

Mathematical Programming and Operations Research

**Modeling, Algorithms, and Complexity
Examples in Excel and Python
(Work in progress)**

Edited by: Robert Hildebrand

Contributors: Robert Hildebrand, Laurent Poirier, Douglas Bish, Diego Moran

Version Compilation date: October 2, 2023

Preface

This entire book is a working manuscript. The first draft of the book is yet to be completed.

This book is being written and compiled using a number of open source materials. We will strive to properly cite all resources used and give references on where to find these resources. Although the material used in this book comes from a variety of licences, everything used here will be CC-BY-SA 4.0 compatible, and hence, the entire book will fall under a CC-BY-SA 4.0 license.

MAJOR ACKNOWLEDGEMENTS

I would like to acknowledge that substantial parts of this book were borrowed under a CC-BY-SA license. These substantial pieces include:

- "A First Course in Linear Algebra" by Lyryx Learning (based on original text by Ken Kuttler). A majority of their formatting was used along with selected sections that make up the appendix sections on linear algebra. We are extremely grateful to Lyryx for sharing their files with us. They do an amazing job compiling their books and the templates and formatting that we have borrowed here clearly took a lot of work to set up. Thank you for sharing all of this material to make structuring and formating this book much easier! See subsequent page for list of contributors.
- "Foundations of Applied Mathematics" with many contributors. See <https://github.com/Foundations-of-Applied-Mathematics>. Several sections from these notes were used along with some formatting. Some of this content has been edited or rearranged to suit the needs of this book. This content comes with some great references to code and nice formatting to present code within the book. See subsequent page with list of contributors.
- "Linear Inequalities and Linear Programming" by Kevin Cheung. See <https://github.com/dataopt/lineqlpbook>. These notes are posted on GitHub in a ".Rmd" format for nice reading online. This content was converted to L^AT_EX using Pandoc. These notes make up a substantial section of the Linear Programming part of this book.
- Linear Programming notes by Douglas Bish. These notes also make up a substantial section of the Linear Programming part of this book.

I would also like to acknowledge Laurent Porrier and Diego Moran for contributing various notes on linear and integer programming.

I would also like to thank Jamie Fravel for helping to edit this book and for contributing chapters, examples, and code.

Contents

Contents	5
1 Resources and Notation	5
2 Mathematical Programming	9
2.1 Linear Programming (LP)	10
2.2 Mixed-Integer Linear Programming (MILP)	11
2.3 Non-Linear Programming (NLP)	13
2.3.1 Convex Programming	13
2.3.2 Non-Convex Non-linear Programming	14
2.3.3 Machine Learning	14
2.4 Mixed-Integer Non-Linear Programming (MINLP)	15
2.4.1 Convex Mixed-Integer Non-Linear Programming	15
2.4.2 Non-Convex Mixed-Integer Non-Linear Programming	15
I Linear Programming	17
2.4.3 Fractional Knapsack	19
II Integer Programming	21
3 Integer Programming Formulations	23
3.1 Knapsack Problem	23
3.2 Capital Budgeting	26
3.3 Set Covering	29
3.3.1 Covering (Generalizing Set Cover)	33
3.4 Assignment Problem	33
3.5 Facility Location	35
3.5.1 Capacitated Facility Location	36
3.5.2 Uncapacitated Facility Location	38
3.6 Graph Coloring	40
3.7 Basic Modeling Tricks - Using Binary Variables	42
3.7.1 Big M constraints - Activating/Deactivating Inequalities	44
3.7.2 Either Or Constraints	45
3.7.3 If then implications - opposite direction	45
3.7.4 Multi Term Disjunction with application to 2D packing	48
3.7.4.1 Strip Packing Problem	48

6 ■ CONTENTS

3.7.5	SOS1 Constraints	51
3.7.6	SOS2 Constraints	51
3.7.7	Piecewise linear functions with SOS2 constraint	52
3.7.7.1	SOS2 with binary variables	54
3.7.8	Maximizing a minimum	54
3.7.9	Relaxing (nonlinear) equality constraints	55
3.7.10	Exact absolute value	55
3.7.10.1	Exact 1 -norm	55
3.7.10.2	Maximum	56
3.8	Network Flow	57
3.8.1	Maximum flow	57
3.8.2	Minimum Cost Network Flow	58
3.8.3	Multi-Commodity Minimum Cost Network Flow with Integrality Constraints	59
3.9	Job Shop Scheduling	62
3.9.1	JSSP Components	62
3.9.2	Mathematical Model	64
3.9.3	Job Shop Scheduling Variations	66
3.10	Quadratic Assignment Problem (QAP)	67
3.11	Generalized Assignment Problem (GAP)	69
3.11.1	In special cases	69
3.11.2	Explanation of definition	69
3.12	Other material	70
3.12.1	Binary reformulation of integer variables	70
3.13	Literature and Resources	72
3.14	MIP Solvers and Modeling Tools	73
4	Algorithms to Solve Integer Programs	75
4.1	Foundational Principle - LP is a relaxation for IP	76
4.1.1	Rounding LP Solution can be bad!	77
4.1.2	Rounding LP solution can be infeasible!	78
4.2	Branch and Bound	78
4.2.1	Binary Integer Programming	78
4.2.2	Branch and bound on general integer variables	80
4.3	Cutting Planes	83
4.3.1	Cover Inequalities	83
4.3.2	Chvátal Cuts	84
4.3.3	Cutting Planes Procedure	85
4.3.4	Gomory Cuts	88
4.4	Interpreting Output Information and Progress	90
4.5	Branching Decision Rules in Integer Programming	91
4.6	Decomposition Methods - Advanced approaches to integer programming	93
4.6.1	Lagrangian Relaxation	93
4.6.2	Dantzig-Wolfe Reformulation and Column Generation	94
4.6.3	Benders Decomposition	95

4.7	Literature and Resources	96
5	Exponential Size Formulations	97
5.1	Cutting Stock	97
5.1.1	Pattern formulation	100
5.1.2	Column Generation	102
5.1.3	Cutting Stock - Multiple widths	103
5.2	Traveling Salesman Problem	104
5.2.1	Miller Tucker Zemlin (MTZ) Model	106
5.2.2	Dantzig-Fulkerson-Johnson (DFJ) Model	113
5.2.3	Traveling Salesman Problem - Branching Solution	116
5.2.4	Traveling Salesman Problem Variants	116
5.2.4.1	Many salespersons (m-TSP)	116
5.2.4.2	TSP with order variants	118
5.2.5	Multi vehicle model with capacities	118
5.3	Vehicle Routing Problem (VRP)	119
5.4	The Clarke-Wright Saving Algorithm for VRP	119
5.4.1	Concept	120
5.4.2	Saving Calculation	120
5.4.3	Algorithm Steps	120
5.4.4	Advantages and Limitations	120
5.5	Literature and other notes	121
5.5.1	TSP In Excel	121
5.5.2	Resources	121
6	Algorithms and Complexity	123
6.1	Big-O Notation	123
6.2	Algorithms - Example with Bubble Sort	127
6.2.1	Sorting	127
6.3	Problem, instance, size	132
6.3.1	Problem, instance	132
6.3.2	Format and examples of problems/instances	132
6.3.3	Size of an instance	132
6.4	Complexity Classes	133
6.4.1	P	134
6.4.2	NP	135
6.4.3	Problem Reductions	136
6.4.4	Significance of Reductions	138
6.4.5	Examples of Problem Reductions	138
6.4.5.1	VERTEX COVER to SET COVER	138
6.4.5.2	3SAT to 3D MATCHING	138
6.4.5.3	Discussion	139
6.4.6	NP-Hard	139
6.4.7	NP-Complete	139

8 ■ CONTENTS

6.5	Problems and Algorithms	141
6.5.1	Matching Problem	141
6.5.1.1	Greedy Algorithm for Maximal Matching	142
6.5.1.2	Other algorithms to look at	143
6.5.2	Minimum Spanning Tree	143
6.5.3	Kruskal's algorithm	144
6.5.3.1	Prim's Algorithm	144
6.5.4	Traveling Salesman Problem	144
6.5.4.1	Nearest Neighbor - Construction Heuristic	144
6.5.4.2	Double Spanning Tree - 2-Apx	145
6.5.4.3	Christofides - Approximation Algorithm - (3/2)-Apx	146
6.6	Resources	146
6.6.1	Advanced - NP Completeness Reductions	147
6.7	Other material for Integer Linear Programming	148
	Exercises	152
	Solutions	152
6.7.1	Other discrete problems	153
6.7.2	Assignment Problem and the Hungarian Algorithm	153
6.7.3	History of Computation in Combinatorial Optimization	153
7	Heuristics for TSP	155
7.1	Construction Heuristics	155
7.1.1	Random Solution	155
7.1.2	Nearest Neighbor	155
7.1.3	Insertion Method	156
7.2	Improvement Heuristics	156
7.2.1	2-Opt (Subtour Reversal)	156
7.2.2	3-Opt	158
7.2.3	k -Opt	158
7.3	Meta-Heuristics	158
7.3.1	Hill Climbing (2-Opt for TSP)	158
7.3.2	Simulated Annealing	160
7.3.3	Tabu Search	161
7.3.4	Genetic Algorithms	162
7.3.5	Greedy randomized adaptive search procedure (GRASP)	162
7.3.6	Ant Colony Optimization	162
7.4	Computational Comparisons	162
7.4.1	VRP - Clark Wright Algorithm	163
8	Forecasting and Stochastic Programming	165
9	Decomposition Methods	167
A	Linear Algebra	169

A.1 Contributors	169
A.1.1 Graph Theory	3

Introduction

Letter to instructors

This is an introductory book for students to learn optimization theory, tools, and applications. The two main goals are (1) students are able to apply the of optimization in their future work or research (2) students understand the concepts underlying optimization solver and use this knowledge to use solvers effectively.

This book was based on a sequence of course at Virginia Tech in the Industrial and Systems Engineering Department. The courses are *Deterministic Operations Research I* and *Deterministic Operations Reserach II*. The first course focuses on linear programming, while the second course covers integer programming an nonlinear programming. As such, the content in this book is meant to cover 2 or more full courses in optimization.

The book is designed to be read in a linear fashion. That said, many of the chapters are mostly independent and could be rearranged, removed, or shortened as desited. This is an open source textbook, so use it as you like. You are encouraged to share adaptations and improvements so that this work can evolve over time.

Letter to students

This book is designed to be a resource for students interested in learning what optimizaiton is, how it works, and how you can apply it in your future career. The main focus is being able to apply the techniques of optimization to problems using computer technology while understanding (at least at a high level) what the computer is doing and what you can claim about the output from the computer.

Quite importantly, keep in mind that when someone claims to have *optimized* a problem, we want to know what kind of gaurantees they have about how good their solution is. Far too often, the solution that is provided is suboptimal by 10%, 20%, or even more. This can mean spending excess amounts of money, time, or energy that could have been saved. And when problems are at a large scale, this can easily result in millions of dollars in savings.

For this reason, we will learn the perspective of *mathematical programming* (a.k.a. mathematical optimization). The key to this study is that we provide gaurantees on how good a solution is to a given problem. We will also study how difficult a problem is to solve. This will help us know (a) how long it might take to solve it and (b) how good a of a solution we might expect to be able to find in reasonable amount of time.

2 ■ CONTENTS

We will later study *heuristic* methods - these methods typically do not come with guarantees, but tend to help find quality solutions.

Note: Although there is some computer programming required in this work, this is not a course on programming. Thanks to the fantastic modelling packages available these days, we are able to solve complicated problems with little programming effort. The key skill we will need to learn is *mathematical modeling*: converting words and ideas into numbers and variables in order to communicate problems to a computer so that it can solve a problem for you.

As a main element of this book, we would like to make the process of using code and software as easy as possible. Attached to most examples in the book, there will be links to code that implements and solves the problem using several different tools from Excel and Python. These examples can should make it easy to solve a similar problem with different data, or more generally, can serve as a basis for solving related problems with similar structure.

How to use this book

Skim ahead. We recommend that before you come across a topic in lecture, that you skim the relevant sections ahead of time to get a broad overview of what is to come. This may take only a fraction of the time that it may take for you to read it.

Read the expected outcomes. At the beginning of each section, there will be a list of expected outcomes from that section. Review these outcomes before reading the section to help guide you through what is most relevant for you to take away from the material. This will also provide a brief look into what is to follow in that section.

Read the text. Read carefully the text to understand the problems and techniques. We will try to provide a plethora of examples, therefore, depending on your understanding of a topic, you many need to go carefully over all of the examples.

Explore the resources. Lastly, we recognize that there are many alternative methods of learning given the massive amounts of information and resources online. Thus, at the end of each section, there will be a number of superb resources that available on the internet in different formats. There are other free textbooks, informational websites, and also number of fantastic videos posted to youtube. We encourage to explore the resources to get another perspective on the material or to hear/read it taught from a different point of view or in presentation style.

Outline of this book

This book is divided in to 4 Parts:

Part I Linear Programming,

Part II Integer Programming,

Part III Discrete Algorithms,

Part IV Nonlinear Programming.

There are also a number of chapters of background material in the Appendix.

The content of this book is designed to encompass 2-3 full semester courses in an industrial engineering department.

Work in progress

This book is still a work in progress, so please feel free to send feedback, questions, comments, and edits to Robert Hildebrand at open.optimization@gmail.com.

1. Resources and Notation

Here are a list of resources that may be useful as alternative references or additional references.

FREE NOTES AND TEXTBOOKS

- Mathematical Programming with Julia by Richard Lusby & Thomas Stidsen
- Linear Programming by K.J. Mtetwa, David
- A first course in optimization by Jon Lee
- Introduction to Optimization Notes by Komei Fukuda
- Convex Optimization by Boyd and Vandenberghe
- LP notes of Michel Goemans from MIT
- Understanding and Using Linear Programming - Matousek and Gärtner [Downloadable from Springer with University account]
- Operations Research Problems Statements and Solutions - Raúl Poler Josefa Mula Manuel Díaz-Madroñero [Downloadable from Springer with University account]

NOTES, BOOKS, AND VIDEOS BY VARIOUS SOLVER GROUPS

- AIMMS Optimization Modeling
- Optimization Modeling with LINGO by Linus Schrage
- The AMPL Book
- Microsoft Excel 2019 Data Analysis and Business Modeling, Sixth Edition, by Wayne Winston - Available to read for free as an e-book through Virginia Tech library at Orieilly.com.
- Lesson files for the Winston Book
- Video instructions for solver and an example workbook
- youtube-OR-course

6 ■ Resources and Notation

GUROBI LINKS

- Go to <https://github.com/Gurobi> and download the example files.
- Essential ingredients
- Gurobi Linear Programming tutorial
- Gurobi tutorial MILP
- GUROBI - Python 1 - Modeling with GUROBI in Python
- GUROBI - Python II: Advanced Algebraic Modeling with Python and Gurobi
- GUROBI - Python III: Optimization and Heuristics
- Webinar Materials
- GUROBI Tutorials

HOW TO PROVE THINGS

- Hammack - Book of Proof

STATISTICS

- Open Stax - Introductory Statistics

LINEAR ALGEBRA

- Beezer - A first course in linear algebra
- Selinger - Linear Algebra
- Cherney, Denton, Thomas, Waldron - Linear Algebra

REAL ANALYSIS

- Mathematical Analysis I by Elias Zakon

DISCRETE MATHEMATICS, GRAPHS, ALGORITHMS, AND COMBINATORICS

- Levin - Discrete Mathematics - An Open Introduction, 3rd edition
- Github - Discrete Mathematics: an Open Introduction CC BY SA
- Keller, Trotter - Applied Combinatorics (CC-BY-SA 4.0)
- Keller - Github - Applied Combinatorics

PROGRAMMING WITH PYTHON

- A Byte of Python
- Github - Byte of Python (CC-BY-SA)

Also, go to <https://github.com/open-optimization/open-optimization-or-examples> to look at more examples.

Notation

- $\mathbf{1}$ - a vector of all ones (the size of the vector depends on context)
- \forall - for all
- \exists - there exists
- \in - in
- \therefore - therefore
- \Rightarrow - implies
- s.t. - such that (or sometimes "subject to".... from context?)
- $\{0, 1\}$ - the set of numbers 0 and 1
- \mathbb{Z} - the set of integers (e.g. $1, 2, 3, -1, -2, -3, \dots$)
- \mathbb{Q} - the set of rational numbers (numbers that can be written as p/q for $p, q \in \mathbb{Z}$ (e.g. $1, 1/6, 27/2$)
- \mathbb{R} - the set of all real numbers (e.g. $1, 1.5, \pi, e, -11/5$)
- \setminus - setminus, (e.g. $\{0, 1, 2, 3\} \setminus \{0, 3\} = \{1, 2\}$)
- \cup - union (e.g. $\{1, 2\} \cup \{3, 5\} = \{1, 2, 3, 5\}$)
- \cap - intersection (e.g. $\{1, 2, 3, 4\} \cap \{3, 4, 5, 6\} = \{3, 4\}$)
- $\{0, 1\}^4$ - the set of 4 dimensional vectors taking values 0 or 1, (e.g. $[0, 0, 1, 0]$ or $[1, 1, 1, 1]$)
- \mathbb{Z}^4 - the set of 4 dimensional vectors taking integer values (e.g., $[1, -5, 17, 3]$ or $[6, 2, -3, -11]$)

8 ■ Resources and Notation

- \mathbb{Q}^4 - the set of 4 dimensional vectors taking rational values (e.g. $[1.5, 3.4, -2.4, 2]$)
- \mathbb{R}^4 - the set of 4 dimensional vectors taking real values (e.g. $[3, \pi, -e, \sqrt{2}]$)
- $\sum_{i=1}^4 i = 1 + 2 + 3 + 4$
- $\sum_{i=1}^4 i^2 = 1^2 + 2^2 + 3^2 + 4^2$
- $\sum_{i=1}^4 x_i = x_1 + x_2 + x_3 + x_4$
- \square - this is a typical Q.E.D. symbol that you put at the end of a proof meaning "I proved it."
- For $x, y \in \mathbb{R}^3$, the following are equivalent (note, in other contexts, these notations can mean different things)
 - $x^\top y$ *matrix multiplication*
 - $x \cdot y$ *dot product*
 - $\langle x, y \rangle$ *inner product*

and evaluate to $\sum_{i=1}^3 x_i y_i = x_1 y_1 + x_2 y_2 + x_3 y_3$.

A sample sentence:

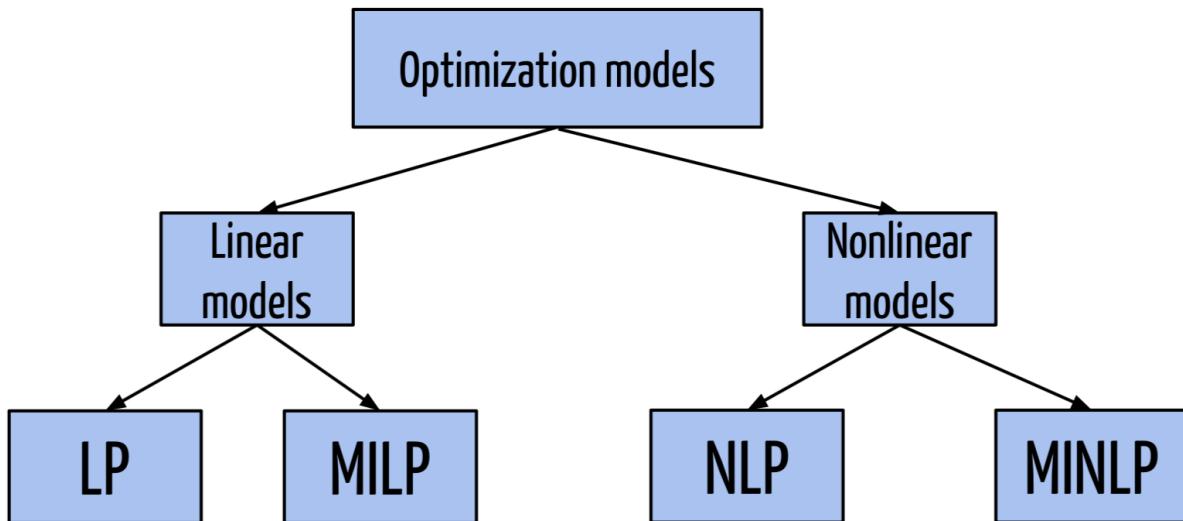
$$\forall x \in \mathbb{Q}^n \exists y \in \mathbb{Z}^n \setminus \{0\}^n s.t. x^\top y \in \{0, 1\}$$

"For all non-zero rational vectors x in n -dimensions, there exists a non-zero n -dimensional integer vector y such that the dot product of x with y evaluates to either 0 or 1."

2. Mathematical Programming

Outcomes

We will state main general problem classes to be associated with in these notes. These are Linear Programming (LP), Mixed-Integer Linear Programming (MILP), Non-Linear Programming (NLP), and Mixed-Integer Non-Linear Programming (MINLP).



© problem-class-diagram¹
Figure 2.1: problem-class-diagram

Along with each problem class, we will associate a complexity class for the general version of the problem. See chapter 6 for a discussion of complexity classes. Although we will often state that input data for a problem comes from \mathbb{R} , when we discuss complexity of such a problem, we actually mean that the data is rational, i.e., from \mathbb{Q} , and is given in binary encoding.

¹problem-class-diagram, from [problem-class-diagram](#). [problem-class-diagram](#), [problem-class-diagram](#).

2.1 Linear Programming (LP)

Some linear programming background, theory, and examples will be provided in ??.

Linear Programming (LP):

Polynomial time (P)

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *linear programming* problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{2.1}$$

Linear programming can come in several forms, whether we are maximizing or minimizing, or if the constraints are \leq , $=$ or \geq . One form commonly used is *Standard Form* given as

Linear Programming (LP) Standard Form:

Polynomial time (P)

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *linear programming* problem in *standard form* is

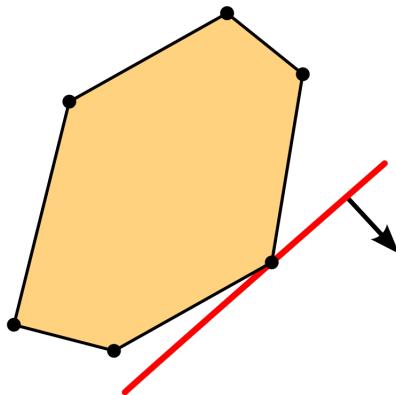
$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{2.2}$$

Figure 2.2

Exercise 2.1:

Start with a problem in form given as (2.1) and convert it to standard form (2.2) by adding at most m many new variables and by enlarging the constraint matrix A by at most m new columns.

²wiki/File/linear-programming.png, from wiki/File/linear-programming.png. wiki/File/linear-programming.png, wiki/File/linear-programming.png.



© wiki/File/linear-programming.png²

Figure 2.2: Linear programming constraints and objective.

2.2 Mixed-Integer Linear Programming (MILP)

Mixed-integer linear programming will be the focus of Sections 3.5, 4, and ???. Recall that the notation \mathbb{Z} means the set of integers and the set \mathbb{R} means the set of real numbers. The first problem of interest here is a *binary integer program* (BIP) where all n variables are binary (either 0 or 1).

Binary Integer programming (BIP):

NP-Complete

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *binary integer programming* problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \{0, 1\}^n \end{aligned} \tag{2.1}$$

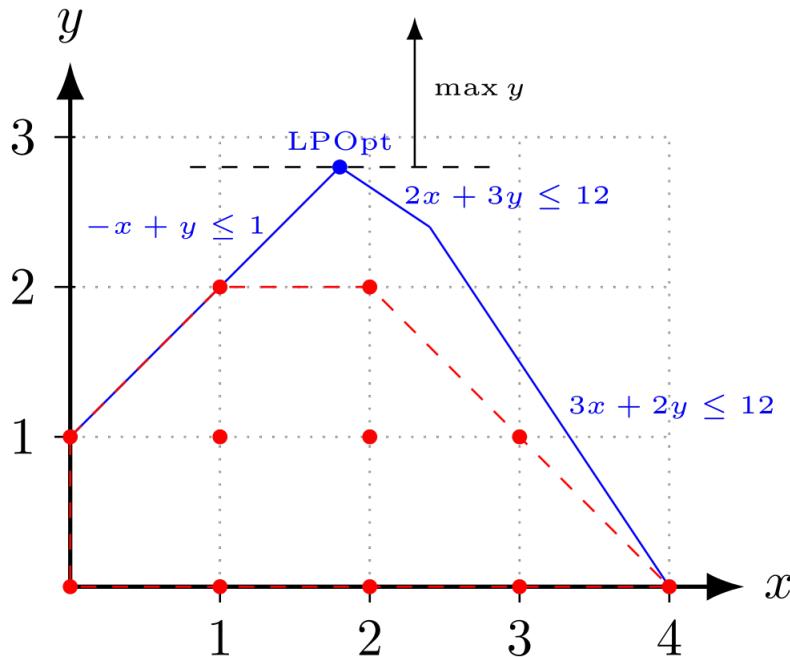
A slightly more general class is the class of *Integer Linear Programs* (ILP). Often this is referred to as *Integer Program* (IP), although this term could leave open the possibility of non-linear parts.

Figure 2.3

Integer Linear Programming (ILP):

NP-Complete

²wiki/File/integer-programming.png, from wiki/File/integer-programming.png. wiki/File/integer-programming.png, wiki/File/integer-programming.png.



© wiki/File/integer-programming.png³

Figure 2.3: Comparing the LP relaxation to the IP solutions.

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *integer linear programming* problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{Z}^n \end{aligned} \tag{2.2}$$

An even more general class is *Mixed-Integer Linear Programming (MILP)*. This is where we have n integer variables $x_1, \dots, x_n \in \mathbb{Z}$ and d continuous variables $x_{n+1}, \dots, x_{n+d} \in \mathbb{R}$. Succinctly, we can write this as $x \in \mathbb{Z}^n \times \mathbb{R}^d$, where \times stands for the *cross-product* between two spaces.

Below, the matrix A now has $n+d$ columns, that is, $A \in \mathbb{R}^{m \times n+d}$. Also note that we have not explicitly enforced non-negativity on the variables. If there are non-negativity restrictions, this can be assumed to be a part of the inequality description $Ax \leq b$.

Mixed-Integer Linear Programming (MILP):

NP-Complete

Given a matrix $A \in \mathbb{R}^{m \times (n+d)}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^{n+d}$, the *mixed-integer linear program-*

ming problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{Z}^n \times \mathbb{R}^d \end{aligned} \tag{2.3}$$

2.3 Non-Linear Programming (NLP)

NLP:

NP-Hard

Given a function $f(x): \mathbb{R}^d \rightarrow \mathbb{R}$ and other functions $f_i(x): \mathbb{R}^d \rightarrow \mathbb{R}$ for $i = 1, \dots, m$, the *nonlinear programming* problem is

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d \end{aligned} \tag{2.1}$$

Nonlinear programming can be separated into convex programming and non-convex programming. These two are very different beasts and it is important to distinguish between the two.

