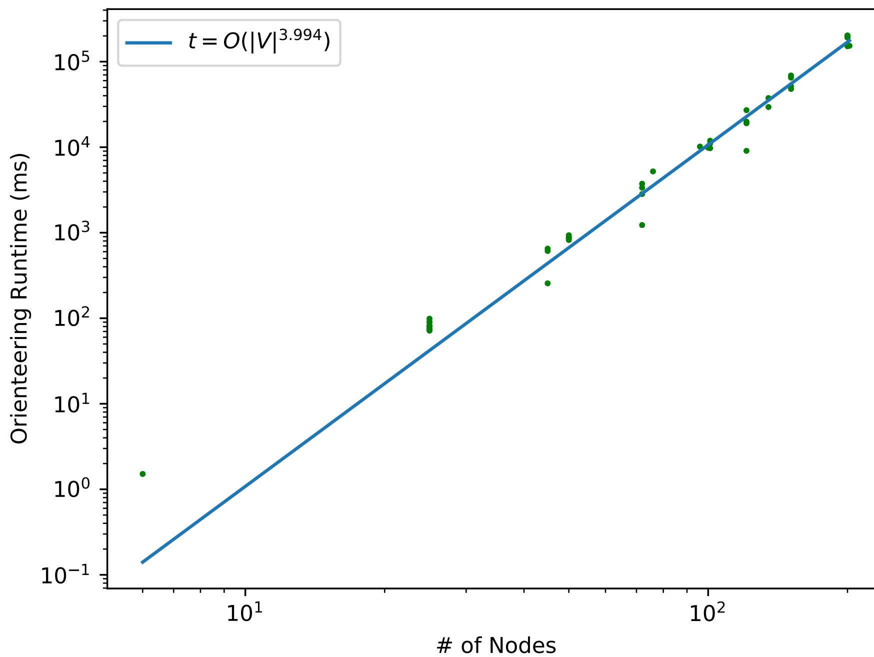
Figure 5.4: How the objective function value and the lower bound change over the iterations of column generation

5.2.3 Running Time

We measured the running time of each call to BisectionORienteerING for every dataset and averaged them to calculate the average orienteering running time. We then used nonlinear regression to fit a function of the form $y = f(x) = bx^a$ to the data.



(a) Running time versus number of nodes



(b) Log-log scale plot of running time versus number of nodes. Note that the slope of the blue line in this plot denotes the exponent of $|V|$

Figure 5.5: Average orienteering runtime versus the number of clients. Each dot represent a data set

As you can see in figs. 5.5a and 5.5b, the estimated running time complexity is proportional to $O(|V|^{3.994})$ which is very close to theoretical complexity of $O(|V|^4)$ we encountered in section 3.4.

The general DVRP time complexity depends on the effectiveness of the heuristics and the number of times BisectionORienteerING is called. On average in our experiments, BisectionORienteerING was called in around 20% of column generation iterations and comprised 90% of the running time.

5.3 Possible Optimizations

Our main focus in this work was providing a proof-of-concept implementation. However, we did come up with minor optimizations along the way, some of which are explained in sections 3.3 and 4.2. We also used a hash table to store edges in the IterPCA algorithm, and used Cantor's pairing function as the hash function. Since edges are ordered pairs of non-negative integers and Cantor's pairing function is a bijection over $N \times N \rightarrow N$, this hash function is collision-free, which enables us to check the existence of an edge in constant time.

One possible optimization to BisectionORienteerING could be parallelizing the for loop starting at line 18. By taking a closer look at this loop, we can see that all its iterations are independent of each other. Therefore, one can broadcast the required data such as the set of nodes, the distance matrix, the distance limit, and the rewards to multiple machines and reduce the resulting paths by taking the path with maximum reward.

Another improvement to BisectionORienteerING is using a smart way to guess the furthest node on the arborescences (t). Currently the algorithm has to loop over all the nodes and run BinarySearch for each guess, which causes the time complexity to be $O(|V|^4)$. However, if we can come up with smarter guesses for t, we could potentially decrease the time complexity to $o(|V|^4)$ (maybe even $O(|V|^3)$) while keeping our $(3 + \epsilon)$ approximation factor guarantee.

Chapter 6

Conclusion

DVRP is one of the more difficult variants of VRP to approximate. In this thesis, we tackled LP-relaxation of DVRP using column generation. We also discussed an approximation algorithm for solving the rooted orienteering subproblem. Furthermore, we combined these two to come up with our own practical $(3+\epsilon)$-approximation algorithm to approximately solve the LP-relaxation of DVRP. Finally, we implemented our proposed algorithm and presented the results of our experiments.

We were able to achieve an average improvement of 18%. Furthermore, we also observed that the U values were far smaller than the theoretical bound $3 + \epsilon$. We hypothesize that it may be possible to improve both practical and theoretical approximation factors, which is left as a possible line of future work. We also hypothesize the possibility of constant approximation factors for the integer program (1.7), which is left as a possible line of future work as well.

References

Arkin, E. M., Hassin, R., & Levin, A. (2006). Approximations for minimum and min-max vehicle routing problems. *Journal of Algorithms*, *59*(1), 1–18. https://doi.org/https://doi.org/10.1016/j.jalgor.2005.01.007.

Bansal, N., Blum, A., Chawla, S., & Meyerson, A. (2004). Approximation algorithms for deadline-TSP and vehicle routing with time-windows. *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 166–174.

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., & Vance, P. H. (1998). Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research*, *46*(3), 316–329. http://www.jstor.org/stable/222825.

Beardwood, J., Halton, J. H., & Hammersley, J. M. (1959). The shortest path through many points. *Mathematical Proceedings of the Cambridge Philosophical Society*, *55*(4), 299–327.

Blum, A., Chawla, S., Karger, D. R., Lane, T., Meyerson, A., & Minkoff, M. (2003). Approximation algorithms for orienteering and discounted-reward TSP. *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, 46–55. https://doi.org/10.1109/SFCS.2003.1238180.

Charikar, M., Chekuri, C., Cheung, T.-y., Dai, Z., Goel, A., Guha, S., & Li, M. (1999). Approximation algorithms for directed Steiner problems. *Journal of Algorithms*, *33*(1), 73–91.

Chekuri, C., Korula, N., & Pál, M. (2012). Improved algorithms for orienteering and related problems. *ACM Transactions on Algorithms (TALG)*, *8*(3), 23.

Desrosiers, J., & Lübbecke, M. E. (2005). A Primer in Column Generation. In G. Desaulniers, J. Desrosiers, & M. M. Solomon (Eds.), *Column Generation* (pp. 1–32). Springer US. https://doi.org/10.1007/0-387-25486-2_1.

Desrosiers, J., Soumis, F., & Desrochers, M. (1984). Routing with time windows by column generation. *Networks*, *14*(4), 545–565. https://doi.org/10.1002/net.3230140406.

Ford, L. R., & Fulkerson, D. R. (2004). A Suggested Computation for Maximal Multi-Commodity Network Flows. *Management Science*, *50*(12_supplement), 1778–1780. https://doi.org/10.1287/mnsc.1040.0269.

Friggstad, Z., Gollapudi, S., Kollias, K., Sarlós, T., Swamy, C., & Tomkins, A. (2018). Orienteering Algorithms for Generating Travel Itineraries. In Y. Chang, C. Zhai, Y. Liu, & Y. Maarek (Eds.), *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM 2018, Marina Del Rey, CA, USA, February 5-9, 2018* (pp. 180–188). ACM. https://doi.org/10.1145/3159652.3159697.

Friggstad, Z., & Swamy, C. (2014). Approximation algorithms for regret-bounded vehicle routing and applications to distance-constrained vehicle routing. *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, 744–753.

Friggstad, Z., & Swamy, C. (2017). Compact, Provably-Good LPs for Orienteering and Regret-Bounded Vehicle Routing. In F. Eisenbrand & J. Koenemann (Eds.), *Integer Programming and Combinatorial Optimization* (pp. 199–211). Springer International Publishing.

Garey, M. R., & Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

Gilmore, P. C., & Gomory, R. E. (1961). A linear programming approach to the cutting-stock problem. *Operations research*, *9*(6), 849–859.

Gilmore, P. C., & Gomory, R. E. (1963). A linear programming approach to the cutting stock problem—Part II. *Operations research*, *11*(6), 863–888.

Grötschel, M., Lovász, L., & Schrijver, A. (2012). *Geometric algorithms and combinatorial optimization* (Vol. 2). Springer Science & Business Media.

Guenin, B., Könemann, J., & Tuncel, L. (2014). *A gentle introduction to optimization*. Cambridge University Press.

Halperin, E., & Krauthgamer, R. (2003). Polylogarithmic Inapproximability. *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, 585–594. https://doi.org/10.1145/780542.780628.

Moscato, P., & Norman, M. G. (1994). An analysis of the performance of traveling salesman heuristics on infinite-size fractal instances in the euclidean plane. *ORSA Journal on Computing*.

Nagarajan, V., & Ravi, R. (2011). The directed orienteering problem. *Algorithmica*, *60*(4), 1017–1030.

Nagarajan, V., & Ravi, R. (2012). Approximation Algorithms for Distance Constrained Vehicle Routing Problems. *Networks*, *59*(2), 209–214. https://doi.org/10.1002/net.20435.

Post, I., & Swamy, C. (2017). *Combinatorial Algorithms for Rooted Prize-Collecting Walks and their Applications to Orienteering* [Manuscript]. http://www.math.uwaterloo.ca/~cswamy.

Toth, P., & Vigo, D. (Eds.). (2001). *The Vehicle Routing Problem*. Society for Industrial; Applied Mathematics.