2.3.1. Convex Programming

Here the functions are all **convex**!

Convex Programming:

Polynomial time (P) (typically)

Given a convex function $f(x): \mathbb{R}^d \rightarrow \mathbb{R}$ and convex functions $f_i(x): \mathbb{R}^d \rightarrow \mathbb{R}$ for $i = 1, \dots, m$, the *convex programming* problem is

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d \end{aligned} \tag{2.2}$$

Observe that convex programming is a generalization of linear programming. This can be seen by letting $f(x) = c^\top x$ and $f_i(x) = A_i x - b_i$.

2.3.2. Non-Convex Non-linear Programming

When the function f or functions f_i are non-convex, this becomes a non-convex nonlinear programming problem. There are a few complexity issues with this.

IP AS NLP As seen above, quadratic constraints can be used to create a feasible region with discrete solutions. For example

$$x(1-x) = 0$$

has exactly two solutions: $x = 0, x = 1$. Thus, quadratic constraints can be used to model binary constraints.

Binary Integer programming (BIP) as a NLP:

NP-Hard

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *binary integer programming* problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \{0, 1\}^n \\ & x_i(1 - x_i) = 0 \quad \text{for } i = 1, \dots, n \end{aligned} \tag{2.3}$$

2.3.3. Machine Learning

Machine learning problems are often cast as continuous optimization problems, which involve adjusting parameters to minimize or maximize a particular objective. Frequently they are convex optimization problems, but many turn out to be nonconvex. Here are two examples of how these problems arise at a glance. We will see examples in greater detail later in the book.

Loss Function Minimization

In supervised learning, this objective is typically a loss function L that quantifies the discrepancy between the predictions of a model and the true data labels. The aim is to adjust the parameters θ of the model to minimize this loss. Mathematically, this can be represented as:

$$\min_{\theta} L(\theta) = \min_{\theta} \frac{1}{N} \sum_{i=1}^N l(y_i, f(x_i; \theta)) \quad (2.4)$$

where N is the number of data points, l is a per-data-point loss (e.g., squared error for regression or cross-entropy for classification), y_i is the true label for the i -th data point, and $f(x_i; \theta)$ is the model's prediction for the i -th data point with parameters θ .

Clustering Formulation

Clustering, on the other hand, seeks to group or partition data points such that data points in the same group are more similar to each other than those in other groups. One popular method is the k-means clustering algorithm. The objective of k-means is to partition the data into k clusters by minimizing the within-cluster sum of squares (WCSS). The mathematical formulation can be given as:

$$\min_{\mathbf{c}_1, \dots, \mathbf{c}_k} \sum_{j=1}^k \sum_{x_i \in C_j} \|x_i - \mathbf{c}_j\|^2 \quad (2.5)$$

where C_j represents the j -th cluster and \mathbf{c}_j is the centroid of that cluster.

This encapsulation presents a glimpse into how ML problems are framed mathematically. In practice, numerous algorithms, constraints, and regularizations add complexity to these basic formulations.

2.4 Mixed-Integer Non-Linear Programming (MINLP)

2.4.1. Convex Mixed-Integer Non-Linear Programming

2.4.2. Non-Convex Mixed-Integer Non-Linear Programming

Part I

Linear Programming

2.4.3. Fractional Knapsack

We will quickly state a result about the Fractional Knapsack problem (a.k.a. Continuous Knapsack Problem). This will help provide a nice easy problem for our first branch and bound example. The generalization of this problem is called the continuous knapsack problem: given $\mathbf{c}, \mathbf{a} \in \mathbb{R}_+^n$ and $b \in \mathbb{R}_+$, determine

$$\underset{x}{\text{Maximize}} \quad \mathbf{c}^\top \mathbf{x} \quad (2.1a)$$

$$\text{s.t.} \quad \mathbf{a}^\top \mathbf{x} \leq b \quad (2.1b)$$

$$\mathbf{x} \in [0, 1]^n \quad (2.1c)$$

We will assume that the solution $x_i = 1$ for all $i = 1, \dots, n$ is not feasible, i.e., that $\sum_{i=1}^n a_i > b$. This ensures that the inequality $\mathbf{a}^\top \mathbf{x} \leq b$ is not redundant.

The fractional knapsack problem has an exact greedy algorithm. In particular, if we order the variables according to their value versus their weight, then we can choose to add items according to this ordering. This is summarized in the following theorem.

Theorem 2.2:

Assume that the problem is indexed in such a way that $\frac{c_i}{a_i} \geq \frac{c_{i+1}}{a_{i+1}}$. Find the smallest integer k such that $\sum_{i=1}^k a_i \geq b$ (in this cases $k = \lfloor b \rfloor$). The optimal solution \mathbf{x}^* is given by

$$x_i^* = \begin{cases} 1, & \text{if } i < k \\ \frac{\bar{b}}{a_k} & \text{if } i = k \\ 0 & \text{if } i > k \end{cases}$$

where $\bar{b} = b - \sum_{i < k} a_i$. or written differently,

$$\mathbf{x}^* = (1, \dots, 1, \frac{\bar{b}}{a_k}, 0, \dots, 0), \quad (2.2)$$

where $\bar{b} = b - \sum_{i=1}^{k-1} a_i$ is the remaining capacity in constraint (2.1b) after the first $k-1$ variables are assigned to their maximum value.

Part II

Integer Programming

3. Integer Programming Formulations

Outcomes

- A. Learn classic integer programming formulations.
- B. Demonstrate different uses of binary and integer variables.
- C. Demonstrate the format for modeling an optimization problem with sets, parameters, variables, and the model.

In this section, we will describe classical integer programming formulations. These formulations may reflect a real world problem exactly, or may be part of the setup of a real world problem.

3.1 Knapsack Problem

The *knapsack problem* can take different forms depending on if the variables are binary or integer. The binary version means that there is only one item of each item type that can be taken. This is typically illustrated as a backpack (knapsack) and some items to put into it (see Figure 3.1), but has applications in many contexts.

Binary Knapsack Problem:

NP-Complete

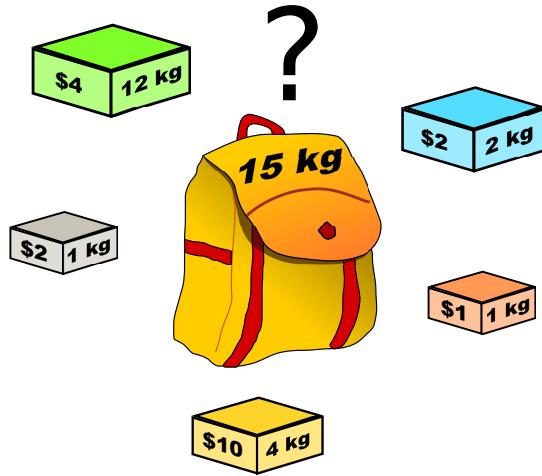
Given a non-negative weight vector $a \in \mathbb{Q}_+^n$, a capacity $b \in \mathbb{Q}_+$, and objective coefficients $c \in \mathbb{Q}^n$,

$$\begin{aligned} & \max c^\top x \\ & \text{s.t. } a^\top x \leq b \\ & \quad x \in \{0, 1\}^n \end{aligned} \tag{3.1}$$

Example: Knapsack

Gurobipy

¹[wiki/File/knapsack](#), from [wiki/File/knapsack](#). [wiki/File/knapsack](#), [wiki/File/knapsack](#).



© wiki/File/knapsack¹

Figure 3.1: Knapsack Problem: which items should we choose take in the knapsack that maximizes the value while respecting the 15kg weight limit?

You have a knapsack (bag) that can only hold $W = 15$ kgs. There are 5 items that you could possibly put into your knapsack. The items (weight, value) are given as: (12 kg, \$4), (2 kg, \$2), (1kg, \$2), (1kg, \$1), (4kg, \$10). Which items should you take to maximize your value in the knapsack? See Figure 3.1.

Variables:

- let $x_i = 0$ if item i is in the bag
- let $x_i = 1$ if item i is not in the bag

Model:

$$\begin{aligned}
 & \text{max } 4x_1 + 2x_2 + 2x_3 + 1x_4 + 10x_5 && \text{(Total value)} \\
 & \text{s.t. } 12x_1 + 2x_2 + 1x_3 + 1x_4 + 4x_5 \leq 15 && \text{(Capacity bound)} \\
 & x_i \in \{0, 1\} \text{ for } i = 1, \dots, 5 && \text{(Item taken or not)}
 \end{aligned}$$

In the integer case, we typically require the variables to be non-negative integers, hence we use the notation $x \in \mathbb{Z}_+^n$. This setting reflects the fact that instead of single individual items, you have item types of which you can take as many of each type as you like that meets the constraint.

Integer Knapsack Problem:*NP-Complete*

Given a non-negative weight vector $a \in \mathbb{Q}_+^n$, a capacity $b \in \mathbb{Q}_+$, and objective coefficients $c \in \mathbb{Q}^n$,

$$\begin{aligned} & \max c^\top x \\ & \text{s.t. } a^\top x \leq b \\ & \quad x \in \mathbb{Z}_+^n \end{aligned} \tag{3.2}$$

We can also consider an equality constrained version

Equality Constrained Integer Knapsack Problem:*NP-Hard*

Given a non-negative weight vector $a \in \mathbb{Q}_+^n$, a capacity $b \in \mathbb{Q}_+$, and objective coefficients $c \in \mathbb{Q}^n$,

$$\max c^\top x \tag{3.3}$$

$$\text{s.t. } a^\top x = b \tag{3.4}$$

$$x \in \mathbb{Z}_+^n \tag{3.5}$$

Example: Min Coins

Gurobipy

Using pennies, nickels, dimes, and quarters, how can you minimize the number of coins you need to to make up a sum of 83¢?

Variables:

- Let p be the number of pennies used
- Let n be the number of nickels used
- Let d be the number of dimes used
- Let q be the number of quarters used

Model

$$\begin{array}{ll}
 \min & p + n + d + q && \text{total number of coins used} \\
 \text{s.t.} & p + 5n + 10d + 25q = 83 && \text{sums to } 83\text{¢} \\
 & p, d, n, q \in \mathbb{Z}_+ && \text{each is a non-negative integer}
 \end{array}$$

3.2 Capital Budgeting

The *capital budgeting* problem is a nice generalization of the knapsack problem. This problem has the same structure as the knapsack problem, except now it has multiple constraints. We will first describe the problem, give a general model, and then look at an explicit example.

Capital Budgeting:

A firm has n projects it could undertake to maximize revenue, but budget limitations require that not all can be completed.

- Project j expects to produce revenue c_j dollars overall.
- Project j requires investment of a_{ij} dollars in time period i for $i = 1, \dots, m$.
- The capital available to spend in time period i is b_i .

Which projects should the firm invest in to maximize its expected return while satisfying its weekly budget constraints?

We will first provide a general formulation for this problem.

Capital Budgeting Model:

Sets:

- Let $I = \{1, \dots, m\}$ be the set of time periods.
- Let $J = \{1, \dots, n\}$ be the set of possible investments.

Parameters:

- c_j is the expected revenue of investment j for $j \in J$
- b_i is the available capital in time period i for i in I
- a_{ij} is the resources required for investment j in time period i , for i in I , for j in J .

Variables:

- let $x_i = 0$ if investment i is chosen
- let $x_i = 1$ if investment i is not chosen

Model:

$$\begin{aligned} \max \quad & \sum_{j=1}^n c_j x_j && \text{(Total Expected Revenue)} \\ \text{s.t.} \quad & \sum_{j=1}^n a_{ij} x_j \leq b_i, \quad i = 1, \dots, m && \text{(Resource constraint week } i) \\ & x_j \in \{0, 1\}, \quad j = 1, \dots, n \end{aligned}$$

Consider the example given in the following table.

Project	$\mathbb{E}[\text{Revenue}]$	Resources required in week 1	Resources required in week 2
1	10	3	4
2	8	1	2
3	6	2	1
Resources available		5	6

Given this data, we can setup our problem explicitly as follows

Example: Capital Budgeting

Gurobipy

Sets:

- Let $I = \{1, 2\}$ be the set of time periods.
- Let $J = \{1, 2, 3\}$ be the set of possible investments.

Parameters:

- c_j is given in column " $\mathbb{E}[\text{Revenue}]$ ".
- b_i is given in row "Resources available".
- a_{ij} given in row j , and column for week i .

Variables:

- let $x_i = 0$ if investment i is chosen
- let $x_i = 1$ if investment i is not chosen

The explicit model is given by

Model:

$$\begin{aligned}
 & \max \quad 10x_1 + 8x_2 + 6x_3 && \text{(Total Expected Revenue)} \\
 & s.t. \quad 3x_1 + 1x_2 + 2x_3 \leq 5 && \text{(Resource constraint week 1)} \\
 & \quad \quad \quad 4x_1 + 2x_2 + 1x_3 \leq 6 && \text{(Resource constraint week 2)} \\
 & \quad \quad \quad x_j \in \{0, 1\}, \quad j = 1, 2, 3
 \end{aligned}$$

3.3 Set Covering

The *set covering* problem can be used for a wide array of problems. We will see several examples in this section.

Set Covering:

NP-Complete

Given a set V with subsets V_1, \dots, V_l , determine the smallest subset $S \subseteq V$ such that $S \cap V_i \neq \emptyset$ for all $i = 1, \dots, l$.

The set cover problem can be modeled as

$$\begin{aligned} \min \quad & 1^\top x \\ \text{s.t.} \quad & \sum_{v \in V_i} x_v \geq 1 \text{ for all } i = 1, \dots, l \\ & x_v \in \{0, 1\} \text{ for all } v \in V \end{aligned} \tag{3.1}$$

where x_v is a 0/1 variable that takes the value 1 if we include item j in set S and 0 if we do not include it in the set S .

Example: Capital Budgeting

Gurobipy

Sets:

- Let $I = \{1, 2\}$ be the set of time periods.
- Let $J = \{1, 2, 3\}$ be the set of possible investments.

Parameters:

- c_j is given in column "E[Revenue]."
- b_i is given in row "Resources available".
- a_{ij} given in row j , and column for week i .

Variables:

- let $x_i = 0$ if investment i is chosen
- let $x_i = 1$ if investment i is not chosen

The explicit model is given by

Model:

$$\begin{aligned}
 \max \quad & 10x_1 + 8x_2 + 6x_3 && \text{(Total Expected Revenue)} \\
 \text{s.t.} \quad & 3x_1 + 1x_2 + 2x_3 \leq 5 && \text{(Resource constraint week 1)} \\
 & 4x_1 + 2x_2 + 1x_3 \leq 6 && \text{(Resource constraint week 2)} \\
 & x_j \in \{0, 1\}, j = 1, 2, 3
 \end{aligned}$$

One specific type of set cover problem is the *vertex cover* problem.

Example: Vertex Cover:

NP-Complete

Given a graph $G = (V, E)$ of vertices and edges, we want to find a smallest size subset $S \subseteq V$ such that every for every $e = (v, u) \in E$, either u or v is in S .

We can write this as a mathematical program in the form:

$$\begin{aligned}
 \min \quad & 1^\top x \\
 \text{s.t.} \quad & x_u + x_v \geq 1 \text{ for all } (u, v) \in E \\
 & x_v \in \{0, 1\} \text{ for all } v \in V.
 \end{aligned} \tag{3.2}$$

Example: Set cover: Fire station placement

Gurobipy

In the fire station problem, we seek to choose locations for fire stations such that any district either contains a fire station, or neighbors a district that contains a fire station. Figure 3.2 depicts the set of districts and an example placement of locations of fire stations. How can we minimize the total number of fire stations that we need?

Sets:

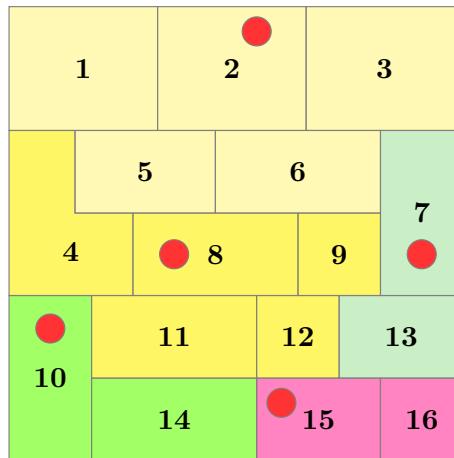
- Let V be the set of districts ($V = \{1, \dots, 16\}$)
- Let V_i be the set of districts that neighbor district i (e.g. $V_1 = \{2, 4, 5\}$).

Variables:

- let $x_i = 1$ if district i is chosen to have a fire station.
- let $x_i = 0$ otherwise.

Model:

$$\begin{aligned}
 \min \quad & \sum_{i \in V} x_i && (\# \text{ open fire stations}) \\
 \text{s.t.} \quad & x_i + \sum_{j \in V_i} x_j \geq 1 && \forall i \in V \quad (\text{Station proximity requirement}) \\
 & x_i \in \{0, 1\} && \text{for } i \in V \quad (\text{station either open or closed})
 \end{aligned}$$

© tikz/Illustration1.pdf²**Figure 3.2: Layout of districts and possible locations of fire stations.****Set Covering - Matrix description:***NP-Complete*

Given a non-negative matrix $A \in \{0, 1\}^{m \times n}$, a non-negative vector, and an objective vector $c \in \mathbb{R}^n$, the set cover problem is

$$\begin{aligned}
 \max \quad & c^\top x \\
 \text{s.t..} \quad & Ax \geq 1 \\
 & x \in \{0, 1\}^n.
 \end{aligned} \tag{3.3}$$

²tikz/Illustration1.pdf, from tikz/Illustration1.pdf. tikz/Illustration1.pdf, tikz/Illustration1.pdf.³tikz/Illustration2.pdf, from tikz/Illustration2.pdf. tikz/Illustration2.pdf, tikz/Illustration2.pdf.⁴tikz/Illustration3.pdf, from tikz/Illustration3.pdf. tikz/Illustration3.pdf, tikz/Illustration3.pdf.

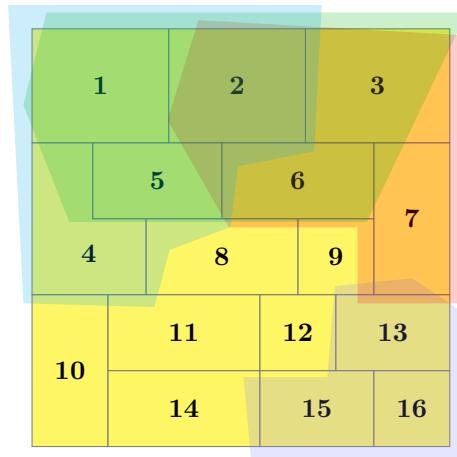
© tikz/Illustration2.pdf³

Figure 3.3: Set cover representation of fire station problem. For example, choosing district 16 to have a fire station covers districts 13, 15, and 16.

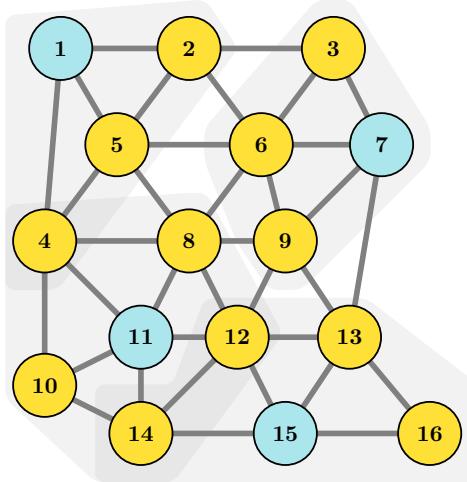
© tikz/Illustration3.pdf⁴

Figure 3.4: Graph representation of fire station problem. Every node is connected to a chosen node by an edge

Example: Vertex Cover with matrix

An alternate way to solve Example: Vertex Cover is to define the *adjacency matrix* A of the graph. The adjacency matrix is a $|E| \times |V|$ matrix with $\{0, 1\}$ entries. The each row corresponds to an edge e and each column corresponds to a node v . For an edge $e = (u, v)$, the corresponding row has a 1 in columns corresponding to the nodes u and v , and a 0 everywhere else. Hence, there are exactly two 1's per row. Applying the formulation above in Set Covering - Matrix description models the problem.

3.3.1. Covering (Generalizing Set Cover)

We could also allow for a more general type of set covering where we have non-negative integer variables and a right hand side that has values other than 1.

Covering:

NP-Complete

Given a non-negative matrix $A \in \mathbb{Z}_+^{m \times n}$, a non-negative vector $b \in \mathbb{Z}^m$, and an objective vector $c \in \mathbb{R}^n$, the set cover problem is

$$\begin{aligned} & \max c^\top x \\ & \text{s.t.. } Ax \geq b \\ & \quad x \in \mathbb{Z}_+^n. \end{aligned} \tag{3.4}$$

3.4 Assignment Problem

The *assignment problem* (machine/person to job/task assignment) seeks to assign tasks to machines in a way that is most efficient. This problem can be thought of as having a set of machines that can complete various tasks (textile machines that can make t-shirts, pants, socks, etc) that require different amounts of time to complete each task, and given a demand, you need to decide how to alloacte your machines to tasks.

Alternatively, you could be an employer with a set of jobs to complete and a list of employees to assign to these jobs. Each employee has various abilities, and hence, can complete jobs in differing amounts of time. And each employee's time might cost a different amout. How should you assign your employees to jobs in order to minimize your total costs?

Assignment Problem:

Given m machines and n jobs, find a least cost assignment of jobs to machines. The cost of assigning job j to machine i is c_{ij} .

Example: Machine Assignment

Sets:

Gurobipy

- Let $I = \{0, 1, 2, 3\}$ set of machines.
- Let $J = \{0, 1, 2, 3\}$ be the set of tasks.

Parameters:

- c_{ij} - the cost of assigning machine i to job j

Variables:

- Let

$$x_{ij} = \begin{cases} 1 & \text{if machine } i \text{ assigned to job } j \\ 0 & \text{otherwise.} \end{cases}$$

Model:

$$\begin{aligned} \min \quad & \sum_{i \in I, j \in J} c_{ij} x_{ij} && \text{(Minimize cost)} \\ \text{s.t.} \quad & \sum_{i \in I} x_{ij} = 1 && \text{for all } j \in J && \text{(All jobs are assigned one machine)} \\ & \sum_{j \in J} x_{ij} = 1 && \text{for all } i \in I && \text{(All machines are assigned to a job)} \\ & x_{ij} \in \{0, 1\} \forall i \in I, j \in J \end{aligned}$$

Example: School Bus Routing Problem

Gurobipy

A school district has a set of schools I and a fleet of school buses J . Each bus needs to be assigned to a school every morning to pick up students. The cost c_{ij} represents the fuel cost for bus j to reach school i and complete the route. The district aims to minimize the total fuel cost while ensuring that each school is served by exactly one bus and each bus is assigned to exactly one school.

The fuel costs can be found in the following table. They are also in csv file format [here](#).

Bus/School	School A	School B	School C	School D	School E
Bus 1	50	80	60	90	100
Bus 2	60	85	55	70	110
Bus 3	75	65	50	85	90
Bus 4	70	90	55	80	95
Bus 5	80	70	60	75	85

We can model this as follows:

Indices:

- i - Index for schools, $i \in I$
- j - Index for buses, $j \in J$

Parameters:

- c_{ij} - Fuel cost for bus j to serve school i .

Decision Variables:

- x_{ij} - 1 if bus j is assigned to school i , 0 otherwise.

Model:

$$\begin{aligned}
 \min \quad & \sum_{i \in I, j \in J} c_{ij} x_{ij} && \text{(Minimize total fuel cost)} \\
 \text{s.t.} \quad & \sum_{i \in I} x_{ij} = 1 && \text{for all } j \in J && \text{(Each bus is assigned to one school)} \\
 & \sum_{j \in J} x_{ij} = 1 && \text{for all } i \in I && \text{(Each school is served by one bus)} \\
 & x_{ij} \in \{0, 1\} \forall i \in I, j \in J
 \end{aligned}$$

3.5 Facility Location

The basic model of the facility location problem is to determine where to place your stores or facilities in order to be close to all of your customers and hence reduce the costs transportation to your customers. Each customer is known to have a certain demand for a product, and each facility has a capacity on how much of that demand it can satisfy. Furthermore, we need to consider the cost of building the facility in a given location.

This basic framework can be applied in many types of problems and there are a number of variants to this problem. We will address two variants: the *capacitated facility location problem* and the *uncapacitated facility location problem*.

3.5.1. Capacitated Facility Location

Capacitated Facility Location:

NP-Complete

Given costs connections c_{ij} and fixed building costs f_i , demands d_j and capacities u_i , the capacitated facility location problem is

Sets:

- Let $I = \{1, \dots, n\}$ be the set of facilities.
- Let $J = \{1, \dots, m\}$ be the set of customers.

Parameters:

- f_i - the cost of opening facility i .
- c_{ij} - the cost of fulfilling the complete demand of customer j from facility i .
- u_i - the capacity of facility i .
- d_j - the demand by customer j .

Variables:

- Let

$$y_i = \begin{cases} 1 & \text{if we open facility } i, \\ 0 & \text{otherwise.} \end{cases}$$

- Let $x_{ij} \geq 0$ be the fraction of demand of customer j satisfied by facility i .

Model:

$$\begin{aligned} \min & \sum_{i=1}^n \sum_{j=1}^m c_{ij}x_{ij} + \sum_{i=1}^n f_i y_i && \text{(total cost)} \\ \text{s.t.} & \sum_{i=1}^n x_{ij} = 1 \text{ for all } j = 1, \dots, m && \text{(assign demand to facility)} \\ & \sum_{j=1}^m d_j x_{ij} \leq u_i y_i \text{ for all } i = 1, \dots, n && \text{(capacity of facility } i) \\ & x_{ij} \geq 0 \text{ for all } i = 1, \dots, n \text{ and } j = 1, \dots, m && \text{(nonnegative fraction of demand satisfied)} \\ & y_i \in \{0, 1\} \text{ for all } i = 1, \dots, n && \text{(open/not open facility)} \end{aligned}$$

$$\begin{aligned}
 \min \quad & \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij} + \sum_{i=1}^m f_i y_i \\
 \text{s.t.} \quad & \sum_{i=1}^m x_{ij} = 1, \quad j = 1, \dots, n \\
 & \sum_{j=1}^m d_j x_{ij} \leq u_i \text{ for all } i = 1, \dots, n \quad (\text{capacity of facility } i) \\
 & x_{ij} \leq y_i, \quad i = 1, \dots, m, \quad j = 1, \dots, n \\
 & x_{ij} \geq 0 \quad i = 1, \dots, m, \quad j = 1, \dots, n \\
 & y_i \in \{0, 1\}, \quad i = 1, \dots, m
 \end{aligned}$$

Example: Capacitated Facility Location: Retail Distribution Example

Gurobipy

Context: A retail company plans to establish distribution centers across a country to serve its stores efficiently. The company faces decisions on which distribution centers to open and which stores each center should serve. Costs associated with opening each center and serving stores from them are known, and each center has a maximum capacity. Additionally, each store has a known demand.

*Given Data: (distribution-data.xlsx)

- Number of potential distribution centers (n): 3
- Number of stores (m): 4

Costs to Open Distribution Centers (f_i):

- Distribution Center 1: \$150,000
- Distribution Center 2: \$100,000
- Distribution Center 3: \$180,000

Cost to Serve a Store from a Distribution Center (c_{ij}):

	Store 1	Store 2	Store 3	Store 4
Center 1	\$50	\$60	\$70	\$85
Center 2	\$75	\$45	\$55	\$50
Center 3	\$65	\$80	\$40	\$60

Capacity of Distribution Centers (u_i):

- Distribution Center 1: 100 units
- Distribution Center 2: 80 units

- Distribution Center 3: 90 units

Demand by Each Store (d_j):

- Store 1: 40 units
- Store 2: 30 units
- Store 3: 20 units
- Store 4: 25 units

*Model Formulation: The capacitated facility location model described earlier can be applied to this scenario with the given data to determine the optimal number and location of distribution centers to open, and which stores each center should serve.

3.5.2. Uncapacitated Facility Location

Uncapacitated Facility Location:

NP-Complete

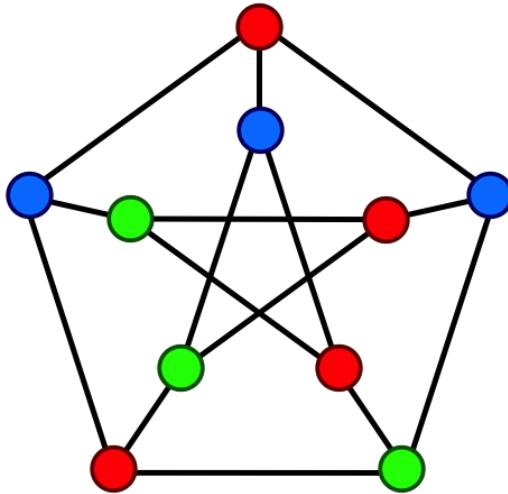
Given costs connections c_{ij} and fixed building costs f_i , the uncapacitated facility location problem is

$$\begin{aligned}
 \min \quad & \sum_{i=1}^n \sum_{j=1}^m c_{ij} z_{ij} + \sum_{i=1}^n f_i x_i \\
 \text{s.t.} \quad & \sum_{i=1}^n z_{ij} = 1 \text{ for all } j = 1, \dots, m \\
 & \sum_{j=1}^m z_{ij} \leq M x_i \text{ for all } i = 1, \dots, n \\
 & z_{ij} \in \{0, 1\} \text{ for all } i = 1, \dots, n \text{ and } j = 1, \dots, m \\
 & x_i \in \{0, 1\} \text{ for all } i = 1, \dots, n
 \end{aligned} \tag{3.1}$$

Here M is a large number and can be chosen as $M = m$, but could be refined smaller if more context is known.

Alternative model!

$$\begin{aligned} \min \quad & \sum_{i=1}^n \sum_{j=1}^m c_{ij} z_{ij} + \sum_{i=1}^n f_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n z_{ij} = 1 \text{ for all } j = 1, \dots, m \\ & z_{ij} \leq x_i \text{ for all } i = 1, \dots, n \text{ for all } j = 1, \dots, m \\ & z_{ij} \in \{0, 1\} \text{ for all } i = 1, \dots, n \text{ and } j = 1, \dots, m \\ & x_i \in \{0, 1\} \text{ for all } i = 1, \dots, n \end{aligned} \tag{3.2}$$



© wiki/File/Petersen-graph-3-coloring.png⁵

Figure 3.5: wiki/File/Petersen-graph-3-coloring.png

3.6 Graph Coloring

Graph coloring Figure 3.5 is is problem of finding a minimum coloring in a graph can be formulated in many ways. Each vertex is assigned a color and two vertices cannot share the same color if they are connected by an edge.

In order to present integer programming formulations, we use x_{ij} to denote binary variables, with $i \in V$ and $1 \leq j \leq n$, where $x_{ij} = 1$ if color j is assigned to vertex i and $x_{ij} = 0$ otherwise. We also define n binary variables w_j for $j = 1, \dots, n$, that indicate whether color j is used in some node, i.e., $w_j = 1$ if $x_{ij} = 1$ for some vertex i . The following is a classical integer programming formulation:

$$\begin{aligned} & \min \sum_{j=1}^n w_j \\ & \sum_{j=1}^n x_{ij} = 1 \quad \forall i \in V, \\ & x_{ij} + x_{kj} \leq w_j \quad \forall \{i, k\} \in E, \quad 1 \leq j \leq n, \\ & x_{ij} \in \{0, 1\} \quad \forall i \in V, 1 \leq j \leq n \\ & w_j \in \{0, 1\} \quad 1 \leq j \leq n. \end{aligned}$$

We can improve upon this model. We suggest two possible models from [MENDEZDIAZ2008159]. Please read their paper for more advanced models with additional families of cuts added.

Color order model: Given a (k) -coloring, selecting any k colors from the set $\{1, \dots, n\}$ results in a valid solution, and these solutions are deemed equivalent. To reduce redundancy among these solutions,

⁵wiki/File/Petersen-graph-3-coloring.png, from wiki/File/Petersen-graph-3-coloring.png.
wiki/File/Petersen-graph-3-coloring.png, wiki/File/Petersen-graph-3-coloring.png.

we stipulate that color j can only be allocated to a vertex if color $j - 1$ has previously been assigned. By doing so, we exclude symmetrical (k) -colorings that use colors with labels exceeding k . To define these valid solutions, we simply need to incorporate the subsequent set of constraints:

$$\begin{aligned} w_j &\leq \sum_{i \in V} x_{ij} \quad \forall 1 \leq j \leq n, \\ w_j &\geq w_{j+1} \quad \forall 1 \leq j \leq n-1. \end{aligned}$$

The reduction of feasible solutions is very significant.

Independent set order model: In a (k) -coloring, permutations of the initial k colors lead to equivalent outcomes. To reduce the occurrence of these similar solutions, the subsequent model introduces stricter constraints than the previous one. It mandates that the count of vertices colored with j should be at least equal to or surpass the count of vertices colored with $j + 1$. We incorporate the ensuing inequalities:

$$\begin{aligned} w_j &\leq \sum_{i \in V} x_{ij} \quad \forall 1 \leq j \leq n \\ \sum_{i=1}^n x_{ij} &\geq \sum_{i=1}^n x_{ij+1} \quad \forall 1 \leq j \leq n-1. \end{aligned}$$

As an exercise, try implementing these different models and compare the solve times.

3.7 Basic Modeling Tricks - Using Binary Variables

In this section, we describe ways to model a variety of constraints that commonly appear in practice. The goal is changing constraints described in words to constraints defined by math.

Binary variables can allow you to model many types of constraints. We discuss here various logical constraints where we assume that $x_i \in \{0, 1\}$ for $i = 1, \dots, n$. We will take the meaning of the variable to be selecting an item.

1. If item i is selected, then item j is also selected.

$$x_i \leq x_j \quad (3.1)$$

- (a) If any of items $1, \dots, 5$ are selected, then item 6 is selected.

$$x_1 + x_2 + \dots + x_5 \leq 5 \cdot x_6 \quad (3.2)$$

Alternatively!

$$x_i \leq x_6 \quad \text{for all } i = 1, \dots, 5 \quad (3.3)$$

2. If item j is not selected, then item i is not selected.

$$x_i \leq x_j \quad (3.4)$$

- (a) If item j is not selected, then all items $1, \dots, i$ are not selected.

$$x_1 + x_2 + \dots + x_i \leq i \cdot x_j \quad (3.5)$$

3. If item j is not selected, then item i is not selected.

$$x_i \leq x_j \quad (3.6)$$

4. Either item i is selected or item j is selected, but not both.

$$x_i + x_j = 1 \quad (3.7)$$

5. Item i is selected or item j is selected or both.

$$x_i + x_j \geq 1 \quad (3.8)$$

6. If item i is selected, then item j is not selected.

$$x_j \leq (1 - x_i) \quad (3.9)$$

7. At most one of items i, j , and k are selected.

$$x_i + x_j + x_k \leq 1 \quad (3.10)$$

8. At most two of items i, j , and k are selected.

$$x_i + x_j + x_k \leq 2 \quad (3.11)$$

9. Exactly one of items i, j , and k are selected.

$$x_i + x_j + x_k = 1 \quad (3.12)$$

These tricks can be connected to create different function values.

Example 3.1: Variable takes one of three values

Suppose that the variable x should take one of the three values $\{4, 8, 13\}$. This can be modeled using three binary variables as

$$\begin{aligned} x &= 4z_1 + 8z_2 + 13z_3 \\ z_1 + z_2 + z_3 &= 1 \\ z_i &\in \{0, 1\} \text{ for } i = 1, 2, 3. \end{aligned}$$

As a convenient addition, if we want to add the possibility that it takes the value 0, then we can model this as

$$\begin{aligned} x &= 4z_1 + 8z_2 + 13z_3 \\ z_1 + z_2 + z_3 &\leq 1 \\ z_i &\in \{0, 1\} \text{ for } i = 1, 2, 3. \end{aligned}$$

We can also model variable increases at different amounts.

Example 3.2: Discount for buying more

Suppose you can choose to buy 1, 2, or 3 units of a product, each with a decreasing cost. The first unit is \$10, the second is \$5, and the third unit is \$3.

$$\begin{aligned} x &= 10z_1 + 5z_2 + 3z_3 \\ z_1 &\geq z_2 \geq z_3 \\ z_i &\in \{0, 1\} \text{ for } i = 1, 2, 3. \end{aligned}$$

Here, z_i represents if we buy the i th unit. The inequality constraints impose that if we buy unit j , then we must buy all units i with $i < j$.

In this section, we describe ways to model a variety of constraints that commonly appear in practice. The goal is changing constraints described in words to constraints defined by math.

3.7.1. Big M constraints - Activating/Deactivating Inequalities

Big M comes again! It's extremely useful when trying to activate constraints based on a binary variable.

For instance, if we don't rent a bus, then we can have at most 3 passengers join us on our trip. Consider passengers A, B, C, D, E and let $x_i \in \{0, 1\}$ be 1 if we take passenger i and 0 otherwise. We can model the constraint that we can have at most 5 passengers as

$$x_A + x_B + x_C + x_D + x_E \leq 3.$$

We want to be able to activate this constraint in the event that we don't rent a bus.

Let $\delta \in \{0, 1\}$ be 1 if rent a bus, and 0 otherwise.

Then we want to say

If $\delta = 0$, then

$$x_A + x_B + x_C + x_D + x_E \leq 3.$$

We can formulate this using a big-M constraint as

$$x_A + x_B + x_C + x_D + x_E \leq 3 + M\delta. \quad (3.13)$$

Notice the two case

$$\begin{cases} x_A + x_B + x_C + x_D + x_E \leq 3 & \text{if } \delta = 0 \\ x_A + x_B + x_C + x_D + x_E \leq 3 + M & \text{if } \delta = 1 \end{cases}$$

In the second case, we choose M to be so large, that the second case inequality is vacuous. That said, choosing smaller M values (that are still valid) will help the computer program solve the problem faster. In this case, it suffices to let $M = 2$.

We can speak about this technique more generally as

Big-M: If then:

We aim to model the relationship

$$\text{If } \delta = 0, \text{ then } a^\top x \leq b. \quad (3.14)$$

By letting M be an upper bound on the quantity $a^\top x - b$, we can model this condition as

$$\begin{aligned} a^\top x - b &\leq M\delta \\ \delta &\in \{0, 1\} \end{aligned} \quad (3.15)$$

3.7.2. Either Or Constraints

“At least one of these constraints holds” is what we would like to model. Equivalently, we can phrase this as an *inclusive or* constraint. This can be modeled with a pair of Big-M constraints.

Either Or:

$$\text{Either } a^\top x \leq b \text{ or } c^\top x \leq d \text{ holds} \quad (3.16)$$

can be modeled as

$$\begin{aligned} a^\top x - b &\leq M_1 \delta \\ c^\top x - d &\leq M_2(1 - \delta) \\ \delta &\in \{0, 1\}, \end{aligned} \quad (3.17)$$

where M_1 is an upper bound on $a^\top x - b$ and M_2 is an upper bound on $c^\top x - d$.

Example 3.3

Either 2 buses or 10 cars are needed shuttle students to the football game.

- Let x be the number of buses we have and
- let y be the number of cars that we have.

Suppose that there are at most $M_1 = 5$ buses that could be rented and at most $M_2 = 20$ cars that could be available.

This constraint can be modeled as

$$\begin{aligned} x - 2 &\leq 5\delta \\ y - 10 &\leq 20(1 - \delta) \\ \delta &\in \{0, 1\}, \end{aligned} \quad (3.18)$$

3.7.3. If then implications - opposite direction

Suppose that we want to model the fact that if we have at most 10 students attending this course, then we must switch to a smaller classroom.

Let $x_i \in \{0, 1\}$ be 1 if student i is in the course or not. Let $\delta \in \{0, 1\}$ be 1 if we need to switch to a smaller classroom.

Thus, we want to model

If

$$\sum_{i \in I} x_i \leq 10$$

then

$$\delta = 1.$$

We can model this as

$$\sum_{i \in I} x_i \geq 10 + 1 + M\delta. \quad (3.19)$$

If inequality, then indicator:

W

Let m be a lower bound on the quantity $a^\top x - b$ and we let ε be a tiny number that is an error bound in verifying if an inequality is violated. **If the data a, b are integer and x is an integer, then we can take $\varepsilon = 1$.**

Now

$$\text{If } a^\top x \leq b \text{ then } \delta = 1 \quad (3.20)$$

can be modeled as

$$a^\top x - b \geq \varepsilon(1 - \delta) + m\delta. \quad (3.21)$$

Proof. We now justify the statement above.

A simple way to understand this constraint is to consider the *contrapositive* of the if then statement that we want to model. The contrapositive says that

$$\text{If } \delta = 0, \text{ then } a^\top x - b > 0. \quad (3.22)$$

To show the contrapositive, we set $\delta = 0$. Then the inequality becomes

$$a^\top x - b \geq \varepsilon(1 - 0) + m0 = \varepsilon > 0.$$

Thus, the contrapositive holds.

If instead we wanted a direct proof:

Case 1: Suppose $a^\top x \leq b$. Then $0 \geq a^\top x - b$, which implies that

$$\delta(a^\top x - b) \geq a^\top x - b$$

Therefore

$$\delta(a^\top x - b) \geq \varepsilon(1 - \delta) + m\delta$$

After rearranging

$$\delta(a^\top x - b - m) \geq \varepsilon(1 - \delta)$$

Implication	Constraint
If $\delta = 0$, then $a^\top x \leq b$	$a^\top x \leq b + M\delta$
If $a^\top x \leq b$, then $\delta = 1$	$a^\top x \geq m\delta + \varepsilon(1 - \delta)$

Table 3.1: Short list: If/then models with a constraint and a binary variable. Here M and m are upper and lower bounds on $a^\top x - b$ and ε is a small number such that if $a^\top x > b$, then $a^\top x \geq b + \varepsilon$.

Implication	Constraint
If $\delta = 0$, then $a^\top x \leq b$	$a^\top x \leq b + M\delta$
If $\delta = 0$, then $a^\top x \geq b$	$a^\top x \geq b + m\delta$
If $\delta = 1$, then $a^\top x \leq b$	$a^\top x \leq b + M(1 - \delta)$
If $\delta = 1$, then $a^\top x \geq b$	$a^\top x \geq b + m(1 - \delta)$
If $a^\top x \leq b$, then $\delta = 1$	$a^\top x \geq b + m\delta + \varepsilon(1 - \delta)$
If $a^\top x \geq b$, then $\delta = 1$	$a^\top x \leq b + M\delta - \varepsilon(1 - \delta)$
If $a^\top x \leq b$, then $\delta = 0$	$a^\top x \geq b + m(1 - \delta) + \varepsilon\delta$
If $a^\top x \geq b$, then $\delta = 0$	$a^\top x \geq b + m(1 - \delta) - \varepsilon\delta$

Table 3.2: Long list: If/then models with a constraint and a binary variable. Here M and m are upper and lower bounds on $a^\top x - b$ and ε is a small number such that if $a^\top x > b$, then $a^\top x \geq b + \varepsilon$.

Since $a^\top x - b - m \geq 0$ and $\varepsilon > 0$, the only feasible choice is $\delta = 1$.

Case 2: Suppose $a^\top x > b$. Then $a^\top x - b \geq \varepsilon$. Since $a^\top x - b \geq m$, both choices $\delta = 0$ and $\delta = 1$ are feasible.

By the choice of ε , we know that $a^\top x - b > 0$ implies that $a^\top x - b \geq \varepsilon$.

Since we don't like strict inequalities, we write the strict inequality as $a^\top x - b \geq \varepsilon$ where ε is a small positive number that is a smallest difference between $a^\top x - b$ and 0 that we would typically observe. As mentioned above, if a, b, x are all integer, then we can use $\varepsilon = 1$.

Now we want an inequality with left hand side $a^\top x - b \geq$ and right hand side to take the value

- ε if $\delta = 0$,
- m if $\delta = 1$.

This is accomplished with right hand side $\varepsilon(1 - \delta) + m\delta$.



Many other combinations of if then statements are summarized in the following table: These two implications can be used to derive the following longer list of implications.

Lastly, if you insist on having exact correspondance, that is, " $\delta = 0$ if and only if $a^\top x \leq b$ " you can simply include both constraints for "if $\delta = 0$, then $a^\top x \leq b$ " and if " $a^\top x \leq b$, then $\delta = 0$ ". Although many problems may be phrased in a way that suggests you need "if and only if", it is often not necessary to use both constraints due to the objectives in the problem that naturally prevent one of these from happening.

For example, if we want to add a binary variable δ that means

$$\begin{cases} \delta = 0 \text{ implies } a^\top x \leq b \\ \delta = 1 \text{ Otherwise} \end{cases}$$

If $\delta = 1$ does not effect the rest of the optimization problem, then adding the constraint regarding $\delta = 1$ is not necessary. Hence, typically, in this scenario, we only need to add the constraint $a^\top x \leq b + M\delta$.

3.7.4. Multi Term Disjunction with application to 2D packing

A disjunction is a generalization of an “or” statement. Suppose that we have n constraints

$$\mathbf{a}_1^\top \mathbf{x} \leq b_1, \quad \mathbf{a}_2^\top \mathbf{x} \leq b_2, \quad \dots, \quad \mathbf{a}_n^\top \mathbf{x} \leq b_n$$

and we want to enforce at least k of them. This can be accomplished linearly by introducing a new binary indicator variable δ_i for each of the disjunctive constraints $i \in \{1, \dots, n\}$:

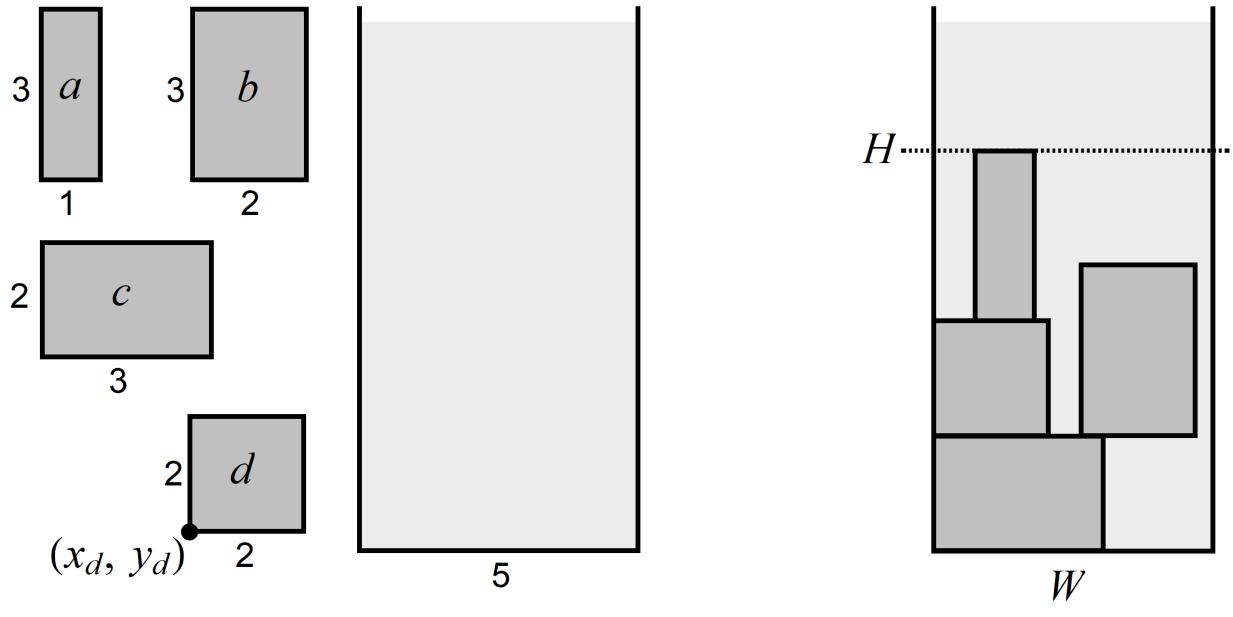
$$\begin{aligned} \mathbf{a}_1^\top \mathbf{x} &\leq b_1 + M(1 - \delta_1) \\ \mathbf{a}_2^\top \mathbf{x} &\leq b_2 + M(1 - \delta_2) \\ &\vdots \\ \mathbf{a}_n^\top \mathbf{x} &\leq b_n + M(1 - \delta_n) \\ \sum_{i=1}^n \delta_i &\geq k \end{aligned}$$

If $\delta_i = 1$, then the i^{th} disjunctive constraint is actively enforced. The last constraint ($\sum_{i=1}^n \delta_i \geq k$) ensures that at least k of the constraints is active.

3.7.4.1. Strip Packing Problem

Suppose we a selection of rectangles I and a 2-dimensional strip with width W and infinite height. Each rectangle $i \in I$ has width w_i and height h_i and we want to pack the rectangles into the strip so that (3.23a) overall height is minimized, (3.23b) overall width is less than W , and (3.23c) none of the rectangles overlap. See Figure 3.6 for an example.

Let (x_i, y_i) denote the position of the lower-left-hand corner of each rectangle $i \in I$. The overlapping constraint (3.23c) is the trickiest part. Consider a pair of rectangles i and j : rectangle j is located entirely to the left of rectangle i if x_j has a value larger than $x_j + w_j$. That is, if $x_j \geq x_j + w_j$. On the other hand, if $y_j \geq y_i + h_i$, then rectangle j is located entirely above rectangle i . If either of these constraints is satisfied then rectangles i and j do not overlap. We could also place i above or to the right of j . This gives four



(a) Pack the rectangles into the strip

(b) Minimize overall height *H*.

Figure 3.6: An Example of the Strip Packing Problem

constraints that we need to satisfy at least one of. The model can be thought of thusly:

$$\text{Minimize } \max_{i \in I} \{y_i + h_i\} \quad (3.23a)$$

$$\text{s.t. } \max_{i \in I} \{x_i + w_i\} \leq W \quad (3.23b)$$

$$\left. \begin{array}{l} x_i + w_i \leq x_j \\ x_j + w_j \leq x_i \\ y_i + h_i \leq y_j \\ y_j + h_j \leq y_i \end{array} \right\} \begin{array}{l} \text{At least one of these} \\ \text{for every distinct pairs} \\ \text{of rectangles } i, j \in I \end{array} \quad (3.23c)$$

$$x_i, y_i \geq 0 \quad \forall i \in I \quad (3.23d)$$

Constraint (3.23c) is a set of four *disjunctive* constraints. This can be expressed linearly by introducing a

new binary indicator variable δ_{ijk} for each of the disjunctive constraints in (3.23c):

$$\begin{aligned} \text{Minimize} \quad & H \\ \text{s.t.} \quad & y_i + h_i \leq H \quad \forall i \in I \quad (3.23a) \\ & x_i + w_i \leq W \quad \forall i \in I \quad (3.23b) \\ & x_i + w_i \leq x_j + M(1 - \delta_{ij1}) \quad \forall i, j \in I : i > j \\ & x_j + w_j \leq x_i + M(1 - \delta_{ij2}) \quad \forall i, j \in I : i > j \\ & y_i + h_i \leq y_j + M(1 - \delta_{ij3}) \quad \forall i, j \in I : i > j \quad (3.23c) \\ & y_j + h_j \leq y_i + M(1 - \delta_{ij4}) \quad \forall i, j \in I : i > j \\ & \sum_{k=1}^4 \delta_{ijk} \geq 1 \quad \forall i, j \in I : i > j \\ & x_i, y_i \geq 0 \quad \forall i \in I \\ & \delta_{ij} \in \{0, 1\}^4 \quad \forall i, j \in I : i > j \end{aligned}$$

If $\delta_{ijk} = 1$, then the k^{th} disjunctive constraint is actively enforced. The last constraint ($\sum_{k=1}^4 \delta_{ijk} \geq 1$) ensures that at least one of the constraints is active. An optimal solution to the example is given in Figure 3.7; it was found via GurobiPy.

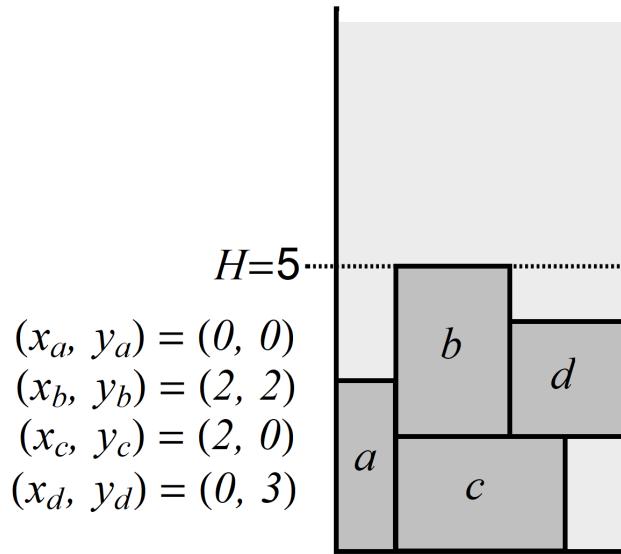


Figure 3.7: The Optimal Solution to the example problem

This problem is considered strongly NP-Hard.

3.7.5. SOS1 Constraints

Definition 3.4: Special Ordered Sets of Type 1 (SOS1)

Special Ordered Sets of type 1 (SOS1) constraint on a vector indicates that at most one element of the vector can non-zero.