Appendix A

Solving DVRP Integer Program

In order to solve the integer program (1.7), we extracted the paths which corresponded to the basic solution of (1.8). We then fed those paths into CPLEX[1] mixed integer optimizer to get an integer solution. Providing guarantees for the integer program (1.7) could be a possible line of future work.

A.1 Integrality Gap

$$\text{Integrality Gap} = \frac{\text{Integral LP Value}}{\text{Fractional LP Value}} \tag{A.1}$$

[1] **"url –https://www.ibm.com/analytics/cplex-optimizer"**.

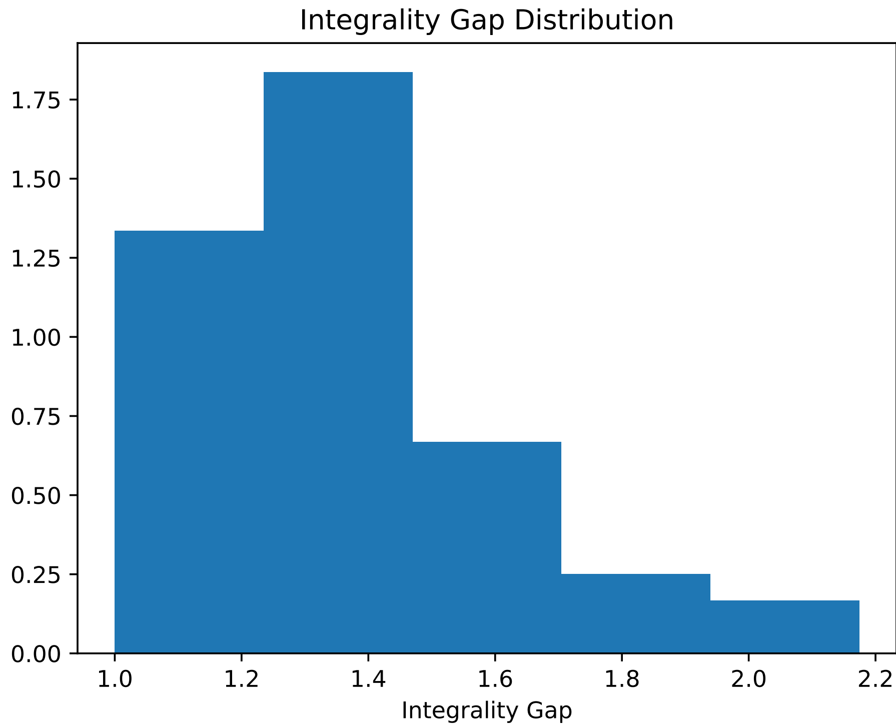


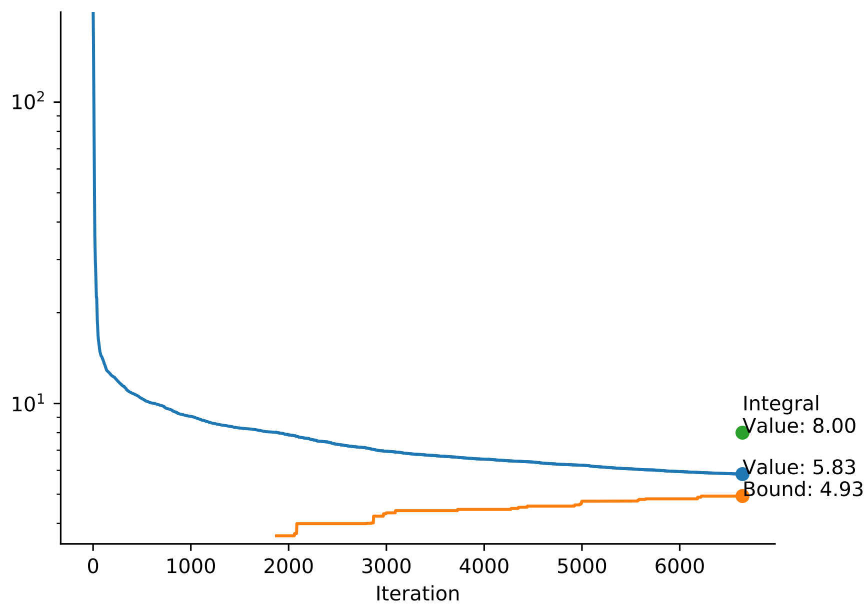
Figure A.1: The Distribution of Integrality Gap

Table A.1: Statistics for Integrality Gap

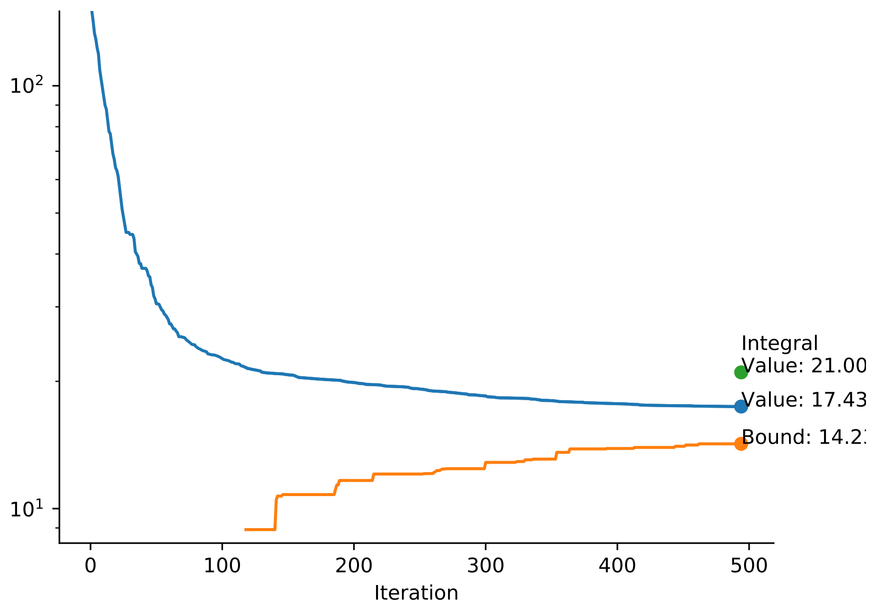
Minimum	Median	Average	Maximum	Standard Deviation
1.00	1.33	1.37	2.17	0.27

A.2 Objective Function Values and Lower Bounds

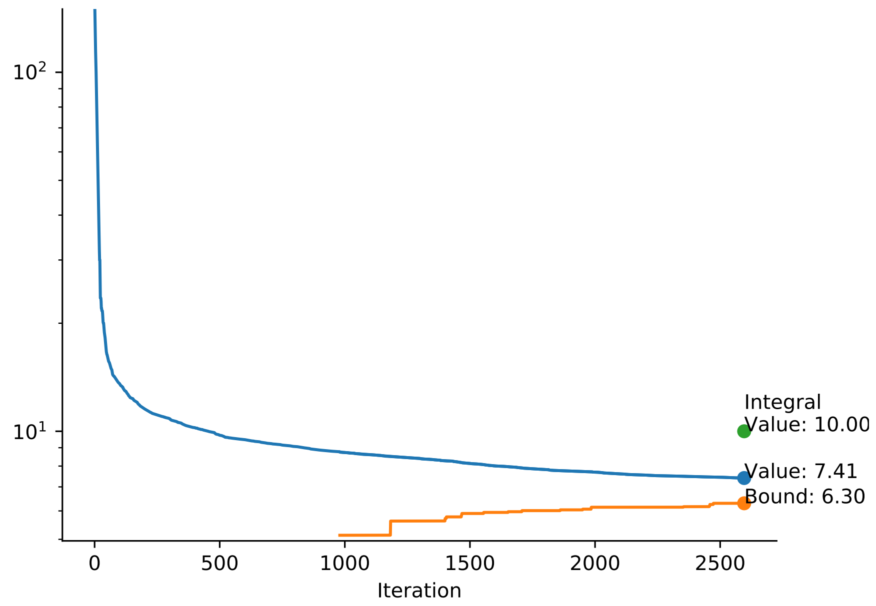
Figures A.2a to A.2j demonstrate how the objective function value of (1.8) and the lower bound change over the course of column generation iterations. The green dots denote the integer solution to (1.7). The lower bound gets a non-trivial value only after the first call to BisectionORienteerING.



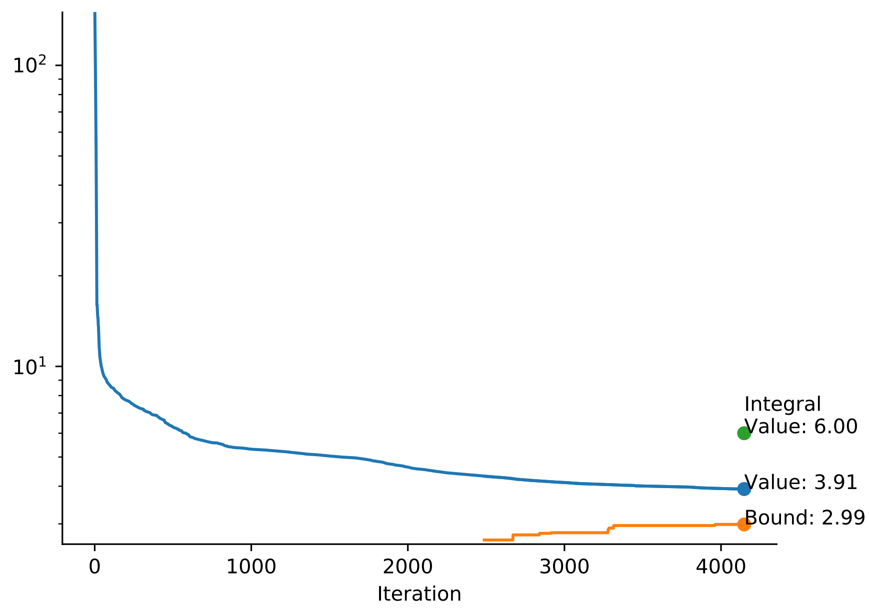
(a) Euclidean dataset with $n = 200$ points