We next give an example of how to use binary variables to model this and then show how much simpler it can be coded using the SOS1 constraint.

Example: SOS1 Constraints

Gurobipy

Solve the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 4x_2 + x_3 + 5x_4 \\ \text{subject to} \quad & 0 \leq x_i \leq 5 \\ & \text{at most one of the } x_i \text{ can be nonzero} \end{aligned}$$

3.7.6. SOS2 Constraints

Definition 3.5: Special Ordered Sets of Type 2 (SOS2)

A Special Ordered Set of Type 2 (SOS2) constraint on a vector indicates that at most two elements of the vector can non-zero AND the non-zero elements must appear consecutively.

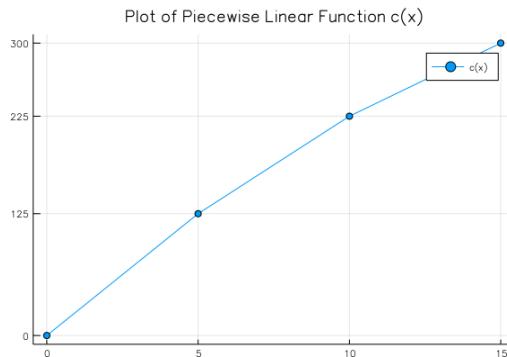
We next give an example of how to use binary variables to model this and then show how much simpler it can be coded using the SOS2 constraint.

Example: SOS2

Gurobipy

Solve the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 4x_2 + x_3 + 5x_4 \\ \text{subject to} \quad & 0 \leq x_i \leq 5 \\ & \text{at most two of the } x_i \text{ can be nonzero} \\ & \text{and the nonzero } x_i \text{ must be consecutive} \end{aligned}$$

© pwl-plot.png⁶**Figure 3.8: scale = 0.35**

3.7.7. Piecewise linear functions with SOS2 constraint

Example: Piecewise Linear Function

Gurobipy

Consider the piecewise linear function $c(x)$ given by

$$c(x) = \begin{cases} 25x & \text{if } 0 \leq x \leq 5 \\ 20x + 25 & \text{if } 5 \leq x \leq 10 \\ 15x + 75 & \text{if } 10 \leq x \leq 15 \end{cases}$$

See Figure 3.8 .

We will use integer programming to describe this function. We will fix $x = a$ and then the integer program will set the value y to $c(a)$.

$$\begin{aligned} \min \quad & 0 \\ \text{Subject to} \quad & x = 5z_2 + 10z_3 + 15z_4 \\ & y = 125z_2 + 225z_3 + 300z_4 \\ & z_1 + z_2 + z_3 + z_4 = 1 \\ & \text{SOS2 : } \{z_1, z_2, z_3, z_4\} \\ & 0 \leq z_i \leq 1 \quad \forall i \in \{1, 2, 3, 4\} \\ & x = a \end{aligned}$$

⁶pwl-plot.png, from pwl-plot.png. pwl-plot.png, pwl-plot.png.

Example: Piecewise Linear Function Application

Gurobipy

Consider the following optimization problem where the objective function includes the term $c(x)$, where $c(x)$ is the piecewise linear function described in Example 12:

$$\max z = 12x_{11} + 12x_{21} + 14x_{12} + 14x_{22} - c(x) \quad (3.24)$$

$$\text{s.t. } x_{11} + x_{12} \leq x + 5 \quad (3.25)$$

$$x_{21} + x_{22} \leq 10 \quad (3.26)$$

$$0.5x_{11} - 0.5x_{21} \geq 0 \quad (3.27)$$

$$0.4x_{12} - 0.6x_{22} \geq 0 \quad (3.28)$$

$$x_{ij} \geq 0 \quad (3.29)$$

$$0 \leq x \leq 15 \quad (3.30)$$

Given the piecewise linear, we can model the whole problem explicitly as a mixed-integer linear program.

$$\begin{aligned}
 \max \quad & 12X_{1,1} + 12X_{2,1} + 14X_{1,2} + 14X_{2,2} - y \\
 \text{Subject to} \quad & x - 5z_2 - 10z_3 - 15z_4 = 0 \\
 & y - 125z_2 - 225z_3 - 300z_4 = 0 \\
 & z_1 + z_2 + z_3 + z_4 = 1 \\
 & X_{1,1} + X_{1,2} - x \leq 5 \\
 & X_{2,1} + X_{2,2} \leq 10 \\
 & 0.5X_{1,1} - 0.5X_{2,1} \geq 0 \\
 & 0.4X_{1,2} - 0.6X_{2,2} \geq 0 \\
 & \text{SOS2 : } \{z_1, z_2, z_3, z_4\} \\
 & \quad \begin{array}{ll} X_{i,j} \geq 0 & \forall i \in \{1,2\}, j \in \{1,2\} \\ 0 \leq z_i \leq 1 & \forall i \in \{1,2,3,4\} \\ 0 \leq x \leq 15 & \\ y \text{ free} & \end{array} \\
 \end{aligned} \quad (3.31)$$

3.7.7.1. SOS2 with binary variables

Modeling Piecewise linear function

- Write down pairs of breakpoints and functions values $(a_i, f(a_i))$.
- Define a binary variable z_i indicating if x is in the interval $[a_i, a_{i+1}]$.
- Define multipliers λ_i such that x is a combination of the a_i 's and therefore the output $y = f(x)$ is a combination of the $f(a_i)$'s.
- Restrict that at most 2 λ_i 's are non-zero and that those 2 are consecutive.

$$\begin{aligned}
 & \min \sum_{i=1}^k \lambda_i f(a_i) \\
 \text{s.t. } & \sum_{i=1}^k \lambda_i = 1 \\
 & x = \sum_{i=1}^k \lambda_i a_i \\
 & \lambda_1 \leq z_1 \\
 & \lambda_i \leq z_{i-1} + z_i \quad \text{for } i = 2, \dots, k-1, \\
 & \lambda_k \leq z_{k-1} \\
 & \lambda_i \geq 0, z_i \in \{0, 1\}.
 \end{aligned}$$

3.7.8. Maximizing a minimum

When the constraints could be general, we will write $x \in X$ to define general constraints. For instance, we could have $X = \{x \in \mathbb{R}^n : Ax \leq b\}$ or $X = \{x \in \mathbb{R}^n : Ax \leq b, x \in \mathbb{Z}^n\}$ or many other possibilities.

Consider the problem

$$\begin{aligned}
 & \max \quad \min\{x_1, \dots, x_n\} \\
 \text{such that} \quad & x \in X
 \end{aligned}$$

Having the minimum on the inside is inconvenient. To remove this, we just define a new variable y and enforce that $y \leq x_i$ and then we maximize y . Since we are maximizing y , it will take the value of the smallest x_i . Thus, we can recast the problem as

$$\begin{aligned} \max \quad & y \\ \text{such that} \quad & y \leq x_i \text{ for } i = 1, \dots, n \\ & x \in X \end{aligned}$$

3.7.9. Relaxing (nonlinear) equality constraints

There are a number of scenarios where the constraints can be relaxed without sacrificing optimal solutions to your problem. In a similar vein of the maximizing a minimum, if because of the objective we know that certain constraints will be tight at optimal solutions, we can relax the equality to an inequality. For example,

$$\begin{aligned} \max \quad & x_1 + x_2 + \dots + x_n \\ \text{such that} \quad & x_i = y_i^2 + z_i^2 \text{ for } i = 1, \dots, n \end{aligned}$$

3.7.10. Exact absolute value

Suppose we need to model an exact equality

$$|x| = t$$

It defines a non-convex set, hence it is not conic representable. If we split x into positive and negative part $x = x^+ - x^-$, where $x^+, x^- \geq 0$, then $|x| = x^+ + x^-$ as long as either $x^+ = 0$ or $x^- = 0$. That last alternative can be modeled with a binary variable, and we get a model of :

$$\begin{aligned} x &= x^+ - x^- \\ t &= x^+ + x^- \\ 0 &\leq x^+, x^- \\ x^+ &\leq Mz \\ x^- &\leq M(1-z) \\ z &\in \{0, 1\} \end{aligned}$$

where the constant M is an a priori known upper bound on $|x|$ in the problem.

3.7.10.1. Exact ℓ_1 -norm

We can use the technique above to model the exact ℓ_1 -norm equality constraint

$$\sum_{i=1}^n |x_i| = c$$

where $x \in \mathbb{R}^n$ is a decision variable and c is a constant. Such constraints arise for instance in fully invested portfolio optimizations scenarios (with short-selling). As before, we split x into a positive and negative part, using a sequence of binary variables to guarantee that at most one of them is nonzero:

$$\begin{aligned} x &= x^+ - x^- \\ 0 &\leq x^+, x^-, \\ x^+ &\leq cz \\ x^- &\leq c(e - z), \\ \sum_i x_i^+ + \sum_i x_i^- &= c, \\ z &\in \{0, 1\}^n, x^+, x^- \in \mathbb{R}^n \end{aligned}$$

3.7.10.2. Maximum

The exact equality $t = \max \{x_1, \dots, x_n\}$ can be expressed by introducing a sequence of mutually exclusive indicator variables z_1, \dots, z_n , with the intention that $z_i = 1$ picks the variable x_i which actually achieves maximum. Choosing a safe bound M we get a model:

$$\begin{aligned} x_i &\leq t \leq x_i + M(1 - z_i), i = 1, \dots, n \\ z_1 + \dots + z_n &= 1, \\ z &\in \{0, 1\}^n \end{aligned}$$

3.8 Network Flow

3.8.1. Maximum flow

The network flow problem is a fundamental problem in operations research and computer science, with applications in a wide variety of fields. It involves finding the maximum amount of flow that can be sent from a source node to a sink node in a network, without exceeding the capacities of the individual arcs, and ensuring that the flow on each arc is non-negative. This problem arises in many real-world situations, such as in transportation, where it can be used to find the maximum amount of goods that can be transported from a factory to a warehouse, or in telecommunications, where it can be used to find the maximum amount of data that can be sent from one server to another. Other applications include supply chain optimization, water distribution, and electrical power grid optimization. In this section, we will describe a linear programming formulation for the network flow problem.

A network can be described as a directed graph $G = (N, A)$, where N is the set of nodes and A is the set of arcs. Each arc $(i, j) \in A$ has a capacity u_{ij} , which is the maximum amount of flow that can traverse the arc from node i to node j .

The network flow problem can be defined as follows: Given a network $G = (N, A)$, a source node s , a sink node t , and capacities u_{ij} on the arcs, find the maximum flow from s to t that respects the capacity constraints.

Sets:

- N : Set of nodes in the network.
- A : Set of arcs in the network.

Parameters:

- u_{ij} : Capacity of arc $(i, j) \in A$, which is the maximum amount of flow that can traverse the arc from node i to node j .
- s : Source node.
- t : Sink node.

Variables:

- f_{ij} : Flow on arc $(i, j) \in A$.

Model: The network flow problem can be formulated as a linear programming problem as follows:

$$\begin{aligned}
 & \text{Maximize} && \sum_{j:(s,j) \in A} f_{sj} - \sum_{j:(j,s) \in A} f_{js} \\
 & \text{Subject to} && \sum_{j:(i,j) \in A} f_{ij} - \sum_{j:(j,i) \in A} f_{ji} = 0, \quad \forall i \in N \setminus \{s,t\} \\
 & && \sum_{j:(s,j) \in A} f_{sj} - \sum_{j:(j,s) \in A} f_{js} = \sum_{j:(j,t) \in A} f_{jt} - \sum_{j:(t,j) \in A} f_{tj} \\
 & && 0 \leq f_{ij} \leq u_{ij}, \quad \forall (i,j) \in A
 \end{aligned}$$

Objective Function - The objective is to maximize the total flow from the source node s to the sink node t .

Constraints - Flow Balance: The first set of constraints ensures that the flow into each node is equal to the flow out of the node, except for the source and sink nodes.

- Flow Conservation: The second constraint ensures that the flow out of the source node is equal to the flow into the sink node.
- Capacity Constraints: The third set of constraints ensures that the flow on each arc does not exceed its capacity.

There are of course many variations on this problem. Most commonly is to compute a minimum cost network flow.

3.8.2. Minimum Cost Network Flow

The minimum cost network flow problem is an extension of the network flow problem that considers not only the capacities of the arcs, but also the costs associated with sending flow along the arcs and the demands at the nodes. The objective is to find the flow of minimum cost that satisfies the demands at the nodes and the capacity constraints on the arcs.

A network can be described as a directed graph $G = (N, A)$, where N is the set of nodes and A is the set of arcs. Each arc $(i, j) \in A$ has a capacity u_{ij} and a cost c_{ij} . Each node $i \in N$ has a demand d_i , which can be positive, negative, or zero. A positive demand means that the node requires that amount of flow, a negative demand means that the node supplies that amount of flow, and a zero demand means that the node neither requires nor supplies flow.

The minimum cost network flow problem can be defined as follows: Given a network $G = (N, A)$, capacities u_{ij} and costs c_{ij} on the arcs, and demands d_i at the nodes, find the flow f_{ij} on the arcs that minimizes the total cost of the flow, while satisfying the demands at the nodes and the capacity constraints on the arcs.

Sets:

- N : Set of nodes in the network.
- A : Set of arcs in the network.

Parameters:

- u_{ij} : Capacity of arc $(i, j) \in A$.
- c_{ij} : Cost per unit of flow on arc $(i, j) \in A$.
- d_i : Demand at node $i \in N$.

Variables:

- f_{ij} : Flow on arc $(i, j) \in A$.

Model: The minimum cost network flow problem can be formulated as a linear programming problem as follows:

$$\begin{aligned} \text{Minimize} \quad & \sum_{(i,j) \in A} c_{ij} f_{ij} \\ \text{Subject to} \quad & \sum_{j:(i,j) \in A} f_{ij} - \sum_{j:(j,i) \in A} f_{ji} = d_i, \quad \forall i \in N \\ & 0 \leq f_{ij} \leq u_{ij}, \quad \forall (i,j) \in A \end{aligned}$$

Objective Function - The objective is to minimize the total cost of the flow.

Constraints - Flow Conservation: The first set of constraints ensures that the flow into each node minus the flow out of the node is equal to the demand at the node.

- Capacity Constraints: The second set of constraints ensures that the flow on each arc does not exceed its capacity.

3.8.3. Multi-Commodity Minimum Cost Network Flow with Integrality Constraints

The multi-commodity minimum cost network flow problem is a generalization of the minimum cost network flow problem that considers multiple commodities flowing through the network. Each commodity has its own demand at each node and its own cost on each arc. The objective is to find the integer flow of minimum cost that satisfies the demands of all commodities at the nodes and the capacity constraints on the arcs.

A network can be described as a directed graph $G = (N, A)$, where N is the set of nodes and A is the set of arcs. Each arc $(i, j) \in A$ has a capacity u_{ij} . Each commodity k has a demand d_{ik} at each node $i \in N$ and a cost c_{ijk} on each arc $(i, j) \in A$.

The multi-commodity minimum cost network flow problem can be defined as follows: Given a network $G = (N, A)$, capacities u_{ij} on the arcs, demands d_{ik} and costs c_{ijk} for each commodity k , find the integer flow f_{ijk} on the arcs for each commodity k that minimizes the total cost of the flow, while satisfying the demands of all commodities at the nodes and the capacity constraints on the arcs.

Sets:

- N : Set of nodes in the network.
- A : Set of arcs in the network.
- K : Set of commodities.

Parameters:

- u_{ij} : Capacity of arc $(i, j) \in A$.
- c_{ijk} : Cost per unit of flow of commodity k on arc $(i, j) \in A$.
- d_{ik} : Demand of commodity k at node $i \in N$.

Variables:

- f_{ijk} : Flow of commodity k on arc $(i, j) \in A$.

Model: The multi-commodity minimum cost network flow problem can be formulated as an integer linear programming problem as follows:

$$\begin{aligned} & \text{Minimize} && \sum_{k \in K} \sum_{(i,j) \in A} c_{ijk} f_{ijk} \\ & \text{Subject to} && \sum_{k \in K} \sum_{j:(i,j) \in A} f_{ijk} - \sum_{k \in K} \sum_{j:(j,i) \in A} f_{jik} \leq u_{ij}, \quad \forall (i,j) \in A \\ & && \sum_{j:(i,j) \in A} f_{ijk} - \sum_{j:(j,i) \in A} f_{jik} = d_{ik}, \quad \forall i \in N, k \in K \\ & && f_{ijk} \in \mathbb{Z}^+, \quad \forall (i,j) \in A, k \in K \end{aligned}$$

Objective Function - The objective is to minimize the total cost of the flow.

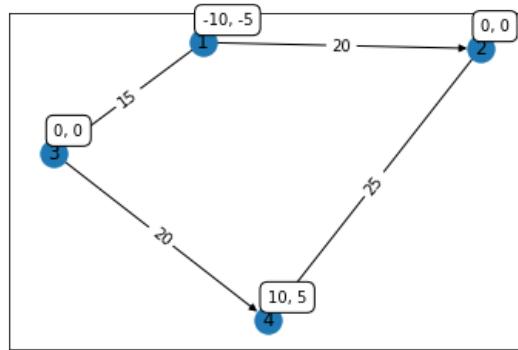
Constraints - Capacity Constraints: The first set of constraints ensures that the total flow on each arc does not exceed its capacity.

- Flow Conservation: The second set of constraints ensures that the flow into each node minus the flow out of the node is equal to the demand of each commodity at the node.
 - Integrality Constraints: The third set of constraints ensures that the flow on each arc for each commodity is an integer.
-
-

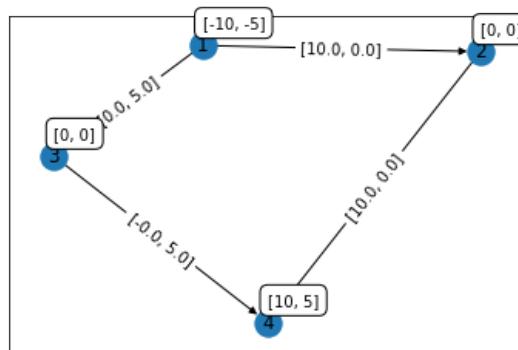
Example: Multi-commodity network flow

Gurobipy

```
# Sample data
nodes = [1, 2, 3, 4]
arcs = [(1, 2), (1, 3), (2, 4), (3, 4)]
commodities = [1, 2]
capacities = {(1, 2): 20, (1, 3): 15, (2, 4): 25, (3, 4): 20}
```



© multi-network-flow-data.png⁷
Figure 3.9: multi-network-flow-data.png



© multi-network-flow-solution.png⁸
Figure 3.10: multi-network-flow-solution.png

```
costs = {(1, 2, 1): 2, (1, 2, 2): 3, (1, 3, 1): 3,
          (1, 3, 2): 2, (2, 4, 1): 1, (2, 4, 2): 2, (3, 4, 1): 2, (3, 4, 2): 1}
demands = {(1, 1): -10, (1, 2): -5, (2, 1): 0, (2, 2): 0, (3, 1): 0, (3, 2): 0,
            (4, 1): 10, (4, 2): 5}
```

Usage

Routing and wavelength assignment (RWA) in optical burst switching of Optical Network would be approached via multi-commodity flow formulas.

⁷multi-network-flow-data.png, from [multi-network-flow-data.png](#). [multi-network-flow-data.png](#), [multi-network-flow-data.png](#).

⁸multi-network-flow-solution.png, from [multi-network-flow-solution.png](#). [multi-network-flow-solution.png](#), [multi-network-flow-solution.png](#).

3.9 Job Shop Scheduling

Example borrowed from: Python MIP example

The Job Shop Scheduling Problem, abbreviated as JSSP, is a classical combinatorial optimization problem that is classified as NP-hard. It is characterized by a collection of jobs that are to be processed on a specific group of machines. Each job has a predetermined processing sequence across the machines, with a fixed processing duration for each machine. A job can only visit a machine once, and machines can only accommodate one job at a time. Once a machine starts a job, it must finish it without any interruptions. The primary goal is to reduce the makespan, which is the total time taken to complete all jobs.

Consider the case of having three jobs and three machines. The jobs have the following processing sequences and times:

- Job j_1 : $m_3(2)$ followed by $m_1(1)$ and then $m_2(2)$.
- Job j_2 : $m_2(1)$ followed by $m_3(2)$ and then $m_1(2)$.
- Job j_3 : $m_3(1)$ followed by $m_2(2)$ and then $m_1(1)$.

Below, we present two potential schedules. The first one is a straightforward approach where jobs are processed sequentially, leading to machines having idle periods. The second approach is optimal, where jobs are executed concurrently.

3.9.1. JSSP Components

The JSSP can be formally described by the following parameters:

- J : Represents the set of jobs, indexed as $\{1, \dots, n\}$.
- I : Represents the set of machines, indexed as $\{1, \dots, m\}$.
- o_r^j : Specifies the machine that handles the r -th operation of job j . The sequence $O^j = (o_1^j, o_2^j, \dots, o_m^j)$ illustrates the processing order for job j .
- p_{ij} : Represents the non-negative processing time of job j on machine i .

The solution to the JSSP must satisfy:

1. Every job j should be processed according to the machine sequence O^j .
2. At any given time, a machine can only process one job.
3. A job, once started on a machine, must run to completion without any breaks.

The objective is to minimize the makespan, or the finishing time of the last job. The problem remains NP-hard even if $n \geq 3$ or $m \geq 3$.

Decision variables include:

- x_{ij} : Starting time of job j on machine i .
- y_{ijk} : A binary variable. It's 1 if job j precedes job k on machine i and 0 otherwise.
- C : The makespan or the maximum completion time.

Example: Job Shop Scheduling Problem Input Data

Basic Parameters The number of jobs, n , is 5 and the number of machines, m , is 3.

Processing Times The matrix of processing times for each job on each machine is given by:

$$\text{times} = \begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 2 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \\ 3 & 4 & 5 \end{bmatrix}$$

Where the element in the i -th row and j -th column represents the time required to process job i on machine j .

Machine Sequences The order in which each job needs to be processed on the machines is described by:

$$\text{machines} = \begin{bmatrix} 2 & 0 & 1 \\ 1 & 2 & 0 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

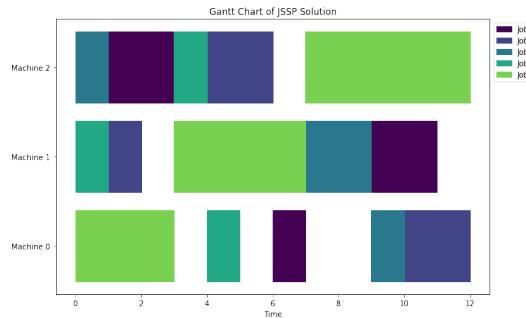
Here, the sequence of machines for each job is given by the rows of the matrix. For instance, job 1 is processed first on machine 2, then on machine 0, and finally on machine 1.

Big M Calculation A large constant, M , is used in the model formulation. It is computed as the sum of all processing times, given by:

$$M = \sum_{i=1}^n \sum_{j=1}^m \text{times}[i][j]$$

Figure 3.11

⁹jssp.png, from jssp.png. jssp.png. jssp.png.

© jssp.png⁹**Figure 3.11: scale = 1.5**

APPLICATIONS OF JSSP The JSSP has numerous practical applications, including:

- Manufacturing: Scheduling tasks in a production line to optimize throughput.
- Computer Science: Allocating tasks to processors in parallel computing environments.
- Healthcare: Scheduling surgeries and other treatments in hospitals.
- Transportation: Optimizing maintenance tasks for vehicles or aircraft in a maintenance facility.

3.9.2. Mathematical Model

Sets:

- J : Represents the set of jobs, indexed as $\{1, \dots, n\}$.
- I : Represents the set of machines, indexed as $\{1, \dots, m\}$.

Parameters:

- o_r^j : Specifies the machine that handles the r -th operation of job j . The sequence $O^j = (o_1^j, o_2^j, \dots, o_m^j)$ illustrates the processing order for job j .
- p_{ij} : Represents the non-negative processing time of job j on machine i .

Variables:

$$x_{ij} = \text{start time of job } j \text{ on machine } i.$$

$$y_{ijk} = \begin{cases} 1, & \text{if job } j \text{ precedes job } k \text{ on machine } i, \\ & i \in I, j, k \in J, j \neq k \\ 0, & \text{otherwise} \end{cases}$$

Model:

$$\min C_{\max}$$

s.t.

Overlapping constraints

$$\begin{aligned} x_{o_r^j} &\geq x_{o_{r-1}^j} + p_{o_{r-1}^j} & \forall r \in \{2, \dots, m\}, j \in J \\ x_{ij} &\geq x_{ik} + p_{ik} - M \cdot y_{ijk} & \forall j, k \in J, j \neq k, i \in I \\ x_{ik} &\geq x_{ij} + p_{ij} - M \cdot (1 - y_{ijk}) & \forall j, k \in J, j \neq k, i \in I \end{aligned}$$

Order tasks for job j
Ordering on machine i
Reverse ordering on machine i

Bounding objective

$$C_{\max} \geq x_{o_m^j} + p_{o_m^j} \quad \forall j \in J \quad \text{last task of job } j \text{ completed before end}$$

Domain of variables

$$\begin{aligned} x_{ij} &\geq 0 & \forall i \in J, i \in I \\ y_{ijk} &\in \{0, 1\} & \forall j, k \in J, i \in I \\ C_{\max} &\geq 0 \end{aligned}$$

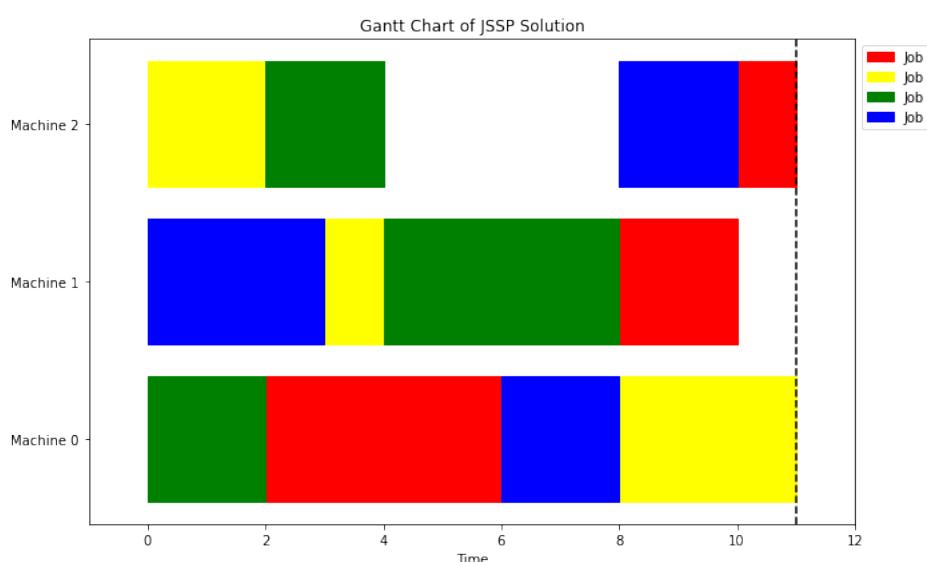
(3.1a)

Example: Duplo Scheduling

Gurobipy

The Duplo in class exercise is solved with Gurobi in the link.

The optimal value is 11.





3.9.3. Job Shop Scheduling Variations

Makespan minimization of assigning jobs to machines.

In this variation, each machine can handle a number of different types of jobs. Some machines can do certain jobs faster than others.

We want to minimize the completion time of all of the jobs.

In this simple variation, each job only needs to be processed on a single machine, but that choice of machine might vary dependent on which machine can do that job faster and which one is available.

Machines:

- Machine 1: Can perform jobs 1, 2, 3, 4, and 5
- Machine 2: Can perform jobs 6, 7, 8, 9, and 10
- Machine 3: Can perform jobs 11, 12, 13, 14, and 15
- Machine 4: Can perform jobs 1, 6, 11, 2, and 7
- Machine 5: Can perform jobs 3, 8, 13, 4, and 9

Jobs:

- Job 1: Machine 1 (processing time = 2 hours), Machine 4 (processing time = 1 hour)
- Job 2: Machine 1 (processing time = 3 hours), Machine 4 (processing time = 2 hours)
- Job 3: Machine 1 (processing time = 2 hours), Machine 5 (processing time = 1 hour)
- Job 4: Machine 1 (processing time = 4 hours), Machine 5 (processing time = 2 hours)

- Job 5: Machine 1 (processing time = 1 hour)
- Job 6: Machine 2 (processing time = 3 hours), Machine 4 (processing time = 2 hours)
- Job 7: Machine 2 (processing time = 2 hours), Machine 4 (processing time = 1 hour)
- Job 8: Machine 2 (processing time = 4 hours), Machine 5 (processing time = 2 hours)
- Job 9: Machine 2 (processing time = 1 hour), Machine 5 (processing time = 1 hour)
- Job 10: Machine 2 (processing time = 2 hours)
- Job 11: Machine 3 (processing time = 3 hours), Machine 4 (processing time = 2 hours)
- Job 12: Machine 3 (processing time = 2 hours), Machine 4 (processing time = 1 hour)
- Job 13: Machine 3 (processing time = 4 hours), Machine 5 (processing time = 2 hours)
- Job 14: Machine 3 (processing time = 1 hour), Machine 5 (processing time = 1 hour)
- Job 15: Machine 3 (processing time = 2 hours)

Objective: Minimize the makespan (i.e. the total time it takes to complete all the jobs)

Constraints:

- Each job can only be performed on the specified machines in the order listed
- A machine can only work on one job at a time
- There are no precedence constraints between jobs.

We leave this as an exercise for the reader to model and solve.

3.10 Quadratic Assignment Problem (QAP)

Resources

- *An applied case of quadratic assignment problem in hospital department layout*
- *See Quadratic Assignment Problem: A survey and Applications.*

The quadratic assignment problem must choose the assignment of n facilities to n locations. Each facility sends some flow to each other facility, and there is a distance to consider between locations. The objective is to minimize to distance times the flow of the assignment.

Example: Hospital Layout On any given day in the hospital, there will be patients that move from various locations in the hospital to various other locations in the hospital. For example, patients move from the operating room to a recovery room, or from the emergency room to the operating room, etc.

We would like to chose the locations of these places in the hospital to minimize the amount of total distance traveled by all the patients.

Quadratic Assignment Problem:

NP-Complete

Given flow f_{ij} connections c_{ij} and fixed building costs f_i , demands d_j and capacities u_i , the capacitated facility location problem is

Sets:

- Let $I = \{1, \dots, n\}$ be the set of facilities.
- Let $K = \{1, \dots, n\}$ be the set of locations.

Parameters:

- f_{ij} - flow from facility i to facility j .
- d_{kl} - distance from location k to location l .
- c_{ik} - cost to setup facility i at location k .

Variables:

- Let

$$x_{ik} = \begin{cases} 1 & \text{if we place facility } i \text{ in location } k, \\ 0 & \text{otherwise.} \end{cases}$$

Model:

$$\begin{aligned} \min & \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n \sum_{l=1}^n f_{ij} d_{kl} x_{ik} x_{jl} + \sum_{i=1}^n \sum_{k=1}^n c_{ik} x_{ik} && \text{(total cost)} \\ \text{s.t.} & \sum_{i=1}^n x_{ik} = 1 \text{ for all } k = 1, \dots, n && \text{(assign facility to location } k) \\ & \sum_{k=1}^n x_{ik} = 1 \text{ for all } i = 1, \dots, n && \text{(assign one location to facility } i) \\ & x_{ik} \in \{0, 1\} \text{ for all } i = 1, \dots, n, \text{ and } k = 1, \dots, n && \text{(binary decisions)} \end{aligned}$$

3.11 Genaralized Assignment Problem (GAP)

https://en.wikipedia.org/wiki/Generalized_assignment_problem In applied mathematics, the maximum **generalized assignment problem** is a problem in combinatorial optimization. This problem is a generalization of the assignment problem in which both tasks and agents have a size. Moreover, the size of each task might vary from one agent to the other.

This problem in its most general form is as follows: There are a number of agents and a number of tasks. Any agent can be assigned to perform any task, incurring some cost and profit that may vary depending on the agent-task assignment. Moreover, each agent has a budget and the sum of the costs of tasks assigned to it cannot exceed this budget. It is required to find an assignment in which all agents do not exceed their budget and total profit of the assignment is maximized.

3.11.1. In special cases

In the special case in which all the agents' budgets and all tasks' costs are equal to 1, this problem reduces to the assignment problem. When the costs and profits of all tasks do not vary between different agents, this problem reduces to the multiple knapsack problem. If there is a single agent, then, this problem reduces to the knapsack problem.

3.11.2. Explanation of definition

In the following, we have n kinds of items, a_1 through a_n and m kinds of bins b_1 through b_m . Each bin b_i is associated with a budget t_i . For a bin b_i , each item a_j has a profit p_{ij} and a weight w_{ij} . A solution is an assignment from items to bins. A feasible solution is a solution in which for each bin b_i the total weight of assigned items is at most t_i . The solution's profit is the sum of profits for each item-bin assignment. The goal is to find a maximum profit feasible solution.

Mathematically the generalized assignment problem can be formulated as an integer program:

$$\text{maximize } \sum_{i=1}^m \sum_{j=1}^n p_{ij}x_{ij}. \quad (3.1)$$

$$\text{subject to } \sum_{j=1}^n w_{ij}x_{ij} \leq t_i \quad i = 1, \dots, m; \quad (3.2)$$

$$\sum_{i=1}^m x_{ij} = 1 \quad j = 1, \dots, n; \quad (3.3)$$

$$x_{ij} \in \{0, 1\} \quad i = 1, \dots, m, \quad j = 1, \dots, n; \quad (3.4)$$

3.12 Other material

3.12.1. Binary reformulation of integer variables

If an integer variable has small upper and lower bounds, it can sometimes be advantageous to recast it as a sequence of binary variables - for either modeling, the solver, or both. Although there are technically many ways to do this, here are the two most common ways.

Full reformulation:

u many binary variables

For a non-negative integer variable x with upper bound u , modeled as

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \quad (3.1)$$

this can be reformulated with u binary variables z_1, \dots, z_u as

$$\begin{aligned} x &= \sum_{i=1}^u i z_i = z_1 + 2z_2 + \dots + uz_u \\ 1 &\geq \sum_{i=1}^u z_i = z_1 + z_2 + \dots + z_u \\ z_i &\in \{0, 1\} \quad \text{for } i = 1, \dots, u \end{aligned} \quad (3.2)$$

We call this the *full reformulation* because there is a binary variable z_i associated with every value i that x could take. That is, if $z_3 = 1$, then the second constraint forces $z_i = 0$ for all $i \neq 3$ (that is, z_3 is the only non-zero binary variable), and hence by the first constraint, $x = 3$.

Binary reformulation:

$O(\log u)$ many binary variables

For a non-negative integer variable x with upper bound u , modeled as

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \quad (3.3)$$

this can be reformulated with u binary variables $z_1, \dots, z_{\lfloor \log(u) \rfloor}$ as

$$\begin{aligned} x &= \sum_{i=0}^{\lfloor \log(u) \rfloor} 2^i z_i = z_0 + 2z_1 + 4z_2 + 8z_3 + \dots + 2^{\lfloor \log(u) \rfloor} z_{\lfloor \log(u) \rfloor} \\ z_i &\in \{0, 1\} \quad \text{for } i = 1, \dots, \lfloor \log(u) \rfloor \end{aligned} \quad (3.4)$$



We call this the *log reformulation* because this requires only logarithmically many binary variables in terms of the upper bound u . This reformulation is particularly better than the full reformulation when the upper bound u is a “larger” number, although we will leave it ambiguous as to how larger a number need to be in order to be described as a “larger” number.

3.13 Literature and Resources

Resources

- The AIMMS modeling has many great examples. It can be book found here:[AIMMS Modeling Book](#).
- [MIT Open Courseware](#)
- For many real world examples, see this book *Case Studies in Operations Research Applications of Optimal Decision Making*, edited by Murty, Katta G. Or find it [here](#).
- [GUROBI modeling examples by GUROBI](#)
- [GUROBI modeling examples by Open Optimization](#) that are linked in this book

Knapsack Problem

- [Video! - Michel Belaire \(EPFL\) teaching knapsack problem](#)

Set Cover

- [Video! - Michel Belaire \(EPFL\) explaining set covering problem](#)
- See [AIMMS - Media Selection](#) for an example of set covering applied to media selection.

Facility Location

- [Wikipedia - Facility Location Problem](#)
- See [GUROBI Modeling Examples - Facility Location](#).

Other examples

- [Sudoku](#)
- [AIMMS - Employee Training](#)
- [AIMMS - Media Selection](#)
- [AIMMS - Diet Problem](#)
- [AIMMS - Farm Planning Problem](#)
- [AIMMS - Pooling Probem](#)
- [INFORMS - Impact](#)
- [INFORMS - Success Story - Bus Routing](#)

Notes from AIMMS modeling book.

- [AIMMS - Practical guidelines for solving difficult MILPs](#)
- [AIMMS - Linear Programming Tricks](#)
- [AIMMS - Formulating Optimization Models](#)
- [AIMMS - Practical guidelines for solving difficult linear programs](#)

Modeling Tricks

- [JuMP tips and tricks](#)
- [Mosek Modeling Cookbook](#)

Further Topics

- [Precedence Constraints](#)

3.14 MIP Solvers and Modeling Tools

- AMPL
- GAMS
- AIMMS
- Python-MIP
- Pyomo
- PuLP
- JuMP
- GUROBI
- CPLEX (IBM)
- Express
- SAS
- Coin-OR (CBC, CLP, IPOPT)
- SCIP

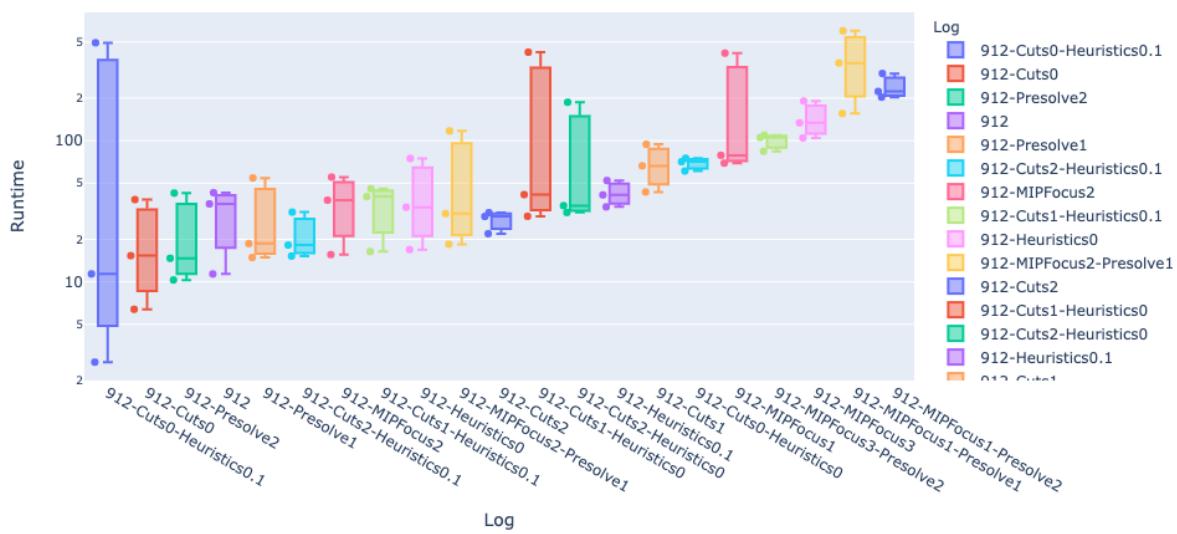
4. Algorithms to Solve Integer Programs

Outcomes

1. Understand misconceptions in difficulty of integer programs
2. Learn basic concepts of algorithms used in solvers
3. Practice these basic concepts at an elementary level
4. Apply these concepts to understanding output from a solver

In this section, we seek to understand some of the fundamental approaches used for solving integer programs. These tools have been developed over the past 70 years. As such, advanced solvers today are incredibly complicated and have many possible settings to hope to solve your problem more efficiently. Unfortunately, there is no single approach that is best for all different problems.

Gurobi Performance



© gurobi_performance¹

Figure 4.1: GUROBI Performance on a set of problems while varying different possible settings.

This plot shows the wild variability of performance of different approaches. Thus, it is very unclear which is the “best” method. Furthermore, this plot can look quite different depending on the problem set one is working with. Although we will not emphasize determining optimal settings in this text, we want to make clear that the techniques used in solvers are quite complicated and are tuned very carefully. We will study some elementary versions of techniques used in these solvers.

Although there are many tricks used to improve the solve time, there are three core elements to solving an integer program: *Presolve*, *Primal techniques*, *Cutting Planes*, and *Branch and Bound*.

PRESOLVE contains many tricks to eliminate variables, reduce the problem size, and make format the problem into something that might be easier to solve. We will not focus on this aspect of solving integer programs.

PRIMAL TECHNIQUES use a variety of approaches to try to find feasible solutions. These feasible solutions are extremely helpful in conjunction with branch and bound.

CUTTING PLANES are ways to improve the description by adding additional inequalities. There are many ways to derive cutting planes. We will learn just a couple to get an idea of how these work.

BRANCH AND BOUND is a method to decompose the problem into smaller subproblems and also to certify optimality (or at least provide a bound to how close to optimal a solution is) by removing sets of subproblems that can be argued to be suboptimal. We will look at an elementary branch and bound approach. Understanding this technique is key to explaining the output of an integer programming solver.

We will begin this chapter with a comparison of solving the linear programming relaxation compared to solving an integer program. We will then use this understanding as fundamental to both the techniques of cutting planes and branch and bound. We will end this section with an example of output from GUROBI and explain how to interpret this information.

4.1 Foundational Principle - LP is a relaxation for IP

Outcomes

This section describes the foundational principle about relaxations that will be used in our main algorithmic techniques.

Recall that the linear relaxation of an integer program is the linear programming problem after removing the integrality constraints

Integer Program:

$$\begin{aligned} \max \quad & z_{IP} = c^\top x \\ & Ax \leq b \\ & x \in \mathbb{Z}^n \end{aligned}$$

Linear Relaxation:

$$\begin{aligned} \max \quad & z_{LP} = c^\top x \\ & Ax \leq b \\ & x \in \mathbb{R}^n \end{aligned}$$

¹gurobi_performance, from gurobi_performance. gurobi_performance, gurobi_performance.

Theorem 4.1: LP Bounds

It always holds that

$$z_{IP}^* \leq z_{LP}^*. \quad (4.1)$$

Furthermore, if x_{LP}^* is integral (feasible for the integer program), then

$$x_{LP}^* = x_{IP}^* \text{ and } z_{LP}^* = z_{IP}^*. \quad (4.2)$$

Example 4.2

Consider the problem

$$\begin{aligned} \max z &= 3x_1 + 2x_2 \\ 2x_1 + x_2 &\leq 6 \\ x_1, x_2 &\geq 0; x_1, x_2 \text{ integer} \end{aligned}$$

4.1.1. Rounding LP Solution can be bad!

Consider the two variable knapsack problem

$$\max 3x_1 + 100x_2 \quad (4.3)$$

$$x_1 + 100x_2 \leq 100 \quad (4.4)$$

$$x_i \in \{0, 1\} \text{ for } i = 1, 2. \quad (4.5)$$

Then $x_{LP}^* = (1, 0.99)$ and $z_{LP}^* = 1 \cdot 3 + 0.99 \cdot 100 = 3 + 99 = 102$.

But $x_{IP}^* = (0, 1)$ with $z_{IP}^* = 0 \cdot 3 + 1 \cdot 100 = 100$.

Suppose that we rounded the LP solution.

$x_{LP-Rounded-Down}^* = (1, 0)$. Then $z_{LP-Rounded-Down}^* = 1 \cdot 3 = 3$. Which is a terrible solution!

How can we avoid this issue? We will use two main techniques - (1) decomposing the problem via *Branch and Bound* and (2) improve the LP relaxation via cutting planes.

4.1.2. Rounding LP solution can be infeasible!

Now only could it produce a poor solution, it is not always clear how to round to a feasible solution. Consider the two variable knapsack problem

$$\min x_1 + 200x_2 \quad (4.6)$$

$$x_1 + 100x_2 \geq 100 \quad (4.7)$$

$$x_i \in \{0, 1\} \text{ for } i = 1, 2. \quad (4.8)$$

Again, we return $x_{LP}^* = (1, 0.99)$. But in this case, rounding down to $(1, 0)$ is not feasible!

In n -dimensions, there are potentially 2^n integer points to round to, making rounding very complicated in some cases.

4.2 Branch and Bound

4.2.1. Binary Integer Programming

Our goal is to decompose the problem until one of three things happens:

1. We find an **integer** point in a subproblem,
2. We prove that a sub problem is **suboptimal**,
3. We prove that a subproblem is **infeasible**.

Throughout the decomposition process, we will store Lower Bounds. These bounds are created by finding feasible integer solutions. Using these lower bounds, we can prove that other branches are suboptimal.

Here is the full algorithm. We will look at an example next.

Algorithm 1 Branch and Bound for Binary Integer Programming

Require: Binary Integer Linear Problem with max objective

Ensure: Exact Optimal Solution x^*

- 1: Set $LB = -\infty$.
 - 2: Solve LP relaxation.
 - 3: **if** x^* is binary integer **then**
 - 4: Stop!
 - 5: **else**
 - 6: Choose fractional entry x_i^* .
 - 7: Branch onto subproblems:
 - (i) $x_i = 0$
 - (ii) $x_i = 1$
 - 8: Solve LP relaxation of any subproblem.
 - 9: **if** LP relaxation is infeasible **then**
 - 10: Prune this node as "**Infeasible**"
 - 11: **else if** $z^* < LB$ **then**
 - 12: Prune this node as "**Suboptimal**"
 - 13: **else if** x^* is binary integer **then**
 - 14: Prune this node as "**Binary Integer**"
 - 15: Update $LB = \max(LB, z^*)$.
 - 16: **else**
 - 17: Choose fractional entry x_i^* .
 - 18: Branch onto subproblems:
 - (i) $x_i = 0$
 - (ii) $x_i = 1$
 - 19: Return to step 2 until all subproblems are pruned.
 - 20: Return best binary integer solution found.
-

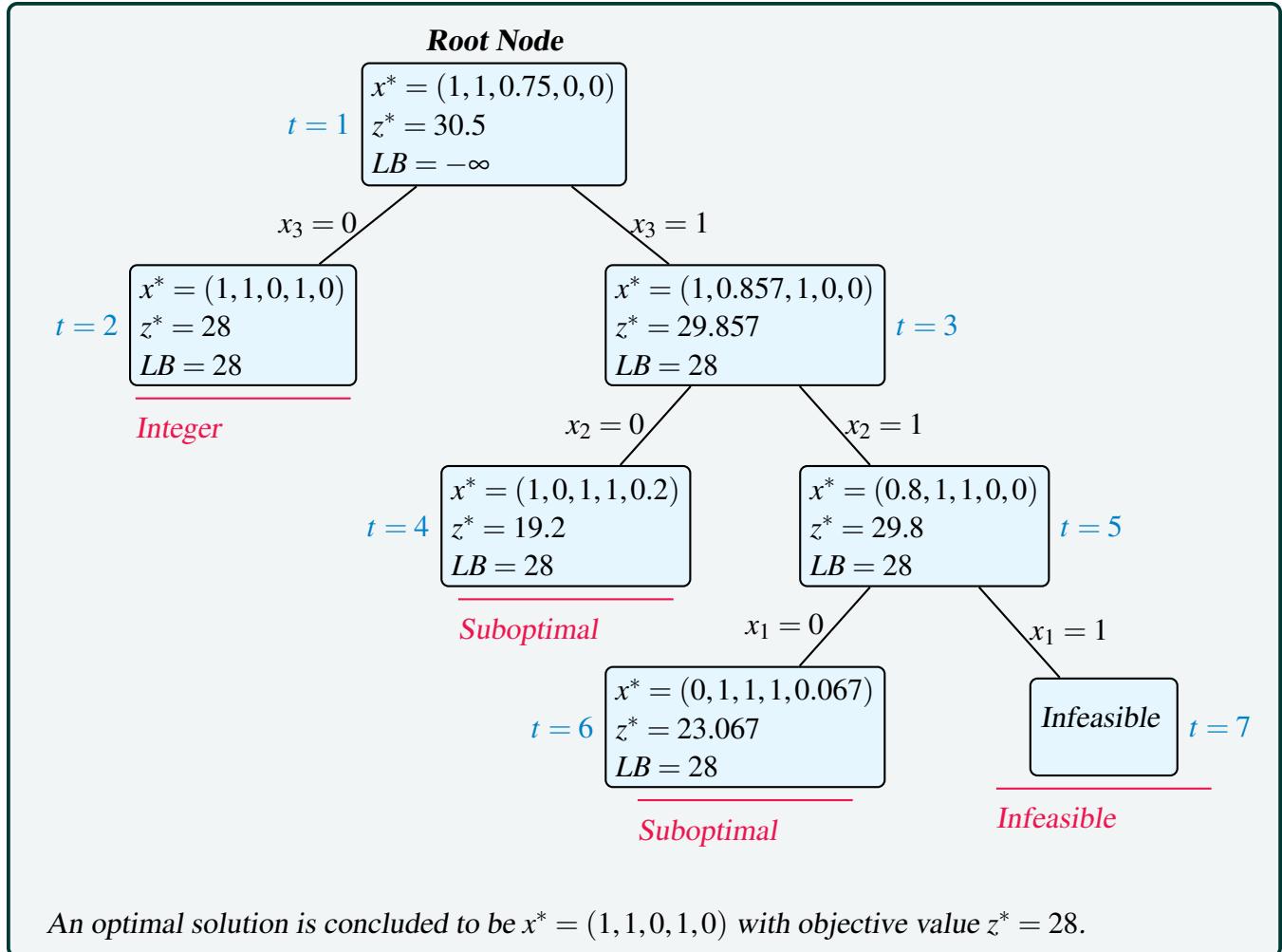
Let's look at a nice example. This is the smallest example that exhibits all the interesting behavior of branch and bound. Typically these branch and bound trees are much much much bigger.

We will use the result from Subsection 2.4.3 that demonstrates that in the single inequality case, the solution to the LP can be easy to write down.

Example 4.3: Binary Knapsack Example

Solve the following problem with branch and bound.

$$\begin{aligned} \max \quad & z = 11x_1 + 15x_2 + 6x_3 + 2x_4 + x_5 \\ \text{Subject to: } & 5x_1 + 7x_2 + 4x_3 + 3x_4 + 15x_5 \leq 15 \\ & x_i \text{ binary}, i = 1, \dots, 5 \end{aligned}$$



4.2.2. Branch and bound on general integer variables

We can apply this concept of branch and bound thinking about general integer variables. Instead of branching to specific values, we restrict to upper and lower bounds on a variable. This cuts the feasible region of a problem (or sub problem) into two smaller regions. All of the same concepts apply here.