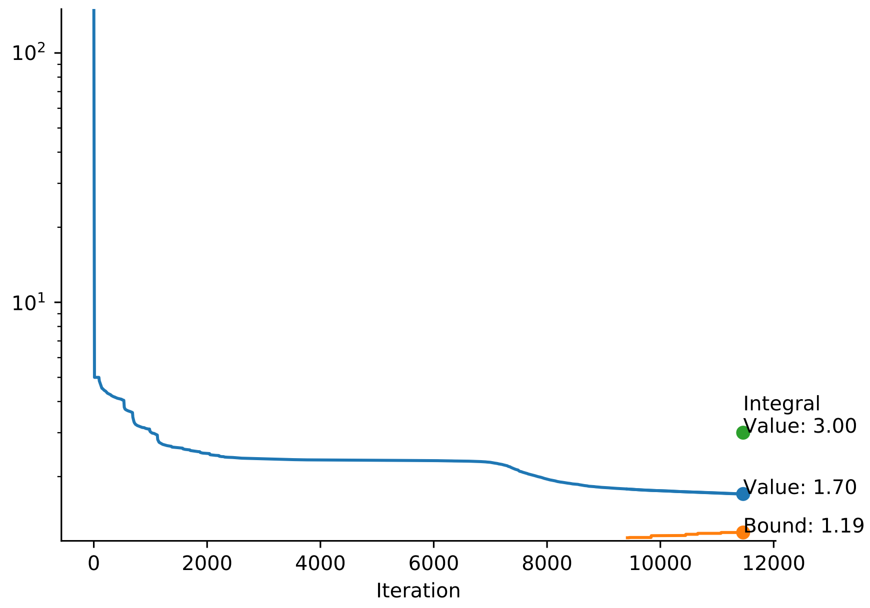
(b) M-n151-k12
$D = 50$



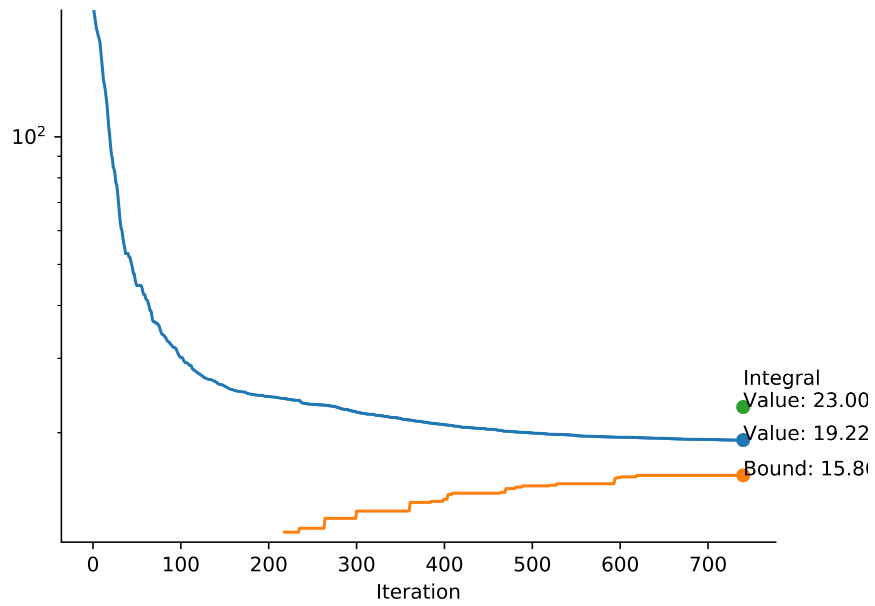
(c) M-n151-k12
$D = 100$



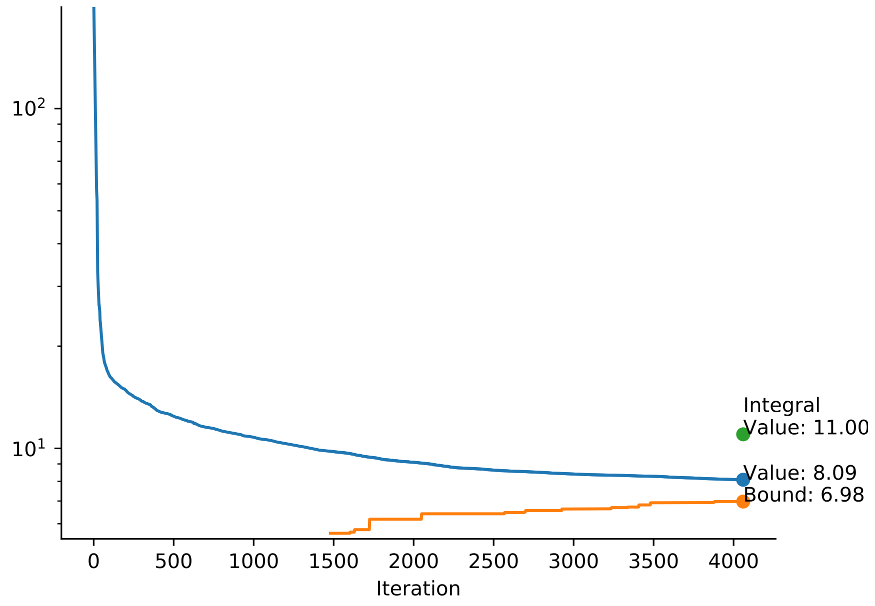
(d) M-n151-k12
$D = 200$



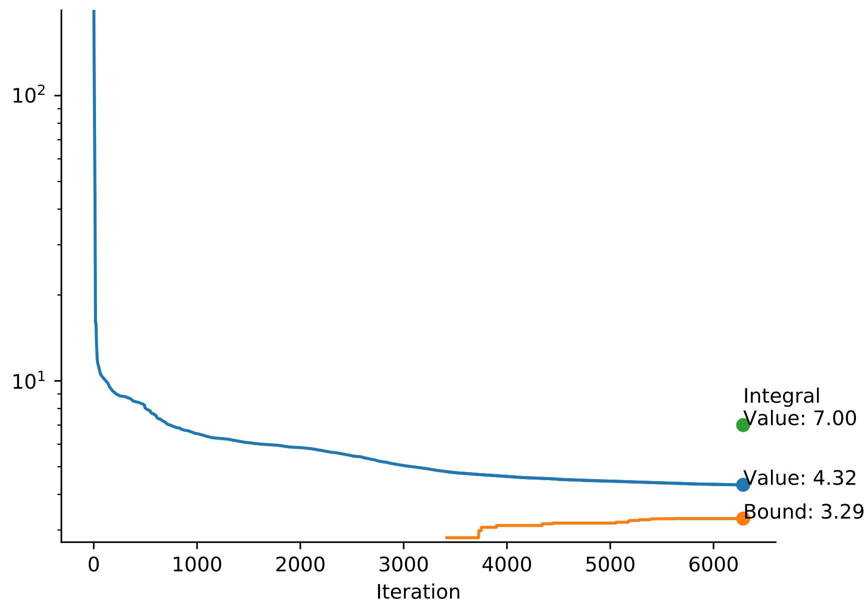
(e) M-n151-k12
$D = 500$



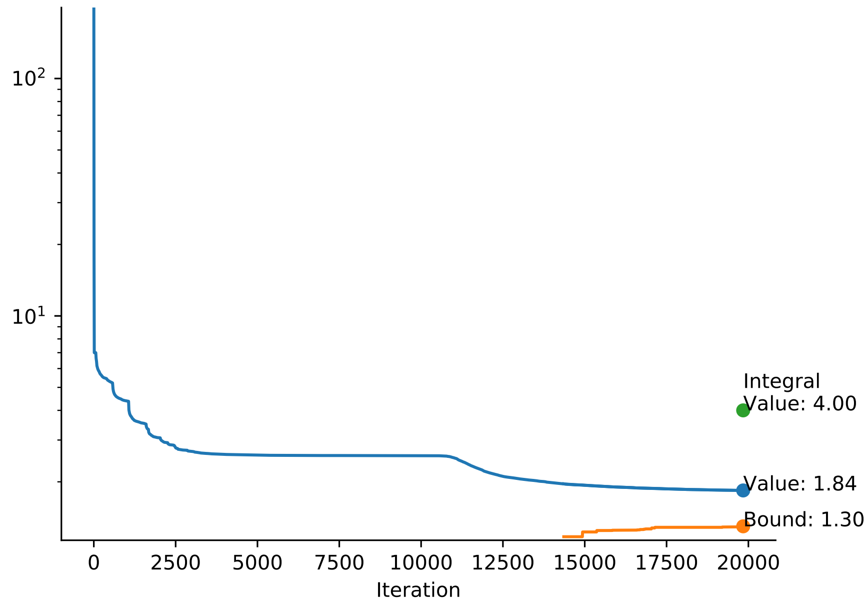
(f) M-n200-k16
$D = 50$



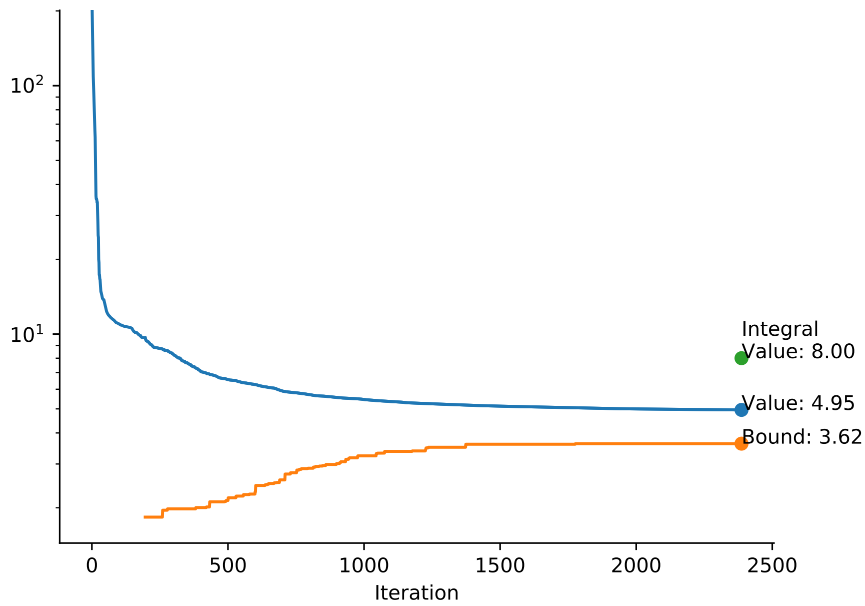
(g) M-n200-k16
$D = 100$



(h) M-n200-k16
$D = 200$



(i) M-n200-k16
$D = 500$

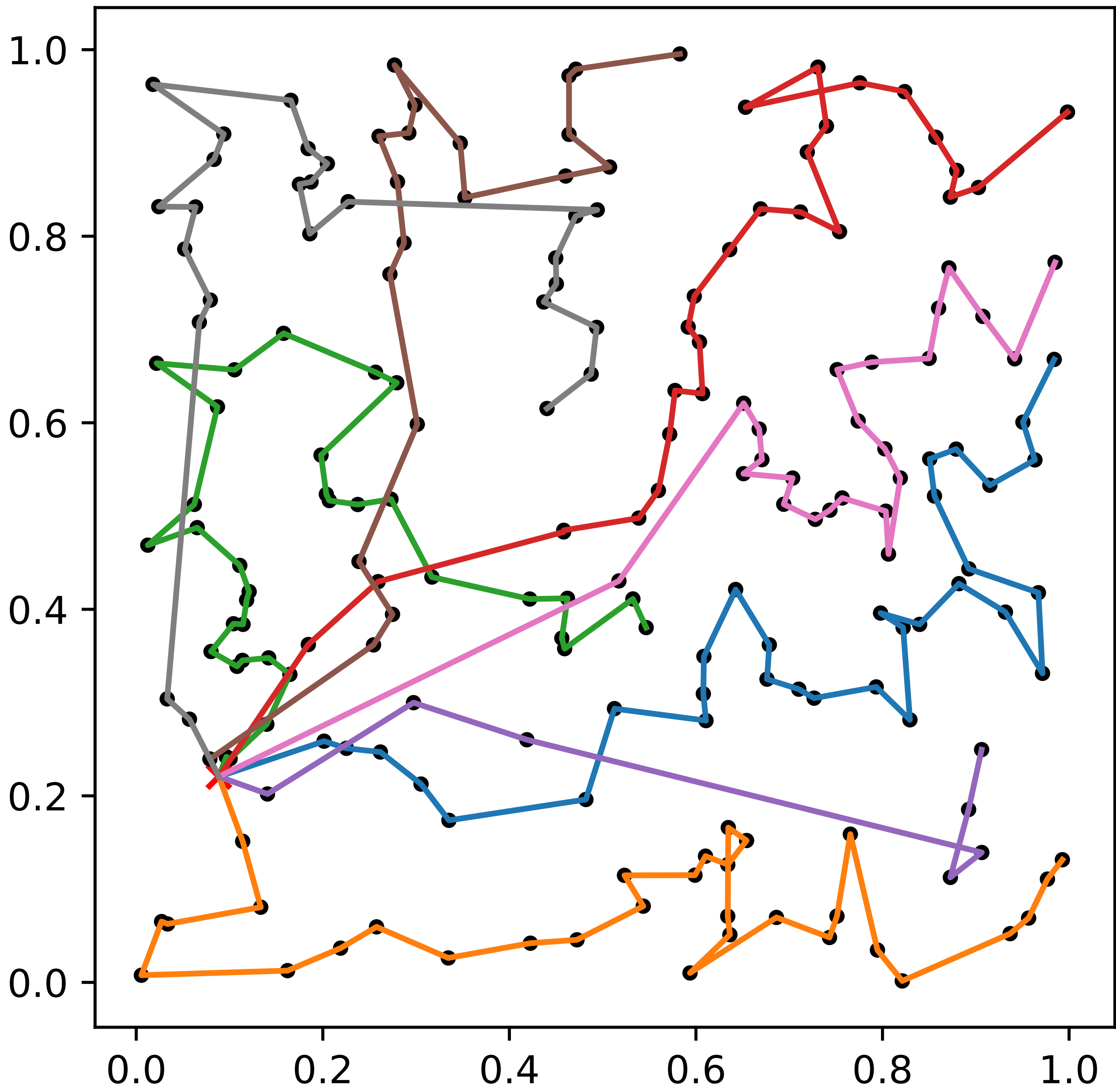


(j) gr202

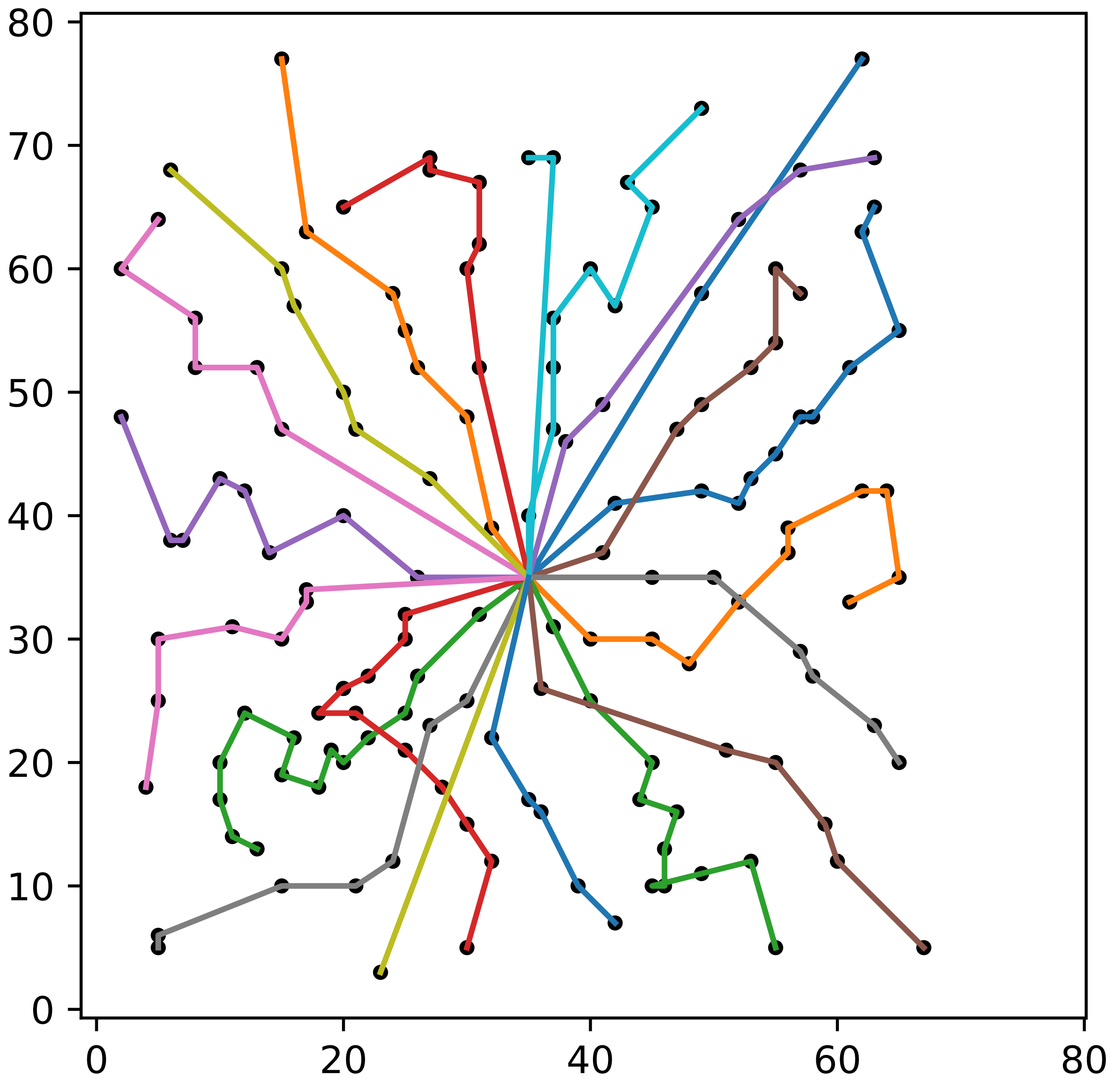
Figure A.2: How the objective function value of (1.8) and the lower bound change over the course of column generation iterations. The green dots denote the integer solution to (1.7)