Algorithm 2 Branch and Bound - Maximization

Require: Integer Linear Problem with max objective**Ensure:** Exact Optimal Solution x^*

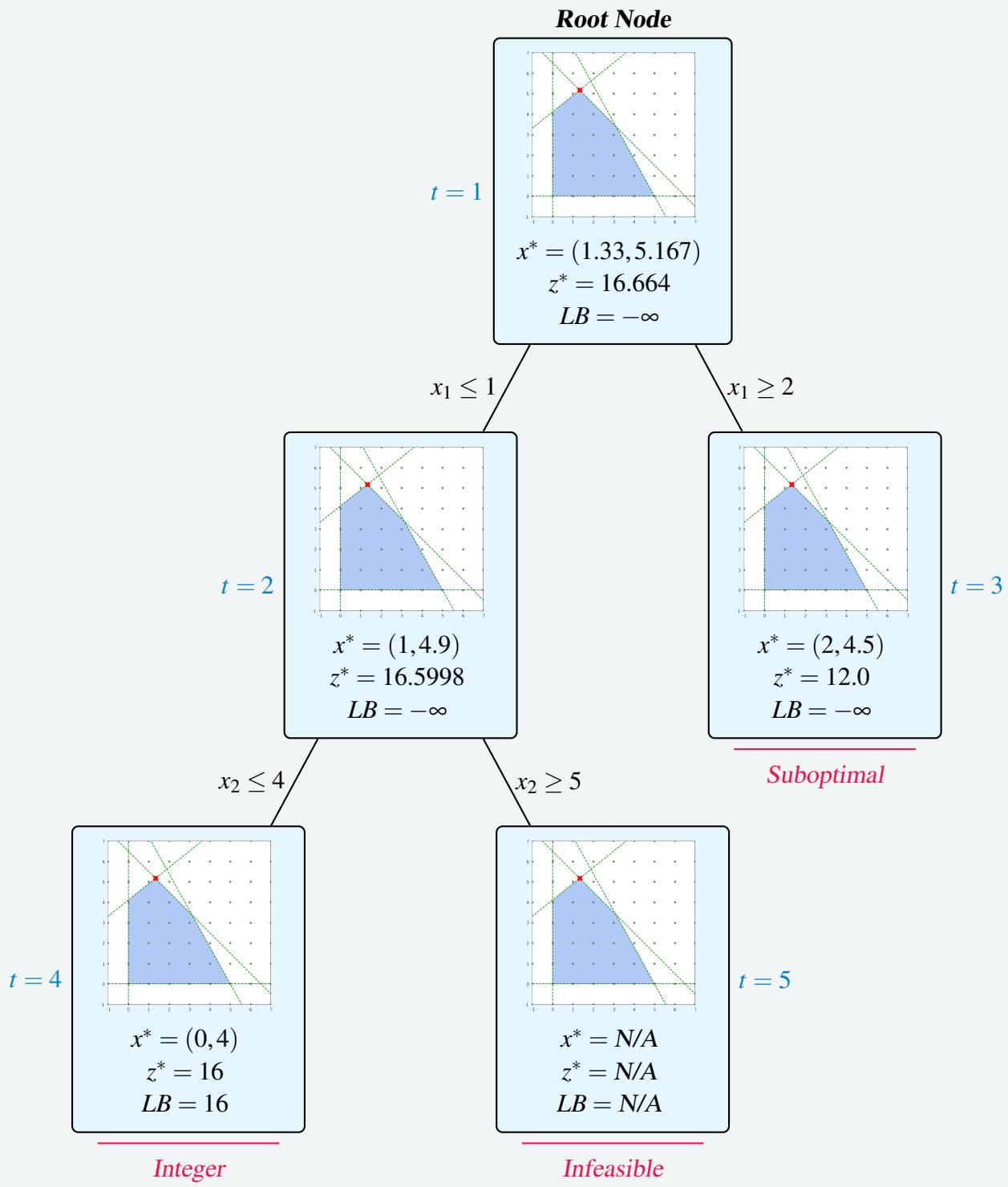
- 1: Set $LB = -\infty$.
 - 2: Solve LP relaxation.
 - 3: **if** x^* is integer **then**
 - 4: Stop!
 - 5: **else**
 - 6: Choose fractional entry x_i^* .
 - 7: Branch onto subproblems:
 - (i) $x_i \leq \lfloor x_i^* \rfloor$
 - (ii) $x_i \geq \lceil x_i^* \rceil$
 - 8: Solve LP relaxation of any subproblem.
 - 9: **if** LP relaxation is infeasible **then**
 - 10: Prune this node as "**Infeasible**"
 - 11: **else if** $z^* < LB$ **then**
 - 12: Prune this node as "**Suboptimal**"
 - 13: **else if** x^* is integer **then**
 - 14: Prune this node as "**Integer**"
 - 15: Update $LB = \max(LB, z^*)$.
 - 16: **else**
 - 17: Choose fractional entry x_i^* .
 - 18: Branch onto subproblems:
 - (i) $x_i \leq \lfloor x_i^* \rfloor$
 - (ii) $x_i \geq \lceil x_i^* \rceil$
 - 19: Return to step 2 until all subproblems are pruned.
 - 20: Return best integer solution found.
-

Here is an example of branching on general integer variables.

Example 4.4: Branching on general integer variables

Consider the two variable example with

$$\begin{aligned}
 & \max -3x_1 + 4x_2 \\
 & 2x_1 + 2x_2 \leq 13 \\
 & -8x_1 + 10x_2 \leq 41 \\
 & 9x_1 + 5x_2 \leq 45 \\
 & 0 \leq x_1 \leq 10, \text{ integer} \\
 & 0 \leq x_2 \leq 10, \text{ integer}
 \end{aligned}$$



An optimal solution is concluded to be $x^* = (0, 4)$ with objective value $z^* = 16$.

4.3 Cutting Planes

Cutting planes are additional inequalities that are valid for the feasible integer solutions that cut off part of the LP relaxation. Cutting planes can create a tighter description of the feasible region that allows for the optimal solution to be obtained by simply solving a strengthened linear relaxation.

We begin with an example of problem specific inequalities that apply only to certain structured constraints.

4.3.1. Cover Inequalities

Consider the binary knapsack problem

$$\begin{aligned} & \max x_1 + 2x_2 + x_3 + 7x_4 \\ \text{s.t. } & 100x_1 + 70x_2 + 50x_3 + 60x_4 \leq 150 \\ & x_i \text{ binary for } i = 1, \dots, 4 \end{aligned}$$

A *cover* S is any subset of the variables whose sum of weights exceed the capacity of the right hand side of the inequality.

For example, $S = \{1, 2, 3, 4\}$ is a cover since $100 + 70 + 50 + 60 > 150$.

Since not all variables in the cover S can be in the knapsack simultaneously, we can enforce the *cover inequality*

$$\sum_{i \in S} x_i \leq |S| - 1 \Rightarrow x_1 + x_2 + x_3 + x_4 \leq 4 - 1 = 3. \quad (4.1)$$

Note, however, that there are other covers that use fewer variables.

A *minimal cover* is a subset of variables such that no other subset of those variables is also a cover. For example, consider the cover $S' = \{1, 2\}$. This is a cover since $100 + 70 > 150$. Since S' is a subset of S , the cover S is not a minimal cover. In fact, S' is a minimal cover since there are no smaller subsets of the set S' that also produce a cover. In this case, we call the corresponding inequality a *minimal cover inequality*. That is, the inequality

$$x_1 + x_2 \leq 2 - 1 = 1 \quad (4.2)$$

is a minimal cover inequality for this problem. The minimal cover inequalities are the "strongest" of all cover inequalities.

Find the two other minimal covers (one of size 2 and one of size 3) and write their corresponding minimal cover inequalities.

Solution. The other minimal covers are

$$S = \{1, 4\} \Rightarrow x_1 + x_4 \leq 1 \quad (4.3)$$

and

$$S = \{2, 3, 4\} \Rightarrow x_2 + x_3 + x_4 \leq 2 \quad (4.4)$$



4.3.2. Chvátal Cuts

Chvátal Cuts are a general technique to produce new inequalities that are valid for feasible integer points.

Chvátal Cuts:

Suppose

$$a_1x_1 + \cdots + a_nx_n \leq d \quad (4.5)$$

is a valid inequality for the polyhedron $P = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$, then

$$\lfloor a_1 \rfloor x_1 + \cdots + \lfloor a_n \rfloor x_n \leq \lfloor d \rfloor \quad (4.6)$$

is valid for the integer points in P , that is, it is valid for the set $P \cap \mathbb{Z}^n$. Equation (4.6) is called a Chvátal Cut.

We will illustrate this idea with an example.

Example 4.5

Recall example 2. The model was

Model

$$\begin{array}{lll} \min & p + n + d + q & \text{total number of coins used} \\ \text{s.t.} & p + 5n + 10d + 25q = 83 & \text{sums to } 83\text{¢} \\ & p, d, n, q \in \mathbb{Z}_+ & \text{each is a non-negative integer} \end{array}$$

From the equality constraint we can derive several inequalities.

1. Divide by 25 and round down both sides:

$$\frac{p + 5n + 10d + 25q}{25} = 83/25 \Rightarrow q \leq 3$$

2. Divide by 10 and round down both sides:

$$\frac{p + 5n + 10d + 25q}{10} = 83/10 \Rightarrow d + 2q \leq 8$$

3. Divide by 5 and round down both sides:

$$\frac{p + 5n + 10d + 25q}{10} = 83/5 \Rightarrow n + 2d + 5q \leq 16$$

4. Multiply by 0.12 and round down both sides:

$$0.12(p + 5n + 10d + 25q) = 0.12(83) \Rightarrow d + 3q \leq 9$$

These new inequalities are all valid for the integer solutions. Consider the new model:

New Model

$$\begin{array}{ll}
 \min & p + n + d + q & \text{total number of coins used} \\
 \text{s.t.} & p + 5n + 10d + 25q = 83 & \text{sums to } 83\text{¢} \\
 & q \leq 3 \\
 & d + 2q \leq 8 \\
 & n + 2d + 5q \leq 16 \\
 & d + 3q \leq 9 \\
 & p, d, n, q \in \mathbb{Z}_+ & \text{each is a non-negative integer}
 \end{array}$$

The solution to the LP relaxation is exactly $q = 3, d = 0, n = 1, p = 3$, which is an integral feasible solution, and hence it is an optimal solution.

4.3.3. Cutting Planes Procedure

Cutting planes can be constructed dynamically so that we only look for cuts that are critical to proving optimality of the problem.

This is done by iteratively solving a relaxation, adding a cut to improve the relaxation, and then resolving the relaxation.

Here is the basic algorithm.

Algorithm 3 (Pure) Cutting Plane Procedure

- 1: **Input:** Current LP relaxation.
 - 2: **Output:** Integral solution (if exists).
 - 3: **procedure** CUTTINGPLANE
 - 4: **while** True **do**
 - 5: Solve the current LP relaxation.
 - 6: **if** solution is integral **then**
 - 7: **return** that solution.
 - 8: **STOP**
 - 9: Add a cutting plane (or many cutting planes) that cut off the LP-optimal solution.
-

Refer to the cutting plane procedure in Figure 3.

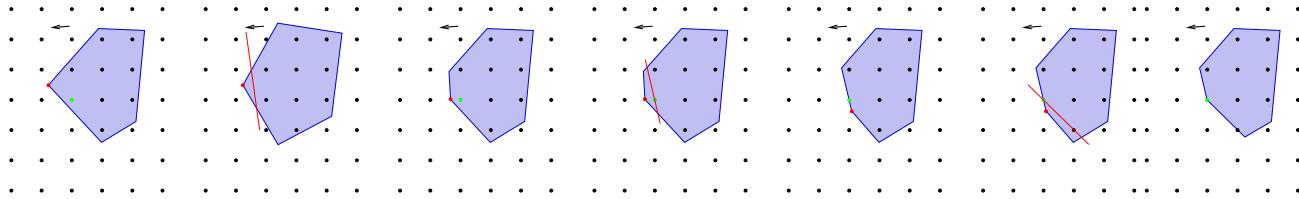


Figure 4.2: The cutting plane procedure.

In practice, this procedure is integrated in some with branch and bound and also other primal heuristics.

Here is a concrete example of how this could play out.

Model	LP Solution
$\max x_1 + x_2$ subject to $-2x_1 + x_2 \leq 0.5$ $x_1 + 2x_2 \leq 10.5$ $x_1 - x_2 \leq 0.5$ $-2x_1 - x_2 \leq -2$	<p>© cutting-plane-1-picture²</p>
$\max x_1 + x_2$ subject to $-2x_1 + x_2 \leq 0.5$ $x_1 + 2x_2 \leq 10.5$ $x_1 - x_2 \leq 0.5$ $-2x_1 - x_2 \leq -2$ $x_1 \leq 3$	<p>© cutting-plane-2-picture³</p>
$\max x_1 + x_2$ subject to $-2x_1 + x_2 \leq 0.5$ $x_1 + 2x_2 \leq 10.5$ $x_1 - x_2 \leq 0.5$ $-2x_1 - x_2 \leq -2$ $x_1 \leq 3$ $x_1 + x_2 \leq 6$	<p>© cutting-plane-3-picture⁴</p>

4.3.4. Gomory Cuts

Gomory cuts are a type of Chvátal cut that is derived from the simplex tableau and can be generated dynamically, and thus is a viable candidate to implement the cutting plane procedure.

Specifically, suppose that

$$x_i + \sum_{i \in N} \tilde{a}_i x_i = \tilde{b}_i \quad (4.7)$$

is an equation in the optimal simplex tableau.

Gomory Cut:

The Gomory cut corresponding to the tableau row (4.7) is

$$\sum_{i \in N} (\tilde{a}_i - \lfloor \tilde{a}_i \rfloor) x_i \geq \tilde{b}_i - \lfloor \tilde{b}_i \rfloor \quad (4.8)$$

We will solve the following problem using only Gomory Cuts.

$$\begin{aligned} \min \quad & x_1 - 2x_2 \\ \text{s.t.} \quad & -4x_1 + 6x_2 \leq 9 \\ & x_1 + x_2 \leq 4 \\ & x \geq 0, \quad x_1, x_2 \in \mathbb{Z} \end{aligned}$$

Step 1: The first thing to do is to put this into standard form by appending slack variables.

$$\begin{aligned} \min \quad & x_1 - 2x_2 \\ \text{s.t.} \quad & -4x_1 + 6x_2 + s_1 = 9 \\ & x_1 + x_2 + s_2 = 4 \\ & x \geq 0, \quad x_1, x_2 \in \mathbb{Z} \end{aligned} \quad (4.9)$$

We can apply the simplex method to solve the LP relaxation.

	Basis	RHS	x_1	x_2	s_1	s_2
Initial Basis	z	0.0	1.0	-2.0	0.0	0.0
	s_1	9.0	-4.0	6.0	1.0	0.0
	s_2	4.0	1.0	1.0	0.0	1.0
:			:			
Optimal Basis	Basis	RHS	x_1	x_2	s_1	s_2
	z	-3.5	0.0	0.0	0.3	0.2
	x_1	1.5	1.0	0.0	-0.1	0.6
	x_2	2.5	0.0	1.0	0.1	0.4

This LP relaxation produces the fractional basic solution $x_{LP} = (1.5, 2.5)$.

Example 4.6

(Gomory cut removes LP solution) We now identify an integer variable x_i that has a fractional basic solution. Since both variables have fractional values, we can choose either row to make a cut. Let's focus on the row corresponding to x_1 .

The row from the tableau expresses the equation

$$x_1 - 0.1s_1 + -0.6s_2 = 1.5. \quad (4.10)$$

Applying the Gomory Cut (4.8), we have the inequality

$$0.9s_1 + 0.4s_2 \geq 0.5. \quad (4.11)$$

The current LP solution is $(x_{LP}, s_{LP}) = (1.5, 2.5, 0, 0)$. Trivially, since $s_1, s_2 = 0$, the inequality is violated.

Example 4.7: (Gomory Cut in Original Space)

The Gomory Cut (4.11) can be rewritten in the original variables using the equations from (4.9). That is, we can use the equations

$$\begin{aligned} s_1 &= 9 + 4x_1 - 6x_2 \\ s_2 &= 4 - x_1 - x_2, \end{aligned} \quad (4.12)$$

which transforms the Gomory cut into the original variables to create the inequality

$$0.9(9 + 4x_1 - 6x_2) + 0.4(4 - x_1 - x_2) \geq 0.5.$$

or equivalently

$$-3.2x_1 + 5.8x_2 \leq 9.2. \quad (4.13)$$

As you can see, this inequality does cut off the current LP relaxation.

Example 4.8: (Gomory cuts plus new tableau)

Now we add the slack variable $s_3 \geq 0$ to make the equation

$$0.9s_1 + 0.4s_2 - s_3 = 0.5. \quad (4.14)$$

Next, we need to solve the new linear programming relaxation and continue the cutting plane procedure.

4.4 Interpreting Output Information and Progress

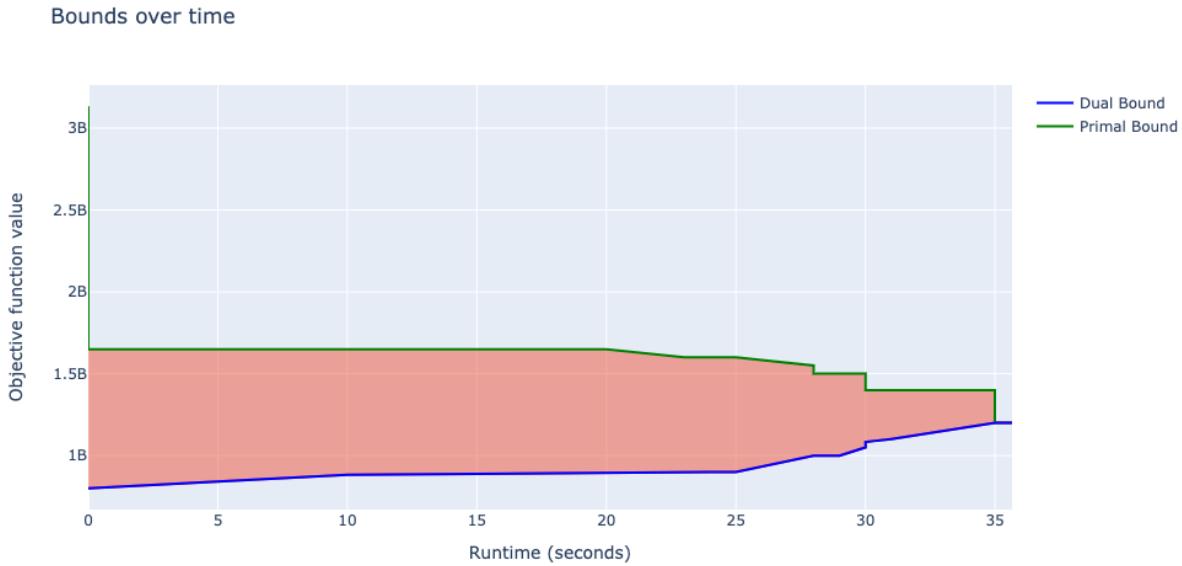


Figure 4.3: This shows the progress of the solver over time.

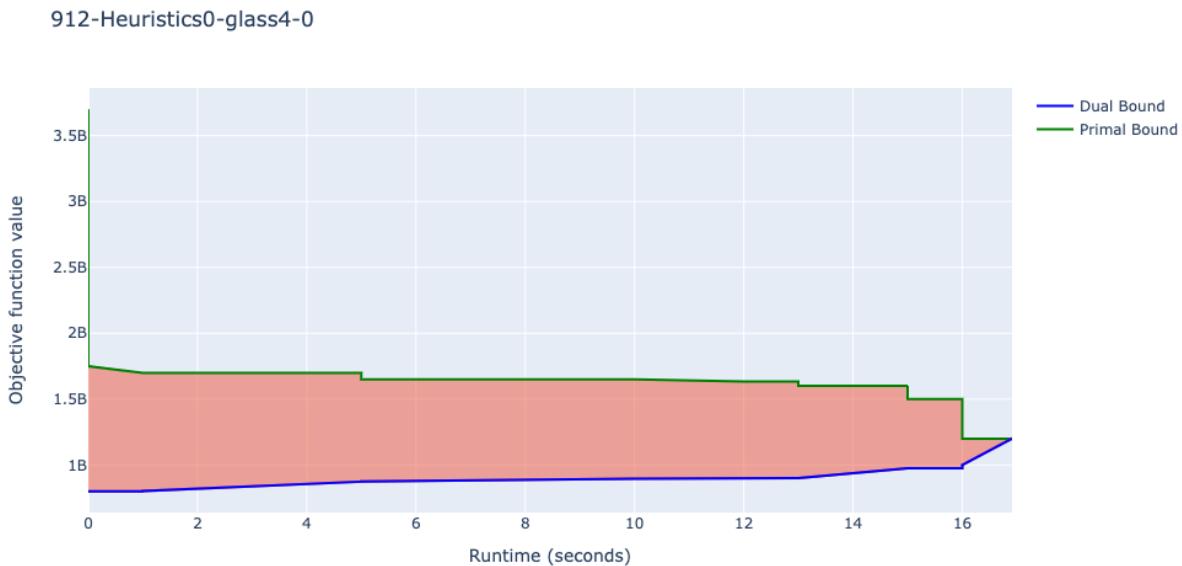


Figure 4.4: This shows the progress of the solver over time.

⁵solve_progress1.png, from solve_progress1.png. solve_progress1.png, solve_progress1.png.

⁶solve_progress2.png, from solve_progress2.png. solve_progress2.png, solve_progress2.png.

4.5 Branching Decision Rules in Integer Programming

Outcomes

A thorough study of branching decision rules is not necessary at this point, but we add some high level discussion here to demonstrate that this is an important part of solving integer programs efficiently.

Integer programming (IP) is a mathematical optimization technique where some or all of the variables are restricted to take integer values. One of the most popular methods for solving integer programming problems is the branch-and-bound method. Central to the branch-and-bound method is the concept of branching, where the solution space is recursively divided into smaller subproblems. The decision on how and where to branch is governed by branching decision rules.

IMPORTANCE OF BRANCHING RULES The efficiency of the branch-and-bound method heavily depends on the choice of branching rules. A good branching rule can significantly reduce the number of subproblems that need to be explored, leading to faster solution times. Conversely, a poor branching rule can lead to an exponential increase in the number of subproblems, making the problem intractable.

COMMON BRANCHING RULES There are several branching rules that have been proposed in the literature. Some of the most common ones include:

- **Most Fractional Variable Rule:** This rule selects the variable that has the fractional part closest to 0.5 for branching. The rationale is that this variable is the most uncertain in terms of its integer value.
- **Least Fractional Variable Rule:** Contrary to the most fractional variable rule, this rule selects the variable with the fractional part furthest from 0.5.
- **Strong Branching:** This rule evaluates the impact of branching on a few variables and selects the one that seems most promising in terms of reducing the objective function value.
- **Pseudocost Branching:** Pseudocosts estimate the impact of branching on a particular variable based on historical data from previous branchings on that variable. For each variable, two pseudocosts are maintained: one for the upward branch (rounding up) and one for the downward branch (rounding down). The variable with the highest expected improvement in the objective function, based on its pseudocosts, is selected for branching.

PROBLEM SPECIFIC BRANCHING Reformulating integer programming problems can reveal hidden structures beneficial for branching. For instance, transforming a problem or linearizing constraints might introduce new variables that offer insightful branching points. These auxiliary variables, emerging from reformulations like the cutting plane method, often encapsulate the problem's combinatorial essence. Branching on them can lead to tighter bounds or more effective pruning of infeasible regions.

While traditional branching is binary, considering multi-way branching—where a node splits into more than two branches—can be advantageous for problems where variables have multiple discrete values. This approach allows for simultaneous exploration of several potential solution paths, enhancing the efficiency of the search process.

In essence, delving into a problem's specific structure and reformulations can pave the way for innovative and more efficient branching strategies.

ADVANCED BRANCHING TECHNIQUES WITH MACHINE LEARNING Branching algorithms in commercial solvers are highly tuned for performance. They often incorporate advanced strategies, including periodic restarts, to improve the branching tree's structure and efficiency. In recent years, machine learning techniques have been employed and studied to devise better branching rules. These techniques can be tailored for specific problem classes or designed for general use cases. Machine learning can be applied in an offline setting, where the algorithm learns from a set of instances and then applies the knowledge to new instances, or in an online fashion, where the algorithm refines its branching rules during the branching process. Furthermore, machine learning has been utilized to estimate the overall size of the branch-and-bound tree required to reach optimality. Notably, such techniques have been implemented in the open-source solver SCIP.

The choice of branching rule is crucial in determining the efficiency of the branch-and-bound algorithm for integer programming. With the advent of machine learning and continuous advancements in optimization techniques, the landscape of branching strategies is rapidly evolving, offering promising avenues for even more efficient integer programming solutions.

There is a few clever ideas out there on how to choose which variables to branch on. We will not go into this here. But for the interested reader, look into

- Strong Branching
- Pseudo-cost Branching

4.6 Decomposition Methods - Advanced approaches to integer programming

Outcomes

A thorough study of the decomposition approaches discussed in this section are not necessary at this point, but we add some high level discussion here regarding the techniques to give the reader a sense of what approaches can be taken when the models chosen are not solving fast enough.

We will discuss briefly three key decomposition approaches: Lagrangian Relaxation, Dantzig-Wolfe Reformulation (with column generation and branch and price), and Benders Decomposition.

We will give more attention to these approaches in a much later part of this book on Advanced Integer Programming Techniques.

4.6.1. Lagrangian Relaxation

Lagrangian relaxation is a powerful technique in integer programming designed to simplify complex problems by moving certain complicating constraints into the objective function. The essence of this method lies in "relaxing" these constraints, allowing them to be violated at a cost. This is achieved by associating each relaxed constraint with a Lagrange multiplier, which then penalizes the objective function based on the degree of violation.

The primary advantage of Lagrangian relaxation is the creation of a strong relaxation for the problem. By transforming the original problem into a more tractable one, solutions can be obtained more efficiently, often providing good bounds for the original problem. Moreover, the dual values of the Lagrange multipliers can offer insights into the relative importance of the relaxed constraints, guiding further refinements or heuristic approaches.

Within a branch-and-bound framework, Lagrangian relaxation can be particularly effective. At each node of the branching tree, the Lagrangian relaxation can be solved to obtain bounds for the subproblem. If the bound indicates that the node cannot lead to a better solution than the current best-known one, the node can be pruned, reducing the overall search space. Additionally, solutions to the relaxed problem can guide branching decisions or serve as starting solutions for heuristics, accelerating the overall solution process.

In summary, Lagrangian relaxation offers a strategic approach to handle complicating constraints in integer programming, facilitating the generation of strong relaxations and enhancing the efficiency of algorithms like branch-and-bound.

See [Fisher2004] ([link](#)) for a tutorial on Lagrangian Relaxations.

4.6.2. Dantzig-Wolfe Reformulation and Column Generation

Dantzig-Wolfe reformulation is a technique in mathematical optimization that decomposes a problem into subproblems by exploiting its block structure. Specifically, it's applied to problems where the constraints can be divided into groups that interact only through certain linking constraints. The reformulation expresses the original variables as combinations of new variables, often referred to as "columns," which represent feasible solutions to the subproblems.

The primary advantage of Dantzig-Wolfe reformulation is that it transforms the original problem into a master problem with a potentially much-reduced set of variables (columns). However, the challenge is that the number of potential columns can be exponentially large, making it impractical to enumerate them all. This is where column generation comes into play.

COLUMN GENERATION is an iterative method to handle the large set of potential columns. Instead of considering all columns simultaneously, the algorithm starts with a subset of promising columns. The master problem is solved with this subset, and then a pricing subproblem is tackled to identify if there are any columns (feasible solutions to the subproblems) that can improve the master problem's objective. If such columns are found, they are added to the master problem, and the process repeats. This "generate-and-solve" approach continues until no more beneficial columns can be identified.

BRANCH-AND-PRICE is an extension of the branch-and-bound method that incorporates column generation. In a branch-and-price algorithm, the branching process is applied to the master problem, but at each node of the branching tree, column generation is used to dynamically generate the necessary columns. This approach combines the power of Dantzig-Wolfe reformulation and column generation with the systematic search of branch-and-bound, allowing for the efficient solution of large-scale integer programming problems with a decomposable structure.

In summary, Dantzig-Wolfe reformulation provides a foundation for decomposing complex optimization problems. When combined with column generation and the branch-and-price framework, it offers a potent toolkit for tackling large-scale, structured integer programming challenges.

Dantzig-Wolfe reformulation, combined with column generation and branch-and-price techniques, has been instrumental in solving a variety of large-scale combinatorial optimization problems. Some of the key applications include:

1. **Cutting Stock Problem:** Perhaps one of the most classic applications, the cutting stock problem involves determining the optimal way to cut raw materials (like rolls of paper or metal beams) to meet specific demand requirements while minimizing waste. The columns in this context represent different cutting patterns.
2. **Vehicle Routing Problem (VRP):** In the VRP, a fleet of vehicles must deliver goods to a set of customers in the most efficient manner. The challenge is to determine routes for each vehicle such that all customers are served, and the total routing cost is minimized. Columns can represent feasible routes or segments of routes for the vehicles.