A.3 Path Visualizations

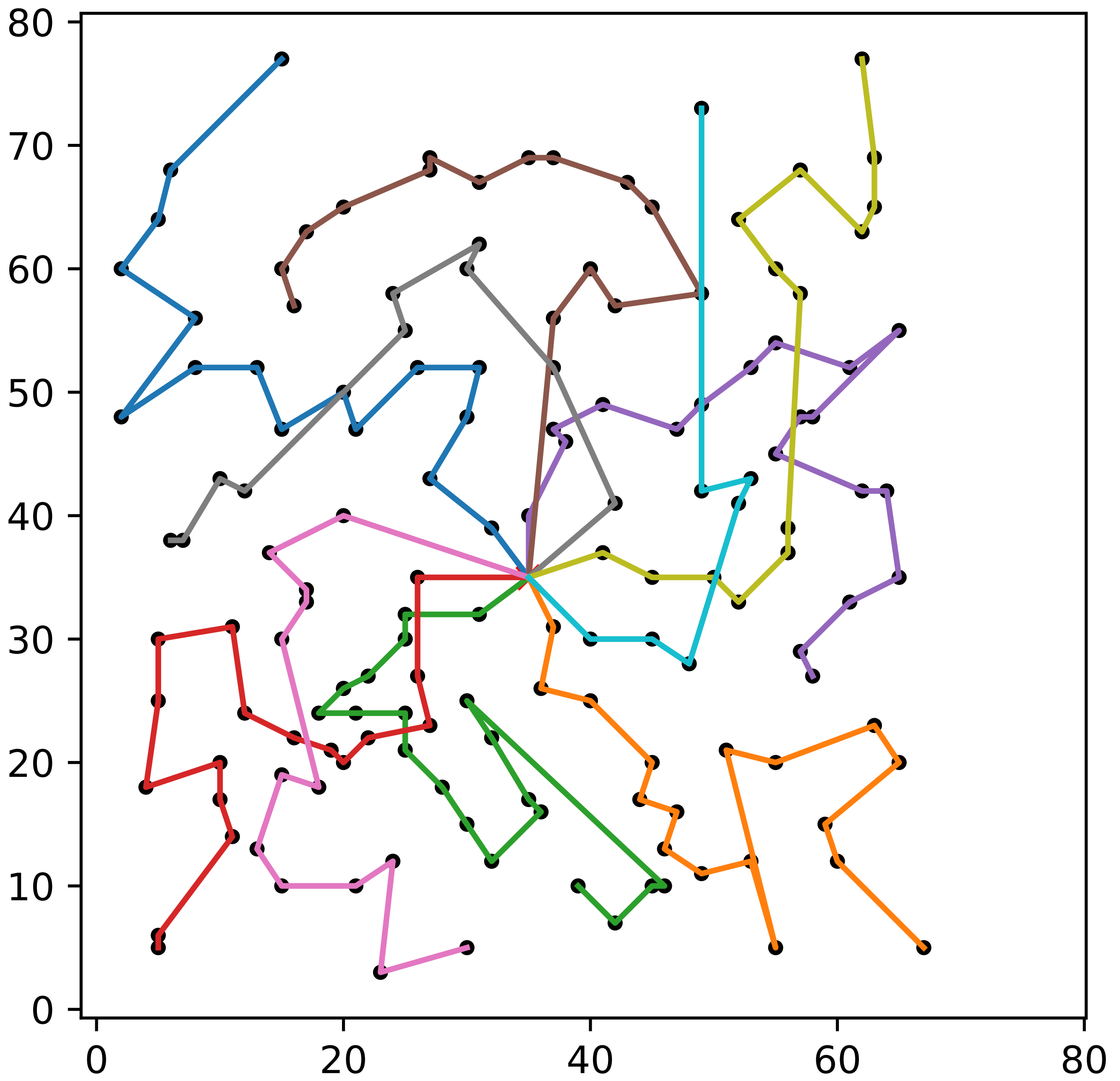
In figs. A.3a to A.3j, different paths are marked by different colors. Also, the depot is marked by a red x.



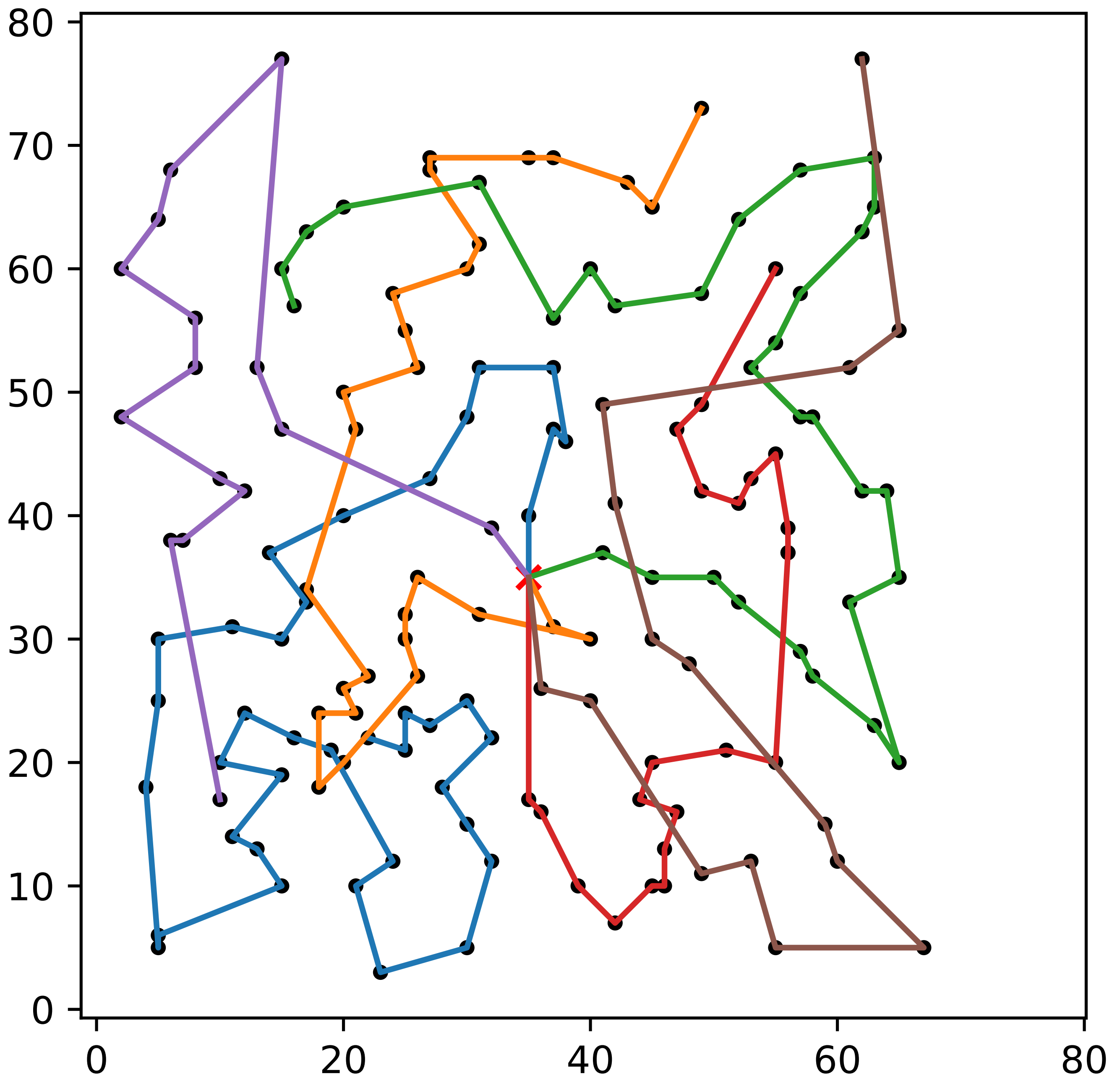
(a) Euclidean dataset with $n = 200$ points