3. **Crew Scheduling:** Airlines and railways face the challenge of assigning crew members to flights or trains while adhering to regulations and minimizing costs. In this context, columns can represent feasible schedules or rotations for the crew members.
4. **Set Partitioning and Covering Problems:** These problems arise in various applications, such as airline crew scheduling, school timetabling, and frequency assignment in telecommunications. Columns represent subsets of items or tasks that can be grouped or covered together.
5. **Capacitated Arc Routing Problem:** Similar to the VRP, but the focus is on serving demands located on edges (or arcs) of a network, like snow plowing or garbage collection on street networks. Columns can represent feasible routes on the network.
6. **Bin Packing:** This problem involves packing items of different volumes into a finite number of bins in a way that minimizes the number of bins used. Columns represent feasible packing patterns for the bins.

These applications underscore the versatility and power of Dantzig-Wolfe reformulation and associated techniques. By decomposing complex problems into more manageable subproblems, these methods provide a structured and efficient approach to tackle large-scale, real-world optimization challenges.

4.6.3. Benders Decomposition

Benders' decomposition is a mathematical optimization technique designed to tackle problems with a specific structure, where decision variables can be partitioned into two sets, often referred to as the "master" and "subproblem" variables. The primary idea behind Benders' decomposition is to solve the problem iteratively, addressing the master problem and subproblem separately, and then linking them through optimality and feasibility cuts.

The method begins by solving a simplified version of the master problem without considering the complicating constraints associated with the subproblem. Once a solution is obtained, it's used to generate and solve the subproblem. The outcome of the subproblem then informs whether the current master solution is feasible or not. If infeasible, a feasibility cut is added to the master problem. If feasible but suboptimal, an optimality cut is introduced. This iterative process continues until convergence is achieved.

One of the most notable applications of Benders' decomposition is in Stochastic Programming. Stochastic Programming deals with optimization problems where some parameters are uncertain and described by probability distributions. In such problems, the decision-making process is often split into two stages: "here-and-now" decisions made before the realization of uncertainty and "wait-and-see" decisions made after. Benders' decomposition is particularly useful here, as the master problem can handle the "here-and-now" decisions, while the subproblem manages the "wait-and-see" decisions for various scenarios of uncertainty. This decomposition allows for efficient exploration of the solution space and provides a structured way to handle the complexities introduced by uncertainty.

In summary, Benders' decomposition offers a strategic approach to decompose and solve complex optimization problems, with its application in Stochastic Programming standing out as a testament to its efficacy in handling uncertainty and multi-stage decision-making.

4.7 Literature and Resources

Resources

LP Rounding

- *Video! - Michel Belaire (EPFL) looking at rounding the LP solution to an IP solution*

Fractional Knapsack problem

- *Video solving the Fractional Knapsack Problem*
- *Blog solving the Fractional Knapsack Problem*

Branch and Bound

- *Video! - Michel Belaire (EPFL) Teaching Branch and Bound Theory*
- *Video! - Michel Belaire (EPFL) Teaching Branch and Bound with Example*
- See [Module by Miguel Casquilho](#) for some nice notes on branch and bound.

Gomory Cuts

- *Pascal Van Hyndryk (Georgia Tech) Teaching Gomory Cuts*
- *Michel Bierlaire (EPFL) Teaching Gomory Cuts*

Benders Decomposition

- *Benders Decomposition - Julia Opt*
- *Youtube! SCIP lecture*

5. Exponential Size Formulations

Outcomes

We will learn two fundamental tools to be used in optimization. The desired outcomes are:

- Understand column generation and how generating solutions templates/patterns can be extremely powerful.
- Understand how cutting plane schemes can be applied when there are exponentially many constraints.

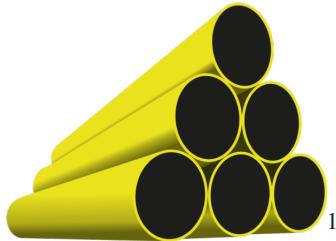
Although typically models need to be a reasonable size in order for us to code them and send them to a solver, there are some ways that we can allow having models of exponential size. The first example here is the cutting stock problem, where we will model with exponentially many variables. The second example is the traveling salesman problem, where we will model with exponentially many constraints. We will also look at some other models for the traveling salesman problem.

5.1 Cutting Stock

This is a classic problem that works excellent for a technique called *column generation*. We will discuss two versions of the model and then show how we can use column generation to solve the second version more efficiently. First, let's describe the problem.

Cutting Stock:

You run a company that sells pipes of different lengths. These lengths are L_1, \dots, L_k . To produce these pipes, you have one machine that produces pipes of length L , and then cuts them into a collection of shorter pipes as needed.



You have an order come in for d_i pipes of length i for $i = 1, \dots, k$. How can you fill the order while cutting up the fewest number of pipes?

Example 5.1: Cutting stock with pipes

A plumber stocks standard lengths of pipe, all of length 19 m. An order arrives for:

- 12 lengths of 4m
- 15 lengths of 5m
- 22 lengths of 6m

How should these lengths be cut from standard stock pipes so as to minimize the number of standard pipes used?

An initial model for this problem could be constructed as follows:

- Let N be an upper bound on the number of pipes that we may need.
- Let $z_j = 1$ if we use pipe i and $z_j = 0$ if we do not use pipe j , for $j = 1, \dots, N$.
- Let x_{ij} be the number of cuts of length L_i in pipe j that we use.

Then we have the following model

$$\begin{aligned}
 & \min \quad \sum_{j=1}^N z_j \\
 \text{s.t.} \quad & \sum_{i=1}^k L_i x_{ij} \leq L z_j \quad \text{for } j = 1, \dots, N \\
 & \sum_{j=1}^N x_{ij} \geq d_i \quad \text{for } i = 1, \dots, k \\
 & z_j \in \{0, 1\} \quad \text{for } j = 1, \dots, N \\
 & x_{ij} \in \mathbb{Z}_+ \quad \text{for } i = 1, \dots, k, j = 1, \dots, N
 \end{aligned} \tag{5.1}$$

Exercise 5.2: Show Bound

In the example above, show that we can choose $N = 16$.

demand multiplier	1	10	100	500	1000	10000	100000	200000	400000
board model time (s)	0.0517	0.6256	24.32	600					
pattern model time (s)	0.0269	0.0251	0.0289	0.0258	0.0236	0.0202	0.022	0.0186	0.0204

Table 5.1: Table comparing computational time in the two models. We stopped computations at 600 seconds. Notice that the pattern model does not care how large the demand is - it still solves in the same amount of time! The demand multiplier k here means that we multiply k times the demand vector used in the example. This grows the number of variables in the Board based model, but doesn't change much in the pattern based model.

For our example above, using $N = 16$, we have

$$\begin{aligned}
 & \min \sum_{j=1}^{16} z_j \\
 \text{s.t. } & 4x_{1j} + 5x_{2j} + 6x_{3j} \leq 19z_j \\
 & \sum_{j=1}^{16} x_{1j} \geq 12 \\
 & \sum_{j=1}^{16} x_{2j} \geq 15 \\
 & \sum_{j=1}^{16} x_{3j} \geq 22 \\
 & z_j \in \{0, 1\} \text{ for } j = 1, \dots, 16 \\
 & x_{ij} \in \mathbb{Z}_+ \text{ for } i = 1, \dots, 3, j = 1, \dots, 16
 \end{aligned} \tag{5.2}$$

Additionally, we could break the symmetry in the problem. That is, suppose the solution uses 10 of the 16 pipes. The current formulation does not restrict which 10 pipes are used. Thus, there are many possible solutions. To reduce this complexity, we can state that we only use the first 10 pipes. We can write a constraint that says *if we don't use pipe j , then we also will not use any subsequent pipes*. Hence, by not using pipe 11, we enforce that pipes 11, 12, 13, 14, 15, 16 are not used. This can be done by adding the constraints

$$z_1 \geq z_2 \geq z_3 \geq \dots \geq z_N. \tag{5.3}$$

Unfortunately, this formulation is slow and does not scale well with demand. In particular, the number of variables is $N + kN$ and the number of constraints is N (plus integrality and non-negativity constraints on the variables). The solution times for this model are summarized in the following table:

5.1.1. Pattern formulation

We could instead list all patterns that are possible to cut each pipe. A pattern is an vector $a \in \mathbb{Z}_+^k$ such that for each i , a_i lengths of L_i can be cut from a pipe of length L . That is

$$\sum_{i=1}^k L_i a_i \leq L \quad (5.4)$$

$a_i \in \mathbb{Z}_+$ for all $i = 1, \dots, k$

In our running example, we have

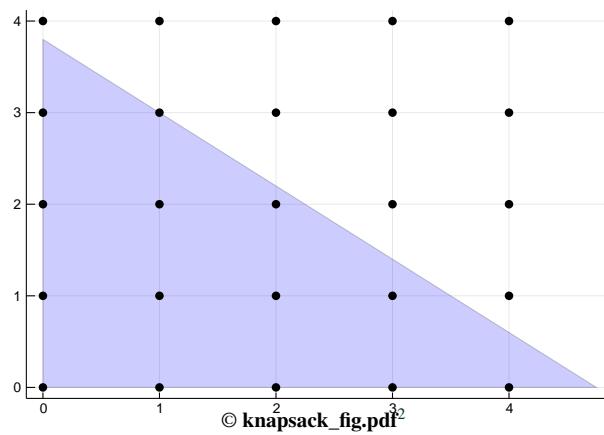
$$4a_1 + 5a_2 + 6a_3 \leq 19 \quad (5.5)$$

$a_i \in \mathbb{Z}_+$ for all $i = 1, \dots, 3$

For visualization purposes, consider the patterns where $a_3 = 0$. That is, only patterns with cuts of length 4m or 5m. All patterns of this type are represented by an integer point in the polytope

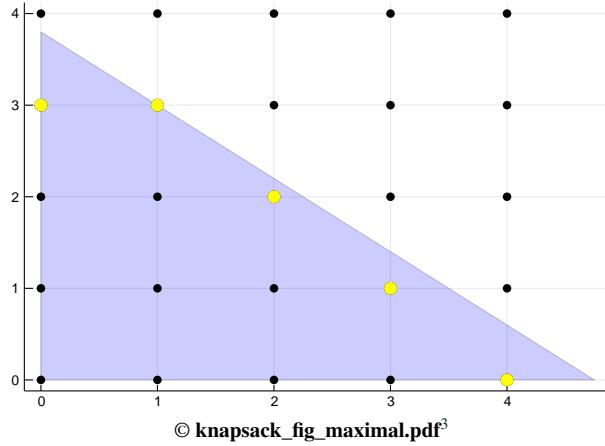
$$P = \{(a_1, a_2) : 4a_1 + 5a_2 \leq 19, a_1 \geq 0, a_2 \geq 0\} \quad (5.6)$$

which we can see here:



where P is the blue triangle and each integer point represents a pattern. Feasible patterns lie inside the polytope P . Note that we only need patterns that are maximal with respect to number of each type we cut. Pictorially, we only need the patterns that are integer points represented as yellow dots in the picture below.

²knapsack_fig.pdf, from knapsack_fig.pdf. knapsack_fig.pdf, knapsack_fig.pdf.



For example, the pattern $[3, 0, 0]$ is not needed (only cut 3 of length 4m) since we could also use the pattern $[4, 0, 0]$ (cut 4 of length 4m) or we could even use the pattern $[3, 1, 0]$ (cut 3 of length 4m and 1 of length 5m).

Example 5.3: Pattern Formulation

Let's list all the possible patterns for the cutting stock problem:

	Patterns									
Cuts of length 4m	0	0	1	0	2	1	2	3	4	1
Cuts of length 5m	0	1	0	2	1	2	2	1	0	3
Cuts of length 6m	3	2	2	1	1	0	0	0	0	0

We can organize these patterns into a matrix.

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 2 & 1 & 2 & 3 & 4 & 1 \\ 0 & 1 & 0 & 2 & 1 & 2 & 2 & 1 & 0 & 3 \\ 3 & 2 & 2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (5.7)$$

Let p be the number of patterns that we have. We create variables $x_1, \dots, x_p \in \mathbb{Z}_+$ that denote the number of times we use each pattern.

Now, we can recast the optimization problem as

$$\min \sum_{i=1}^p x_i \quad (5.8)$$

$$\text{such that } Ax \geq \begin{bmatrix} 12 \\ 15 \\ 22 \end{bmatrix} \quad (5.9)$$

$$x \in \mathbb{Z}_+^p \quad (5.10)$$

³knapsack_fig_maximal.pdf, from knapsack_fig_maximal.pdf. knapsack_fig_maximal.pdf, knapsack_fig_maximal.pdf.

5.1.2. Column Generation

Consider the linear program(??), but in this case we are instead minimizing.

Thus we can write it as

$$\begin{aligned} \min \quad & (c_N - c_B A_B^{-1} A_N) x_N + c_B A_B^{-1} b \\ \text{s.t.} \quad & x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ & x \geq 0 \end{aligned} \tag{5.11}$$

In our LP we have $c = 1$, that is, $c_i = 1$ for all $i = 1, \dots, k$. Hence, we can write it as

$$\begin{aligned} \min \quad & (1_N - 1_B A_B^{-1} N) x_N + 1_B A_B^{-1} b \\ \text{s.t.} \quad & x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ & x \geq 0 \end{aligned} \tag{5.12}$$

Now, if there exists a non-basic variable that could enter the basis and improve the objective, then there is one with a reduced cost that is negative. For a particular non-basic variable, the coefficient on it is

$$(1 - 1_B A_B^{-1} A_N^i) x_i \tag{5.13}$$

where A_N^i is the i -th column of the matrix A_N . Thus, we want to look for a column a of A_N such that

$$1 - 1_B A_B^{-1} a < 0 \Rightarrow 1 < 1_B A_B^{-1} a \tag{5.14}$$

Pricing Problem:

(knapsack problem!)

Given a current basis B of the *master* linear program, there exists a new column to add to the basis that improves the LP objective if and only if the following problem has an objective value strictly larger than 1.

$$\begin{aligned} \max \quad & 1_B A_B^{-1} a \\ \text{s.t.} \quad & \sum_{i=1}^k L_i a_i \leq L \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k \end{aligned} \tag{5.15}$$

Example 5.4: Pricing Problem

Let's make the initial choice of columns easy. We will do this by selecting columns

	Patterns		
Cuts of length 4m	4	0	0
Cuts of length 5m	0	3	0
Cuts of length 6m	0	0	3

So our initial A matrix is

$$A = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{pmatrix} \quad (5.16)$$

Notice that there are enough patterns in the initial A matrix to produce feasible solution. Let's also append an arbitrary column to the A matrix as a potential new pattern.

$$A = \begin{pmatrix} 4 & 0 & 0 & a_1 \\ 0 & 3 & 0 & a_2 \\ 0 & 0 & 3 & a_3 \end{pmatrix} \quad (5.17)$$

Now, let's solve the linear relaxation and compute the tabluea.

$$\begin{aligned} \max \quad & \frac{1}{4}a_1 + \frac{1}{3}a_2 + \frac{1}{3}a_3 \\ \text{s.t.} \quad & 4a_1 + 5a_2 + 6a_3 \leq 19 \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k \end{aligned} \quad (5.18)$$

We then add optimal solution to the master problem as a new column and repeat the procedure.

See Gurobi - Cutting Stock Example for an example of column generation implemented by the Gurobi team.

5.1.3. Cutting Stock - Multiple widths

Here are some solutions:

- <https://github.com/fzsun/cutstock-gurobi>.
- <http://www.dcc.fc.up.pt/~jpp/mpa/cutstock.py>

Here is an AIMMS description of the problem: AIMMS Cutting Stock

5.2 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) stands as one of the most iconic and studied problems in the realm of operations research and combinatorial optimization. At its core, the TSP asks a seemingly simple question: Given a list of cities and the distances between each pair, what is the shortest possible route a salesman can take to visit each city exactly once and return to the original city? While the problem statement is straightforward, finding an optimal solution, especially for a large number of cities, is computationally challenging.

The significance of the TSP extends far beyond its basic premise. It has a myriad of practical applications, ranging from logistics and transportation planning to microchip manufacturing and DNA sequencing. The TSP has been a cornerstone in highlighting the challenges of optimization in real-world scenarios. Moreover, the quest to solve the TSP has led to the development of numerous groundbreaking algorithmic techniques and heuristics. These methods, inspired by the intricacies of the TSP, have found applications in a wide array of other optimization problems, further emphasizing the TSP's foundational role in the evolution of operations research.

Google Maps!

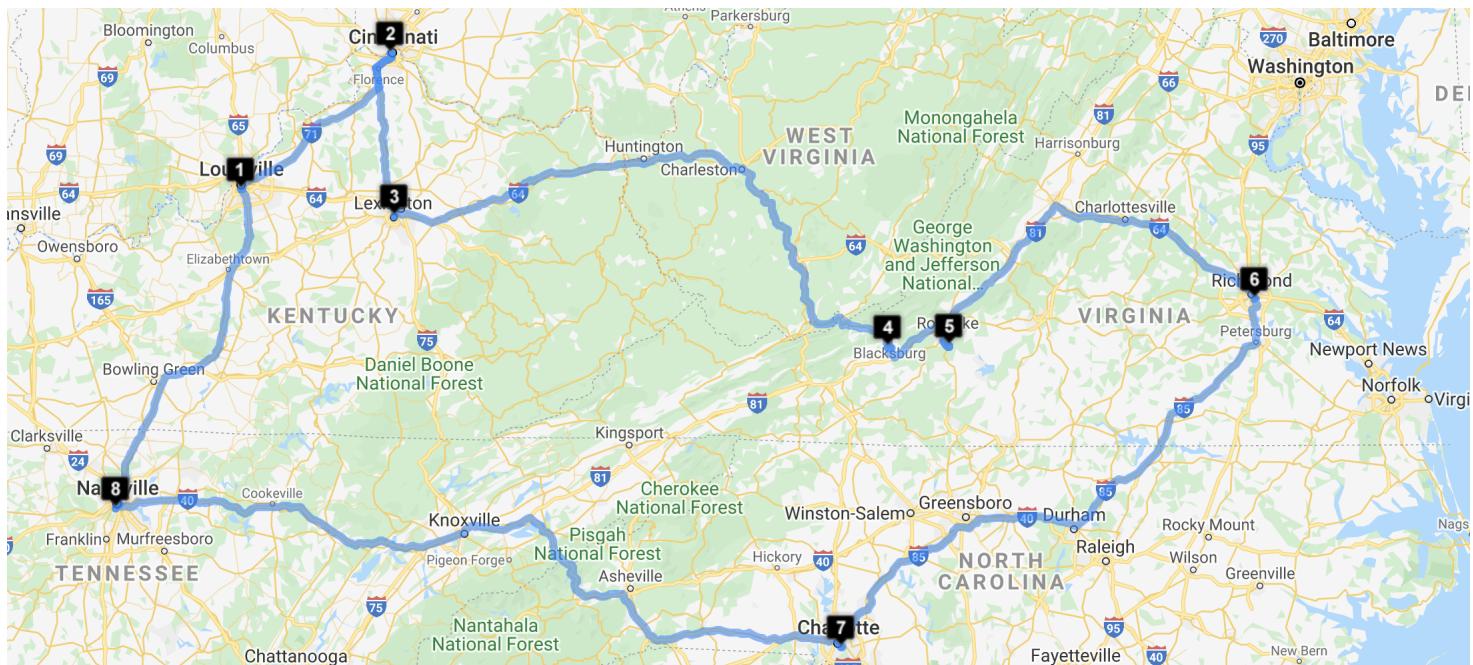


Figure 5.1: Optimal tour through 8 cities. Generated by Gebweb - Optimap. See it also on Google Maps!.

Aside from actual vehicle routing, the TSP has many applications. For example, here are some applications related to Industrial Engineering:

1. **Drilling Operations in PCB Manufacturing:** In the production of printed circuit boards (PCBs), there's a need to drill holes at various locations. The TSP can be used to determine the most efficient

sequence to drill these holes, minimizing the movement of the drilling head and thereby reducing wear and tear on the machinery and saving time.

2. **Laser Cutting:** Similar to the PCB example, in industries where laser cutting of materials is required, the TSP can help in determining the optimal path for the laser cutter to follow. This ensures that the material is cut in the shortest time, with minimal movement of the cutting apparatus.
3. **Robotic Assembly Lines:** In automated assembly lines, robots are often tasked with picking up parts from various locations and assembling them. The TSP can be used to determine the most efficient route for the robot to take, ensuring that products are assembled in the shortest time possible.
4. **Machine Sequencing:** In factories where multiple machines are used to perform different operations on a workpiece, the TSP can help determine the best sequence of machines to minimize the total processing time and movement of the workpiece.
5. **Facility Layout:** When designing the layout of a new manufacturing facility or reorganizing an existing one, the TSP can be employed to determine the optimal placement of machinery and workstations. This ensures that materials and products move through the facility in the most efficient manner, reducing transportation costs and times.
6. **Tool Switching in CNC Machines:** Computer Numerical Control (CNC) machines often have a set of tools that can be switched out depending on the operation. Determining the best order to use these tools to minimize the number of switches (and thus the downtime) can be framed as a TSP.
7. **Welding Operations:** In industries where welding is a primary operation, determining the sequence of weld joints can be crucial to minimize the movement of the welding torch and ensure efficient utilization of resources. The TSP can be applied to find the optimal sequence.
8. **Inventory Management:** In large warehouses, picking operations can be optimized using the TSP. When a list of items needs to be picked from various locations in the warehouse, the TSP can help determine the shortest route for the pickers, reducing the time taken to fulfill orders.

Problem statement

We consider a directed graph, graph $G = (N, A)$ of nodes N and arcs A . Arcs are directed edges. Hence the arc (i, j) is the directed path $i \rightarrow j$.

A *tour*, or Hamiltonian cycle is a cycle that visits all the nodes in N exactly once and returns back to the starting node.

Given costs c_{ij} for each arc $(i, j) \in A$, the goal is to find a minimum cost tour.

Traveling Salesman Problem:

NP-Hard

Given a directed graph $G = (N, A)$ and costs c_{ij} for all $(i, j) \in A$, find a tour of minimum cost.

In the figure, the nodes N are the cities and the arcs A are the directed paths $\text{city } i \rightarrow \text{city } j$.

MODELS When constructing an integer programming model for TSP, we define variables x_{ij} for all $(i, j) \in A$ as

$$x_{ij} = 1 \text{ if the arc } (i, j) \text{ is used and } x_{ij} = 0 \text{ otherwise.}$$

We want the model to satisfy the fact that each node should have exactly one incoming arc and one leaving arc. Furthermore, we want to prevent self loops. Thus, we need the constraints:

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (5.1)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (5.2)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (5.3)$$

Unfortunately, these constraints are not enough to completely describe the problem. The issue is that *subtours* may arise. For instance

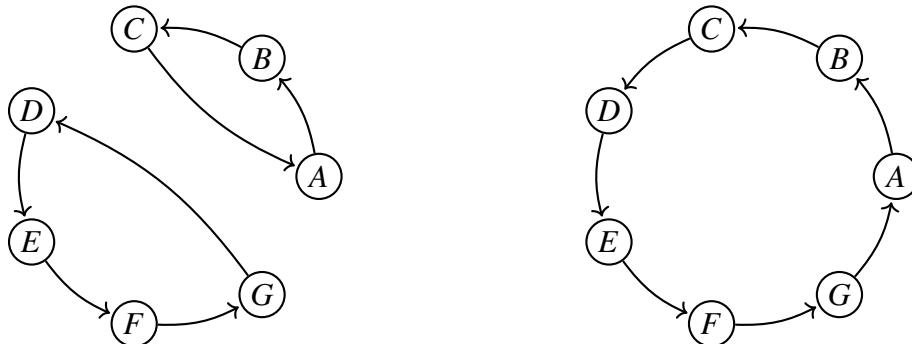


Figure 5.2: Left: An assignment solution where each node has an incoming arc and an outgoing arc. But in this case, there are 2 subtours. Right: An assignment that achieves a proper tour. Note that both of these assignments have the same number of arcs!

5.2.1. Miller Tucker Zemlin (MTZ) Model

The Miller-Tucker-Zemlin (MTZ) model for the TSP uses variables to mark the order for which cities are visited. This model introduce general integer variables to do so, but in the process, creates a formulation that has few inequalities to describe.

Some feature of this model:

- This model adds variables $u_i \in \mathbb{Z}$ with $1 \leq u_i \leq n$ that decide the order in which nodes are visited.
- We set $u_1 = 1$ to set a starting place.

- Crucially, this model relies on the following fact

Let x be a solution to (5.1)-(5.3) with $x_{ij} \in \{0, 1\}$. If there exists a subtour in this solution that contains the node 1, then there also exists a subtour that does not contain the node 1.

The following model adds constraints

$$\text{If } x_{ij} = 1, \text{ then } u_i + 1 \leq u_j. \quad (5.4)$$

This if-then statement can be modeled with a big-M, choosing $M = n$ is a sufficient upper bound. Thus, it can be written as

$$u_i + 1 \leq u_j + n(1 - x_{ij}) \quad (5.5)$$

Setting these constraints to be active enforces the order $u_i < u_j$.

Consider a subtour now $2 \rightarrow 5 \rightarrow 3 \rightarrow 2$. Thus, $x_{25} = x_{53} = x_{32} = 1$. Then using the constraints from (5.5), we have that

$$u_2 < u_5 < u_3 < u_2, \quad (5.6)$$

but this is infeasible since we cannot have $u_2 < u_2$.

As stated above, if there is a subtour containing the node 1, then there is also a subtour not containing the node 1. Thus, we can enforce these constraints to only prevent subtours that don't contain the node 1. Thus, the full tour that contains the node 1 will still be feasible.