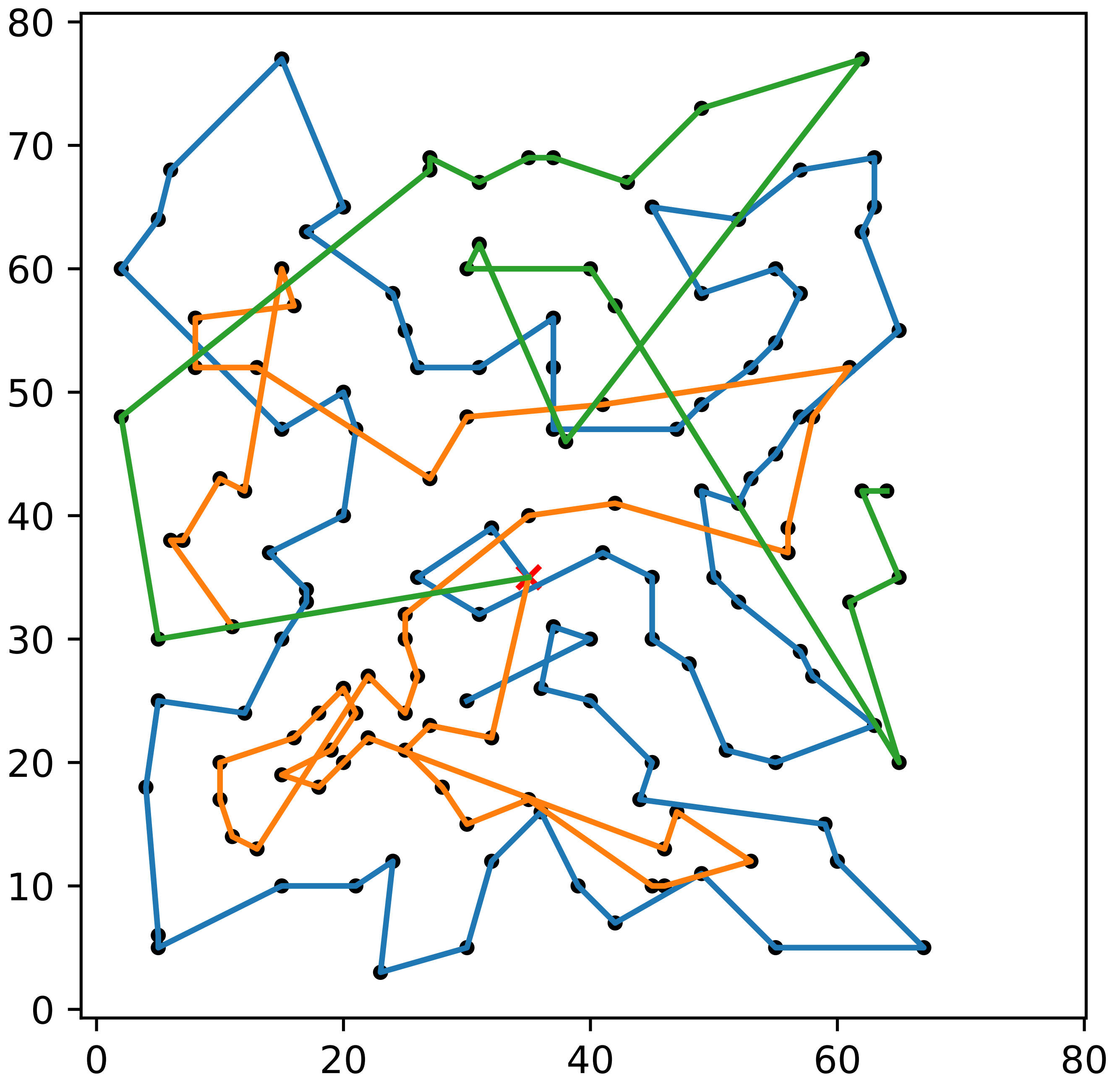
(b) M-n151-k12
$D = 50$



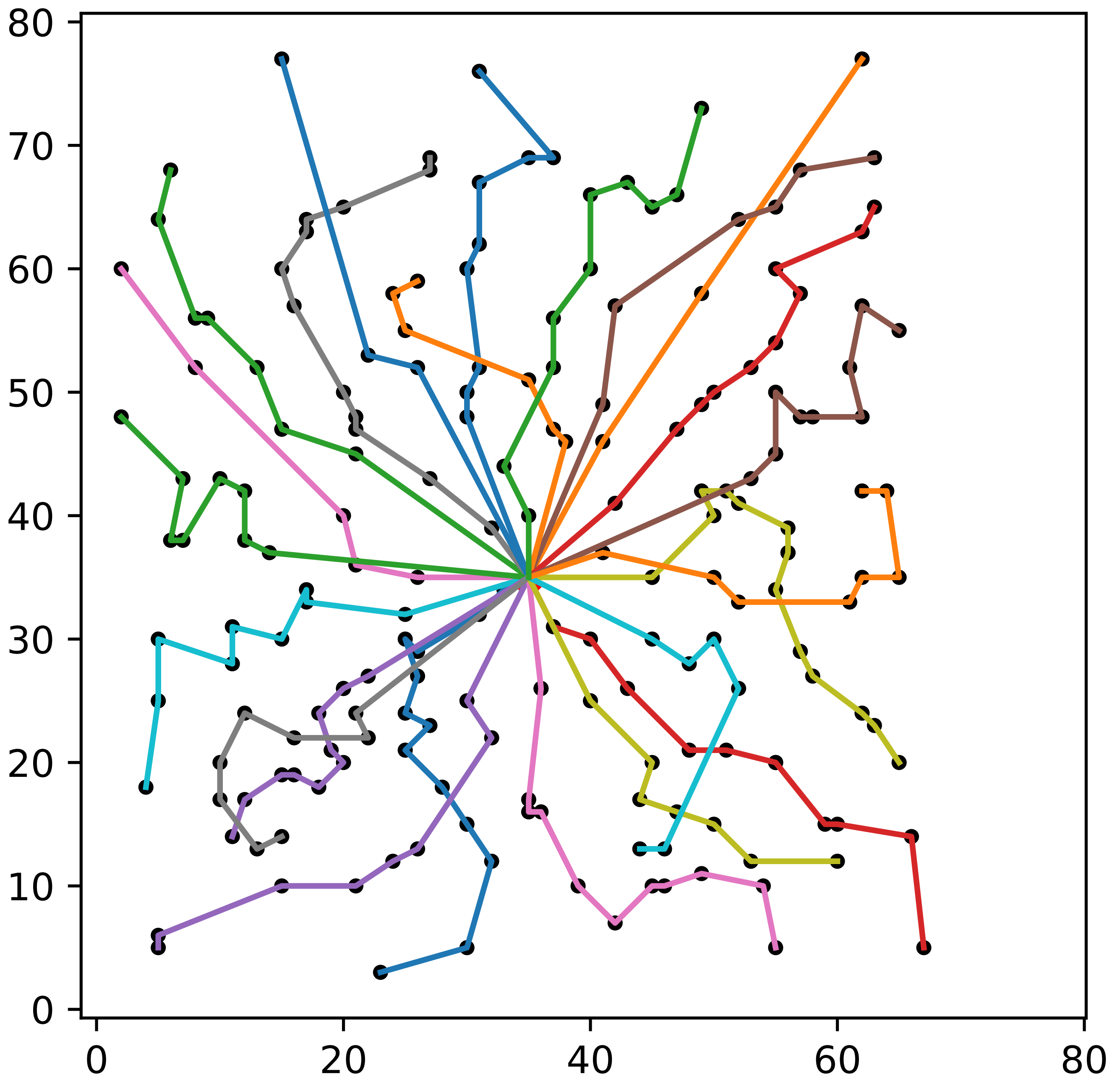
(c) M-n151-k12
$D = 100$



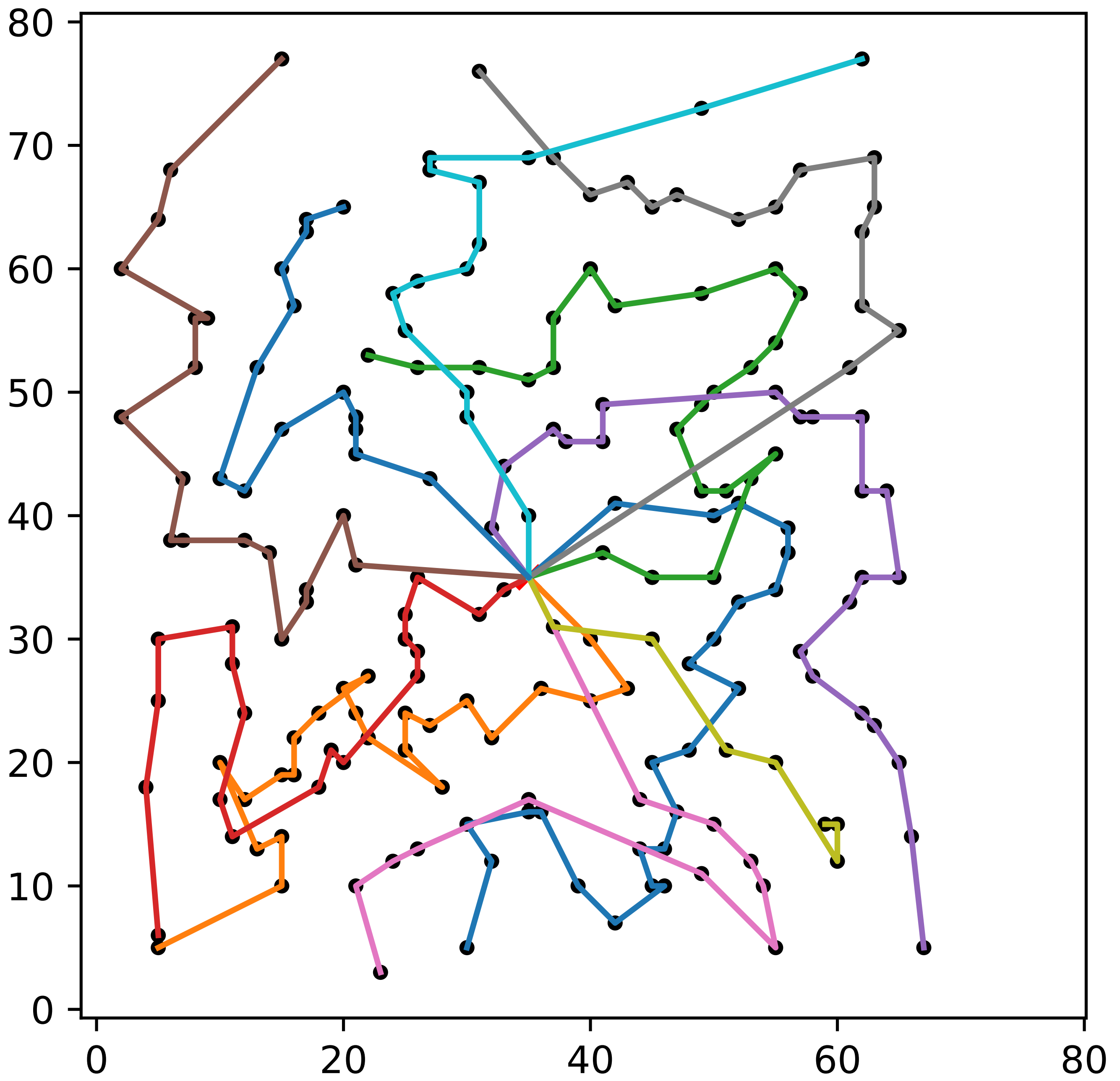
(d) M-n151-k12
$D = 200$



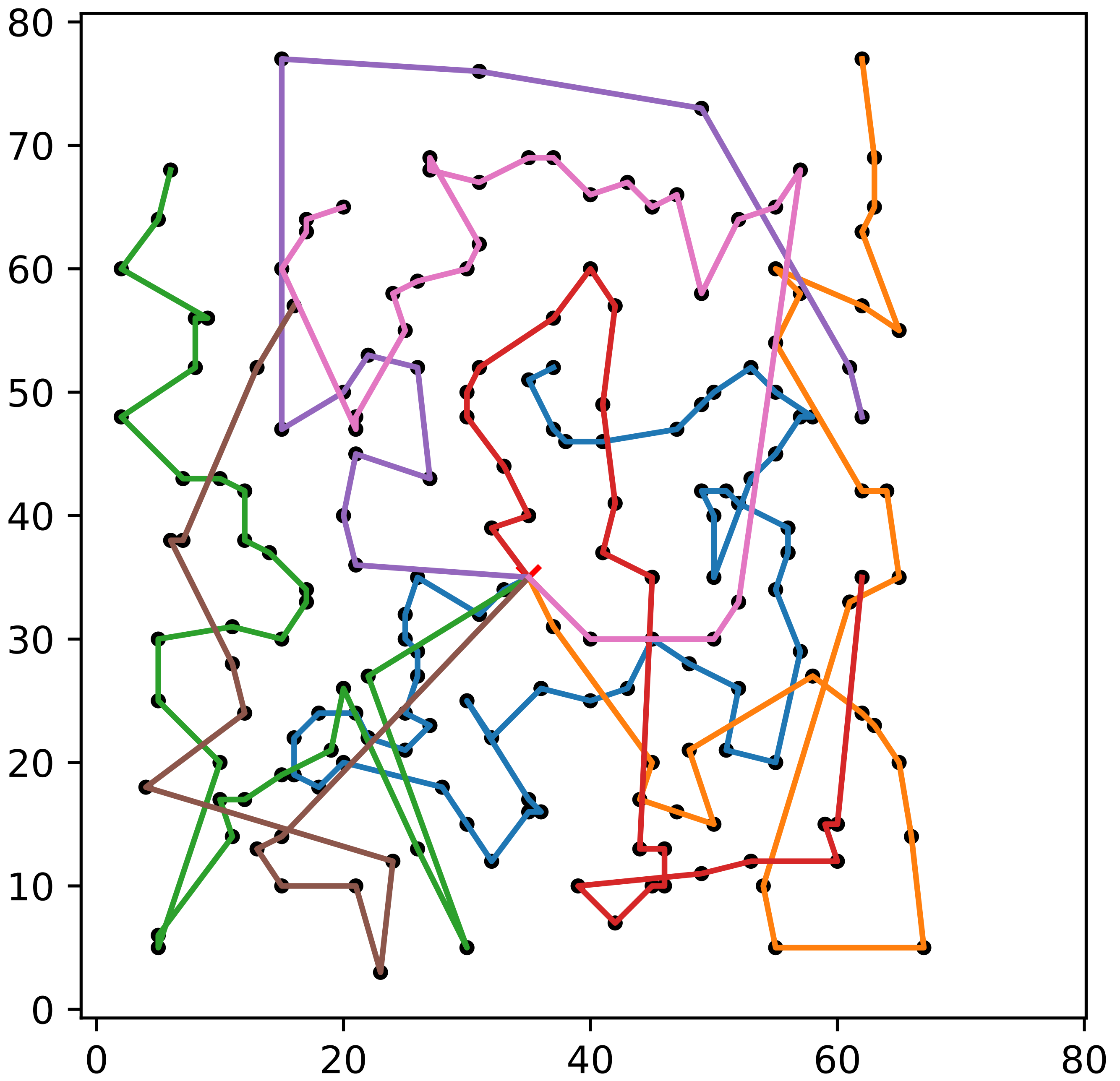
(e) M-n151-k12
$D = 500$



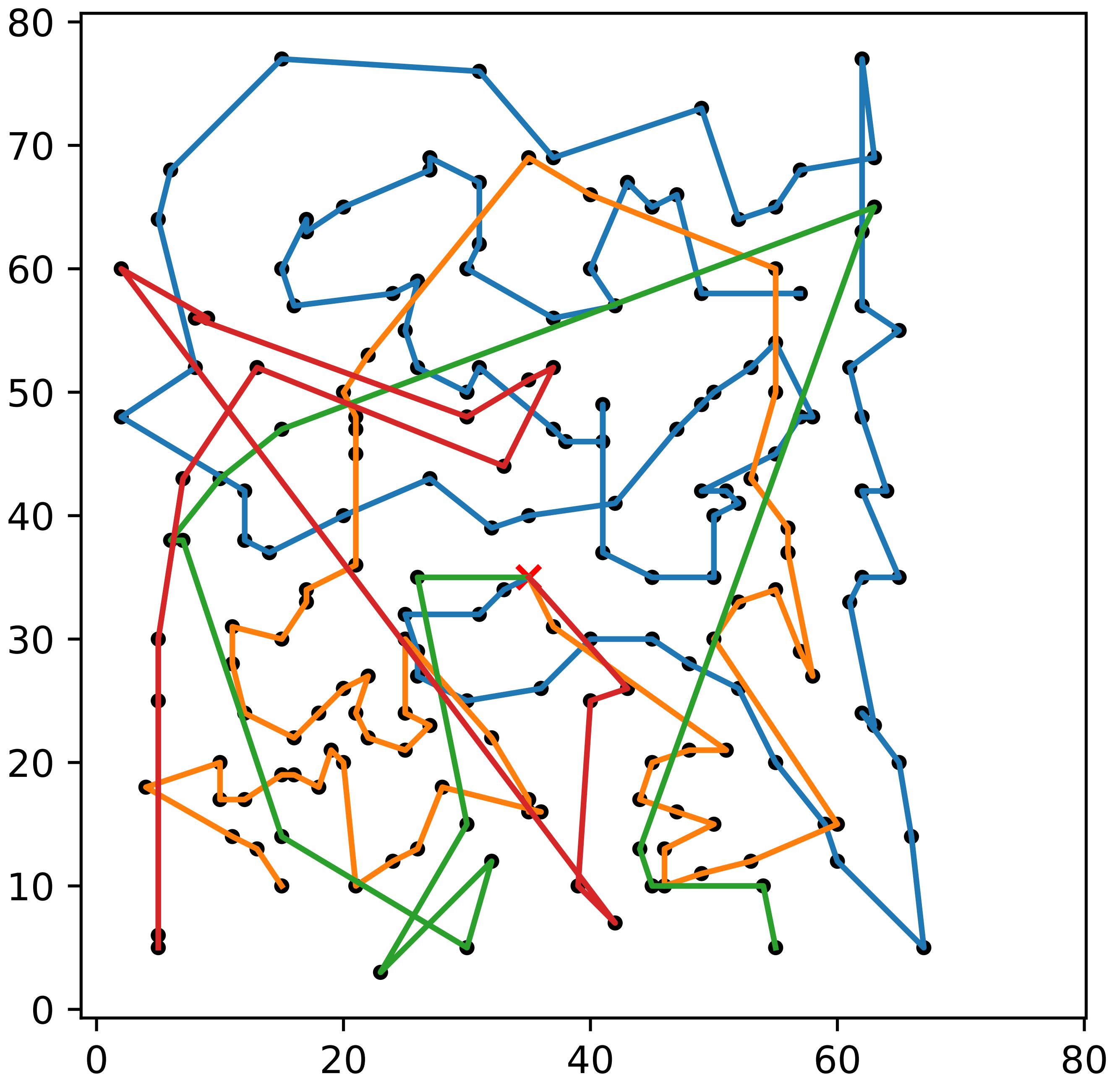
(f) M-n200-k16
$D = 50$



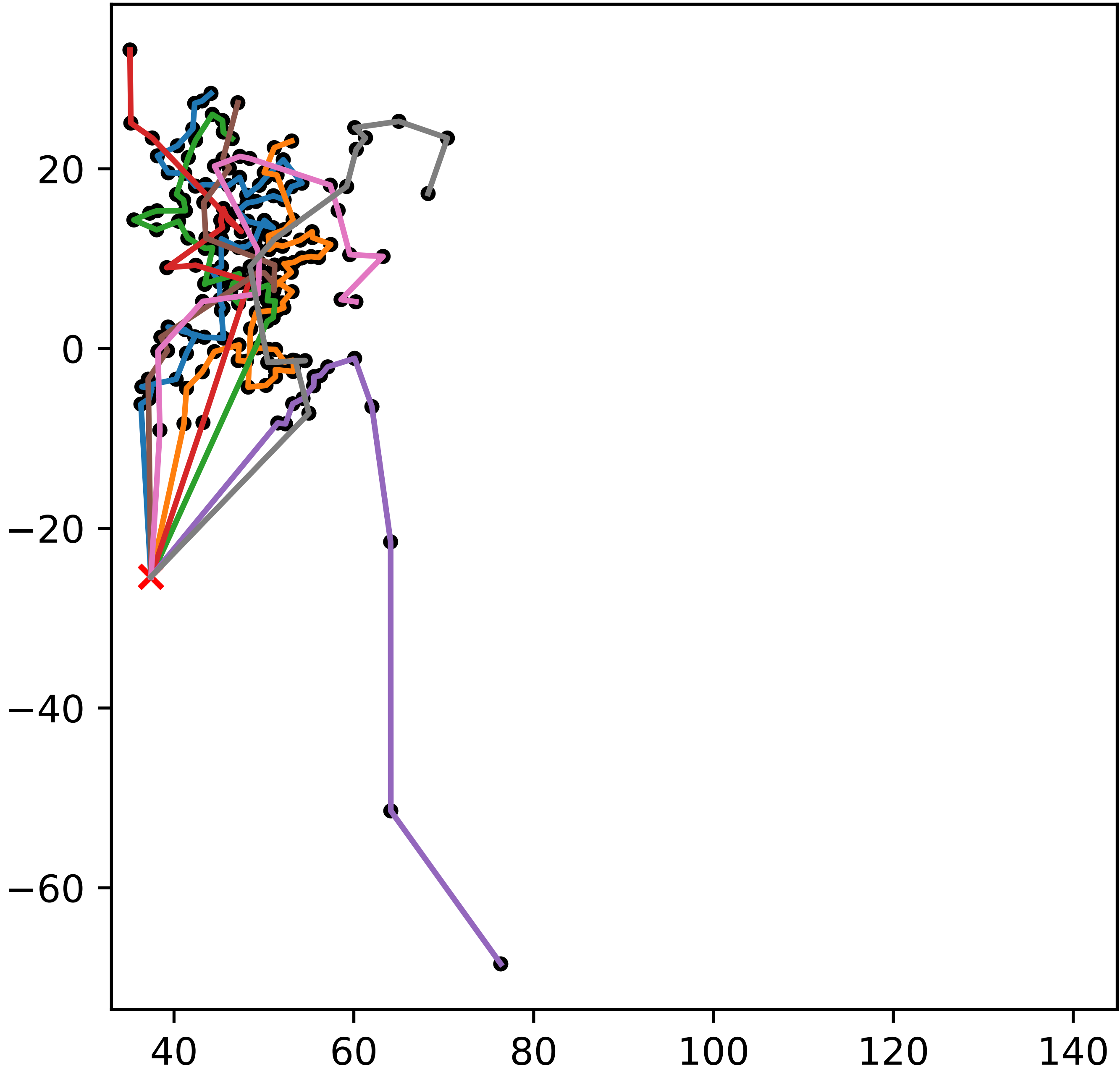
(g) M-n200-k16
$D = 100$



(h) M-n200-k16
$D = 200$



(i) M-n200-k16
$D = 500$



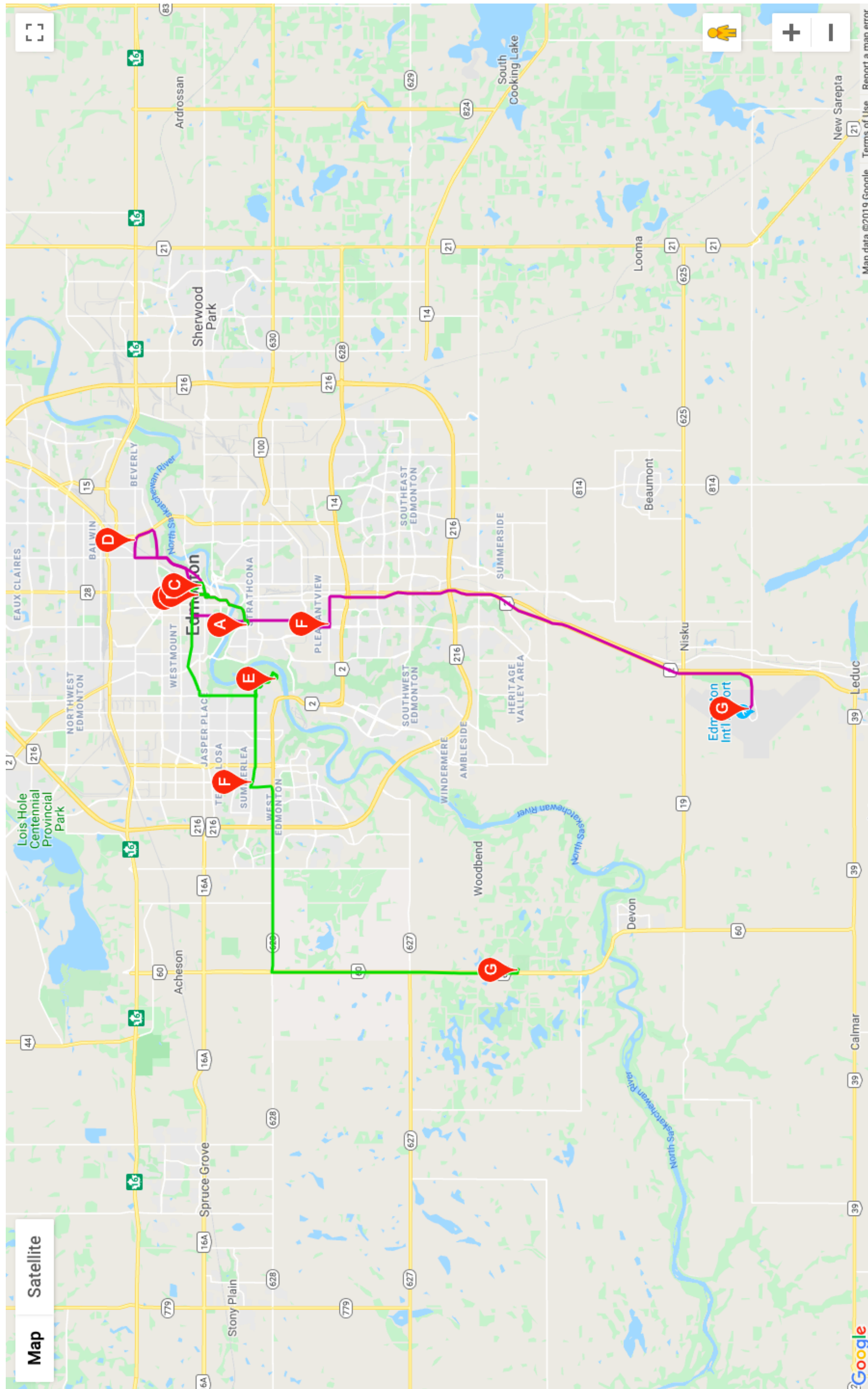
(j) gr202

Figure A.3: Visualizations of paths generated by Integer Program Solver. The depot is marked by a red x

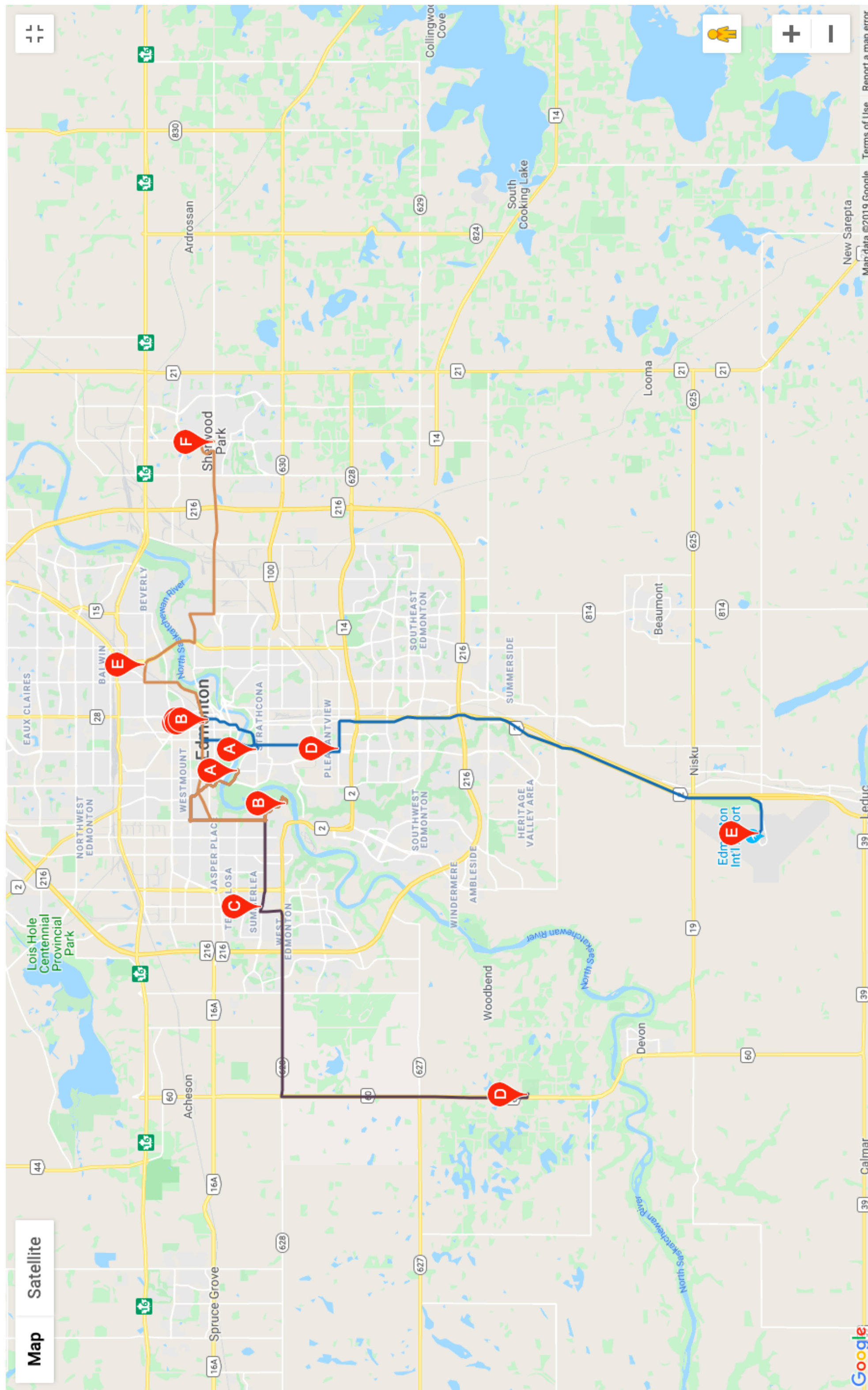
Appendix B

Google Maps Visualizations

The Google Maps distance matrix was not symmetric so our guarantees do not hold, but it was worth including in here. In order to make the matrix symmetric and also make the fewest number of API calls, we only queried the elements below the diagonal ones and mirrored them onto the other side. We also set the diagonal elements to 0. We then ran the Floyd-Warshall algorithm on the new distance matrix to make sure the instance has metric properties described in definition 3.



(a)



(b)

Figure B.1: Visualization of paths generated by the algorithm on real Google Maps data.