This is summarized in the following model:

Traveling Salesman Problem - MTZ Model:

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (5.7)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (5.8)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (5.9)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (5.10)$$

$$u_i + 1 \leq u_j + n(1 - x_{ij}) \quad \text{for all } i, j \in N, i, j \neq 1 \quad [\text{prevents subtours}] \quad (5.11)$$

$$u_1 = 1 \quad (5.12)$$

$$2 \leq u_i \leq n \quad \text{for all } i \in N, i \neq 1 \quad (5.13)$$

$$u_i \in \mathbb{Z} \quad \text{for all } i \in N \quad (5.14)$$

$$x_{ij} \in \{0, 1\} \quad \text{for all } i, j \in N \quad (5.15)$$

Example 5.5: MTZ model for TSP with 4 nodes

Consider the distance matrix:

	1	2	3	4
1	0	1	2	3
2	1	0	1	2
3	2	1	0	4
4	3	2	4	0

Here is the full MTZ model:

$$\begin{aligned} \min \quad & x_{1,2} + 2x_{1,3} + 3x_{1,4} + x_{2,1} + x_{2,3} + 2x_{2,4} + \\ & 2x_{3,1} + x_{3,2} + 4x_{3,4} + 3x_{4,1} + 2x_{4,2} + 4x_{4,3} \end{aligned}$$

Subject to

$$x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} = 1 \quad \text{outgoing from node 1}$$

$$x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} = 1 \quad \text{outgoing from node 2}$$

$$x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} = 1 \quad \text{outgoing from node 3}$$

$$x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} = 1 \quad \text{outgoing from node 4}$$

$$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} = 1 \quad \text{incoming to node 1}$$

$$x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} = 1 \quad \text{incoming to node 2}$$

$$x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} = 1 \quad \text{incoming to node 3}$$

$$x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} = 1 \quad \text{incoming to node 4}$$

$$x_{1,1} = 0 \quad \text{No self loop with node 1}$$

$$x_{2,2} = 0 \quad \text{No self loop with node 2}$$

$$x_{3,3} = 0 \quad \text{No self loop with node 3}$$

$$x_{4,4} = 0 \quad \text{No self loop with node 4}$$

$$u_1 = 1 \quad \text{Start at node 1}$$

$$2 \leq u_i \leq 4, \quad \forall i \in \{2, 3, 4\}$$

$$u_2 + 1 \leq u_3 + 4(1 - x_{2,3})$$

$$u_2 + 1 \leq u_4 + 4(1 - x_{2,4}) \leq 3$$

$$u_3 + 1 \leq u_2 + 4(1 - x_{3,2}) \leq 3$$

$$u_3 + 1 \leq u_4 + 4(1 - x_{3,4}) \leq 3$$

$$u_4 + 1 \leq u_2 + 4(1 - x_{4,2}) \leq 3$$

$$u_4 + 1 \leq u_3 + 4(1 - x_{4,3}) \leq 3$$

$$x_{i,j} \in \{0, 1\} \quad \forall i \in \{1, 2, 3, 4\}, j \in \{1, 2, 3, 4\}$$

$$u_i \in \mathbb{Z}, \quad \forall i \in \{1, 2, 3, 4\}$$

Example 5.6: MTZ model for TSP with 5 nodes

$$\begin{aligned} \min \quad & x_{1,2} + 2x_{1,3} + 3x_{1,4} + 4x_{1,5} + x_{2,1} + x_{2,3} + 2x_{2,4} + 2x_{2,5} + 2x_{3,1} + \\ & x_{3,2} + 4x_{3,4} + x_{3,5} + 3x_{4,1} + 2x_{4,2} + 4x_{4,3} + 2x_{4,5} + \\ & 4x_{5,1} + 2x_{5,2} + x_{5,3} + 2x_{5,4} \end{aligned}$$

$$\text{Subject to } x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} + x_{1,5} = 1$$

$$x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} + x_{2,5} = 1$$

$$x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} + x_{3,5} = 1$$

$$x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} + x_{4,5} = 1$$

$$x_{5,1} + x_{5,2} + x_{5,3} + x_{5,4} + x_{5,5} = 1$$

$$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} + x_{5,1} = 1$$

$$x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} + x_{5,2} = 1$$

$$x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} + x_{5,3} = 1$$

$$x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} + x_{5,4} = 1$$

$$x_{1,5} + x_{2,5} + x_{3,5} + x_{4,5} + x_{5,5} = 1$$

$$x_{1,1} = 0$$

$$x_{2,2} = 0$$

$$x_{3,3} = 0$$

$$x_{4,4} = 0$$

$$x_{5,5} = 0$$

$$u_1 = 1$$

$$2 \leq u_i \leq 5 \quad \forall i \in \{1, 2, 3, 4, 5\}$$

$$u_2 + 1 \leq u_3 + 5(1 - x_{2,3})$$

$$u_2 + 1 \leq u_4 + 5(1 - x_{2,4})$$

$$u_2 + 1 \leq u_5 + 5(1 - x_{2,5})$$

$$u_3 + 1 \leq u_2 + 5(1 - x_{3,2})$$

$$u_3 + 1 \leq u_4 + 5(1 - x_{3,4})$$

$$u_4 + 1 \leq u_2 + 5(1 - x_{4,2})$$

$$u_4 + 1 \leq u_3 + 5(1 - x_{4,3})$$

$$u_3 + 1 \leq u_5 + 5(1 - x_{3,5})$$

$$u_4 + 1 \leq u_5 + 5(1 - x_{4,5})$$

$$u_5 + 1 \leq u_2 + 5(1 - x_{5,2})$$

$$u_5 + 1 \leq u_3 + 5(1 - x_{5,3})$$

$$u_5 + 1 \leq u_4 + 5(1 - x_{5,4})$$

$$x_{i,j} \in \{0, 1\} \quad \forall i \in \{1, 2, 3, 4, 5\}, j \in \{1, 2, 3, 4, 5\}$$

$$u_i \in \mathbb{Z}, \quad \forall i \in \{1, 2, 3, 4, 5\}$$

PROS OF THIS MODEL

- Small description
- Easy to implement

CONS OF THIS MODEL

- Linear relaxation is not very tight. Thus, the solver may be slow when given this model.

Example 5.7: Subtour elimination constraints via MTZ model

Consider the subtour $2 \rightarrow 4 \rightarrow 5 \rightarrow 2$.

For this subtour to exist in a solution, we must have

$$x_{2,4} = 1$$

$$x_{4,5} = 1$$

$$x_{5,2} = 1.$$

Consider the three corresponding inequalities to these variables:

$$u_2 + 1 \leq u_4 + 5(1 - x_{2,4})$$

$$u_4 + 1 \leq u_5 + 5(1 - x_{4,5})$$

$$u_5 + 1 \leq u_2 + 5(1 - x_{5,2}).$$

Since $x_{2,4} = x_{4,5} = x_{5,2} = 1$, these reduce to

$$u_2 + 1 \leq u_5$$

$$u_4 + 1 \leq u_5$$

$$u_5 + 1 \leq u_2.$$

Now, lets add these inequalities together. This produces the inequality

$$u_2 + u_4 + u_5 + 3 \leq u_2 + u_4 + u_5,$$

which reduces to

$$3 \leq 0.$$

This inequality is invalid, and hence no solution can have the values $x_{2,4} = x_{4,5} = x_{5,2} = 1$.

Example 5.8: Weak Model

Consider again the same tour in the last example, that is, the subtour $2 \rightarrow 4 \rightarrow 5 \rightarrow 2$. We are interested to know how strong the inequalities of the problem description are if we allow the variables to be continuous variables. That is, suppose we relax $x_{ij} \in \{0, 1\}$ to be $x_{ij} \in [0, 1]$.

Consider the inequalities related to this tour:

$$\begin{aligned} u_2 + 1 &\leq u_4 + 5(1 - x_{2,4}) \\ u_4 + 1 &\leq u_5 + 5(1 - x_{4,5}) \\ u_5 + 1 &\leq u_2 + 5(1 - x_{5,2}). \end{aligned}$$

A valid solution to this is

$$\begin{aligned} u_2 &= 2 \\ u_4 &= 3 \\ u_5 &= 4 \end{aligned}$$

$$\begin{aligned} 3 &\leq 3 + 5(1 - x_{2,4}) \\ 4 &\leq 4 + 5(1 - x_{4,5}) \\ 5 &\leq 2 + 5(1 - x_{5,2}). \end{aligned}$$

$$\begin{aligned} 0 &\leq 1 - x_{2,4} \\ 0 &\leq 1 - x_{4,5} \\ 3/5 &\leq 1 - x_{5,2}. \end{aligned}$$

$$\begin{aligned} 2 + 1 &\leq 3 + 5(1 - x_{2,4}) \\ 3 + 1 &\leq 4 + 5(1 - x_{4,5}) \\ 4 + 1 &\leq 2 + 5(1 - x_{5,2}). \end{aligned}$$

5.2.2. Dantzig-Fulkerson-Johnson (DFJ) Model

Resources

- Gurobi Modeling Example: TSP

This model does not add new variables. Instead, it adds constraints that conflict with the subtours. For instance, consider a subtour

$$2 \rightarrow 5 \rightarrow 3 \rightarrow 2. \quad (5.16)$$

We can prevent this subtour by adding the constraint

$$x_{25} + x_{53} + x_{32} \leq 2 \quad (5.17)$$

meaning that at most 2 of those arcs are allowed to happen at the same time. In general, for any subtour S , we can have the *subtour elimination constraint*

$$\sum_{(i,j) \in S} x_{ij} \leq |S| - 1 \quad \text{Subtour Elimination Constraint.} \quad (5.18)$$

In the previous example with $S = \{(2,5), (5,3), (3,2)\}$ we have $|S| = 3$, where $|S|$ denotes the size of the set S .

This model suggests that we just add all of these subtour elimination constraints.

Traveling Salesman Problem - DFJ Model:

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (5.19)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (5.20)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (5.21)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (5.22)$$

$$\sum_{(i,j) \in S} x_{ij} \leq |S| - 1 \quad \text{for all subtours } S \quad [\text{prevents subtours}] \quad (5.23)$$

$$x_{ij} \in \{0, 1\} \quad \text{for all } i, j \in N \quad (5.24)$$

Example 5.9: DFJ Model for $n = 4$ nodes

Consider the distance matrix:

	1	2	3	4
1	0	1	2	3
2	1	0	1	2
3	2	1	0	4
4	3	2	4	0

$$\begin{aligned} \min \quad & x_{1,2} + 2x_{1,3} + 3x_{1,4} + x_{2,1} + x_{2,3} + 2x_{2,4} \\ & + 2x_{3,1} + x_{3,2} + 4x_{3,4} + 3x_{4,1} + 2x_{4,2} + 4x_{4,3} \end{aligned}$$

Subject to

$$x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} = 1 \quad \text{outgoing from node 1}$$

$$x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} = 1 \quad \text{outgoing from node 2}$$

$$x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} = 1 \quad \text{outgoing from node 3}$$

$$x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} = 1 \quad \text{outgoing from node 4}$$

$$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} = 1 \quad \text{incoming to node 1}$$

$$x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} = 1 \quad \text{incoming to node 2}$$

$$x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} = 1 \quad \text{incoming to node 3}$$

$$x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} = 1 \quad \text{incoming to node 4}$$

$$x_{1,1} = 0 \quad \text{No self loop with node 1}$$

$$x_{2,2} = 0 \quad \text{No self loop with node 2}$$

$$x_{3,3} = 0 \quad \text{No self loop with node 3}$$

$$x_{4,4} = 0 \quad \text{No self loop with node 4}$$

$$x_{1,2} + x_{2,1} \leq 1 \quad S = [(1,2), (2,1)]$$

$$x_{1,3} + x_{3,1} \leq 1 \quad S = [(1,3), (3,1)]$$

$$x_{1,4} + x_{4,1} \leq 1 \quad S = [(1,4), (4,1)]$$

$$x_{2,3} + x_{3,2} \leq 1 \quad S = [(2,3), (3,2)]$$

$$x_{2,4} + x_{4,2} \leq 1 \quad S = [(2,4), (4,2)]$$

$$x_{3,4} + x_{4,3} \leq 1 \quad S = [(3,4), (4,3)]$$

$$x_{2,1} + x_{1,3} + x_{3,2} \leq 2 \quad S = [(2,1), (1,3), (3,2)]$$

$$x_{1,2} + x_{2,3} + x_{3,1} \leq 2 \quad S = [(1,2), (2,3), (3,1)]$$

$$x_{3,1} + x_{1,4} + x_{4,3} \leq 2 \quad S = [(3,1), (1,4), (4,3)]$$

$$x_{1,3} + x_{3,4} + x_{4,1} \leq 2 \quad S = [(1,3), (3,4), (4,1)]$$

$$x_{2,1} + x_{1,4} + x_{4,2} \leq 2 \quad S = [(2,1), (1,4), (4,2)]$$

$$x_{1,2} + x_{2,4} + x_{4,1} \leq 2 \quad S = [(1,2), (2,4), (4,1)]$$

$$x_{3,2} + x_{2,4} + x_{4,3} \leq 2 \quad S = [(3,2), (2,4), (4,3)]$$

$$x_{2,3} + x_{3,4} + x_{4,2} \leq 2 \quad S = [(2,3), (3,4), (4,2)]$$

$$x_{i,j} \in \{0, 1\} \quad \forall i \in \{1, 2, 3, 4\}, j \in \{1, 2, 3, 4\}$$

Example 5.10

Consider a graph on 5 nodes.

Here are all the subtours of length at least 3 and also including the full length tours.

Hence, there are many subtours to consider.

PROS OF THIS MODEL

- Very tight linear relaxation

CONS OF THIS MODEL

- Exponentially many subtours S possible, hence this model is too large to write down.

SOLUTION: ADD SUBTOUR ELIMINATION CONSTRAINTS AS NEEDED . This is a similar concept as *cutting planes*.

5.2.3. Traveling Salesman Problem - Branching Solution

We will see in the next section

1. That the constraint (5.1)-(5.3) always produce integer solutions as solutions to the linear relaxation.
2. A way to use branch and bound (the topic of the next section) in order to avoid subtours.

5.2.4. Traveling Salesman Problem Variants

5.2.4.1. Many salespersons (m-TSP)

m-Traveling Salesman Problem - DFJ Model:

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (5.25)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (5.26)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \text{ [incoming arc]} \quad (5.27)$$

$$\sum_{j \in N} x_{Dj} = m \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (5.28)$$

$$\sum_{i \in N} x_{iD} = m \quad \text{for all } j \in N \text{ [incoming arc]} \quad (5.29)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \cup \{D\} \text{ [no self loops]} \quad (5.30)$$

$$\sum_{(i,j) \in S} x_{ij} \leq |S| - 1 \quad \text{for all subtours } S \subseteq N \text{ [prevents subtours]} \quad (5.31)$$

$$x_{ij} \in \{0,1\} \quad \text{for all } i, j \in N \cup \{D\} \quad (5.32)$$

When using the MTZ model, you can also easily add in a constraint that restricts any subtour through the deopt to have at most T stops on the tour. This is done by restricting $u_i \leq T$. This could also be done in the DFJ model above, but the algorithm for subtour elimination cuts would need to be modified.

m-Travelling Salesman Problem - MTZ Model - :

Python Code

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (5.33)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (5.34)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \text{ [incoming arc]} \quad (5.35)$$

$$\sum_{j \in N} x_{Dj} = m \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (5.36)$$

$$\sum_{i \in N} x_{iD} = m \quad \text{for all } j \in N \text{ [incoming arc]} \quad (5.37)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \cup \{D\} \text{ [no self loops]} \quad (5.38)$$

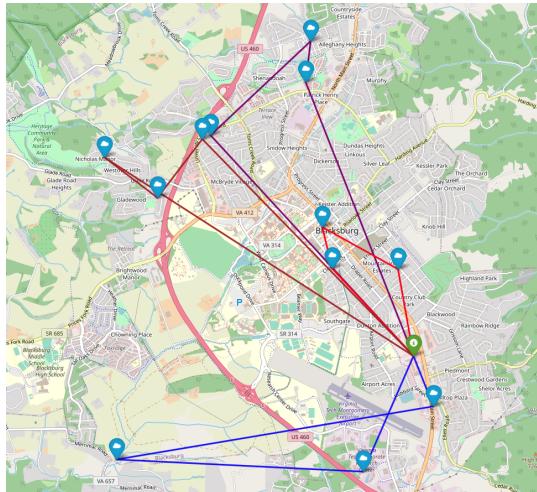
$$u_i + 1 \leq u_j + n(1 - x_{ij}) \quad \text{for all } i, j \in N, i, j \neq 1 \text{ [prevents subtours]} \quad (5.39)$$

$$u_1 = 1 \quad (5.40)$$

$$2 \leq u_i \leq T \quad \text{for all } i \in N, i \neq 1 \quad (5.41)$$

$$u_i \in \mathbb{Z} \quad \text{for all } i \in N \quad (5.42)$$

$$x_{ij} \in \{0,1\} \quad \text{for all } i, j \in N \cup \{D\} \quad (5.43)$$



© m-tsp_solution⁴

[Image html](#)

5.2.4.2. TSP with order variants

Using the MTZ model, it is easy to provide order variants. Such as, city 2 must come before city 3

$$u_2 \leq u_3$$

or city 2 must come directly before city 3

$$u_2 + 1 = u_3.$$

5.2.5. Multi vehicle model with capacities

In this version, we explicitly add extra variable to discuss the routes of each vehicle. This means that we add another index to the x variable.

Let:

c_{ij} : cost of travel from i to j

$$x_{ijk} : \begin{cases} 1 & \text{if vehicle } k \text{ travels directly from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

Objective:

$$\min \sum_{i,j} c_{ij} \sum_k x_{ijk} \quad (5.44)$$

⁴m-tsp_solution, from m-tsp_solution. m-tsp_solution, m-tsp_solution.

Subject to:

$$\sum_i \sum_k x_{ijk} = 1 \quad \forall j \neq \text{depot} \quad (\text{Exactly one vehicle in}) \quad (5.45)$$

$$\sum_j \sum_k x_{ijk} = 1 \quad \forall i \neq \text{depot} \quad (\text{Exactly one vehicle out}) \quad (5.46)$$

$$\sum_i \sum_k x_{ihk} - \sum_j \sum_k x_{hjk} = 0 \quad \forall k, h \quad (\text{It's the same vehicle}) \quad (5.47)$$

$$\sum_i q_i \sum_j x_{ijk} \leq Q_k \quad \forall k \quad (\text{Capacity constraint}) \quad (5.48)$$

$$\sum_{ijk} x_{ijk} = |S| - 1 \quad \forall S \subseteq P(N), 0 \notin S \quad (\text{Subtour elimination}) \quad (5.49)$$

$$x_{ijk} \in \{0, 1\} \quad (5.50)$$

5.3 Vehicle Routing Problem (VRP)

The VRP is a generalization of the TSP and comes in many many forms. The major difference is now we may consider multiple vehicles visiting the around cities. Obvious examples are creating bus schedules and mail delivery routes.

Variations of this problem include

- Time windows (for when a city needs to be visited)
- Prize collecting (possibly not all cities need to be visited, but you gain a prize for visiting each city)
- Multi-depot vehicle routing problem (fueling or drop off stations)
- Vehicle rescheduling problem (When delays have been encountered, how do you adjust the routes)
- Inhomogeneous vehicles (vehicles have different abilities (speed, distance, capacity, etc.).)

To read about the many variants, see: Vehicle Routing: Problems, Methods, and Applications, Second Edition. Editor(s): Paolo Toth and Daniele Vigo. MOS-SIAM Series on Optimization.

For one example of a VRP model, see GUROBI Modeling Examples - technician routing scheduling.

5.4 The Clarke-Wright Saving Algorithm for VRP

[[Under Construction]]

The Vehicle Routing Problem (VRP) is a classic combinatorial optimization problem that aims to serve a number of customers with a fleet of vehicles in the most cost-effective way. The Clarke-Wright Saving Algorithm is a heuristic approach to solving the VRP.

5.4.1. Concept

Given a depot and a set of customers, the idea is to evaluate the ‘savings’ that can be achieved by merging two routes into one, instead of serving each customer by a separate route starting and ending at the depot.

5.4.2. Saving Calculation

The saving s_{ij} of merging the route of customer i and customer j is given by:

$$s_{ij} = c(0, i) + c(0, j) - c(i, j) \quad (5.1)$$

Where:

- $c(x, y)$ represents the cost (or distance) between node x and node y .
- 0 is the depot.

5.4.3. Algorithm Steps

1. For each pair of customers i and j , calculate the saving s_{ij} and list all the savings in descending order.
2. Initialize a route for each customer starting and ending at the depot.
3. Iterate through the savings list. For the largest saving s_{ij} :
 - If customers i and j can be merged without exceeding vehicle capacity and without violating any route constraints, merge their routes.
 - Remove individual routes of i and j .
4. Repeat step 3 until there are no more feasible merges.

5.4.4. Advantages and Limitations

While the Clarke-Wright Saving Algorithm is computationally efficient and can provide good solutions for the VRP, it is not guaranteed to find the optimal solution. It works best as a starting solution, which can be further improved using other optimization techniques.

Borrowed from https://www.researchgate.net/publication/285833854_Chapter_4_Heuristics_for_the_Vehicle_Routing_Problem

The Clarke and Wright Savings Heuristic The Clarke and Wright heuristic [12] initially constructs back and forth routes $(0, i, 0)$ for $(i = 1, \dots, n)$ and gradually merges them by applying

a saving criterion. More specifically, merging the two routes $(0, \dots, i, 0)$ and $(0, j, \dots, 0)$ into a single route $(0, \dots, i, j, \dots, 0)$ generates a saving $s_{ij} = c_{i0} + c_{0j} - c_{ij}$. Since the savings remain the same throughout the algorithm, they can be computed a priori. In the so-called parallel version of the algorithm which appears to be the best (see Laporte and Semet [46]), the feasible route merger yielding the largest saving is implemented at each iteration, until no more merger is feasible. This simple algorithm possesses the advantages of being intuitive, easy to implement, and fast. It is often used to generate an initial solution in more sophisticated algorithms. Several enhancements and acceleration procedures have been proposed for this algorithm (see, e.g., Nelson et al. [59] and Paessens [62]), but given the speed of today's computers and the robustness of the latest metaheuristics, these no longer seem justified.

[12] G. CLARKE AND J. W. WRIGHT, Scheduling of vehicles from a central depot to a number of delivery points, *Operations Research*, 12 (1964), pp. 568-581.

5.5 Literature and other notes

- Gilmore-Gomory Cutting Stock [[Gilmore-Gomory](#)]
- A Column Generation Algorithm for Vehicle Scheduling and Routing Problems
- The Integrated Last-Mile Transportation Problem
- http://www.optimization-online.org/DB_FILE/2017/11/6331.pdf A BRANCH-AND-PRICE ALGORITHM FOR CAPACITATED HYPERGRAPH VERTEX SEPARATION

5.5.1. TSP In Excel

5.5.2. Resources

TSP

Resources

See math.watloo.ca for excellent material on the TSP.

See also this chapter *A Practical Guide to Discrete Optimization*.

Also, watch this excellent talk by Bill Cook "Postcards from the Edge of Possibility": [Youtube!](https://www.youtube.com/watch?v=JyfXWzvDwIY)

TSP with excel solver

Google maps data: [Blog - Python | Calculate distance and duration between two places using google distance matrix API](https://www.blog.pythonlibrary.org/2018/02/08/calculate-distance-and-duration-between-two-places-using-google-distance-matrix-api/)

Resources

[https://www.informs.org/Impact/0.R.-Analytics-Success-Stories/
Optimized-school-bus-routing-helps-school-districts-design-better-policies](https://www.informs.org/Impact/0.R.-Analytics-Success-Stories/Optimized-school-bus-routing-helps-school-districts-design-better-policies)
<https://pubsonline.informs.org/doi/abs/10.1287/inte.2019.1015>
[https://www.informs.org/Resource-Center/Video-Library/
Edelman-Competition-Videos/2019-Edelman-Competition-Videos/
2019-Edelman-Finalist-Boston-Public-Schools](https://www.informs.org/Resource-Center/Video-Library/Edelman-Competition-Videos/2019-Edelman-Competition-Videos/2019-Edelman-Finalist-Boston-Public-Schools)
<https://www.youtube.com/watch?v=LFeeaNPrbY>
Fantastic talk - Very thorough
<https://www.opendoorlogistics.com/tutorials/tutorial-v-vehicle-routing-scheduling/>

Resources

Cutting stock with multiple widths:
Gurobi has an excellent demonstration application to look at: [Gurobi - Cutting Stock Demo Gurobi - Multiple Master Rolls](#)

6. Algorithms and Complexity

Outcomes

1. *Describe asymptotic growth of functions using Big-O notation.*
2. *Analyze algorithms for the asymptotic runtime.*
3. *Classify problem types with respect to notions of how difficult they are to solve.*

How long will an algorithm take to run? How difficult might it be to solve a certain problem? Is the knapsack problem easier to solve than the traveling salesman problem? Or the matching problem? How can we compare the difficulty to solve these problems?

We will understand these questions through complexity theory. We will first use "Big-O" notation to simplify asymptotic analysis of the runtime of algorithms and the size of the input data of an algorithm.

We will then classify problem types as being either easy (in the class P) or probably very hard (in the class NP Hard). We will also learn about the problem classes NP, and NP-Complete.

To begin, watch these videos (Video 1, Video 2) about sorting algorithms. Notice how a different algorithm can produce a much different number of steps needed to solve the problem. The first video explains bubble sort and quick sort. The second video explains insertion sort, and then described the analysis of the algorithms (how many comparisons they make as the number of balls to sort grows. Pay attention to this analysis as this is very crucial in this module.

This video is a great introduction to the basic idea of Big-O notation. We will go over the more formal definition.

Here are two great videos about P versus NP (Video 1, Video 2).

6.1 Big-O Notation

We begin with some definitions that relate the rate of growth of functions. The functions we will look at in the next section will describe the runtime of an algorithm.

Example 6.1: Relations of functions

We want to understand the asymptotic growth of the following functions:

- $f(n) = n^2 + 5$,
- $g(n) = n^3 - 10n^2 - 10$.

When we discuss asymptotic growth, we don't care so much what happens for small values of n , and instead, we want to know what happens for large values of n .

Notice that

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0. \quad (6.1)$$

This is because as n gets large, $g(n) \gg f(n)$. However, this does not preclude the possibility that $g(n) < f(n)$ for some small values of n , (i.e., $n = 1, 2, 3$).

We can, however see that $g(n) > f(n)$ whenever $n \geq N := 20$ (it is probably true for a smaller value of n , but for the sake of the analysis, we don't care).

Thus, we want to say that $g(n)$ grows faster than $f(n)$.

Example 6.2: Asymptotic Technicality

It may be that we consider functions that are not strictly increasing after some point. For example,

- $f(n) = \sin(n)(n^2 + 5)$,
- $g(n) = 10n^2 - 10$.

Still, we would like to say that $f(n)$ is bounded somehow by $g(n)$. But! The limit $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ does not exist!

For this, we use the \limsup notation. That is, we notice that

$$\limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty. \quad (6.2)$$

This completely captures our goal here. However, we will give an alternative definition that allows us to not have to think about the \limsup .

Definition 6.3: Big-O

For two functions $f(n)$ and $g(n)$, we say that $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq c g(n) \quad \text{for all } n \geq n_0. \quad (6.3)$$

Example 6.4:

Consider $f(n) = 5n^2 + 10n + 7$ and $g(n) = n^2$. We want to show that $f(n) = O(g(n))$.

Let's try $c = 22$ and $n_0 = 1$. We need to show that Equation 6.3 is satisfied.

Note first that we always have

$$1. n^2 \leq n^2 \text{ and therefore } 5n^2 \leq 5n^2$$

Note that if $n \geq 1$, then

$$2. n \leq n^2 \text{ and therefore } 10n \leq 10n^2$$

$$3. 1 \leq n^2 \text{ and therefore } 7 \leq 7n^2$$

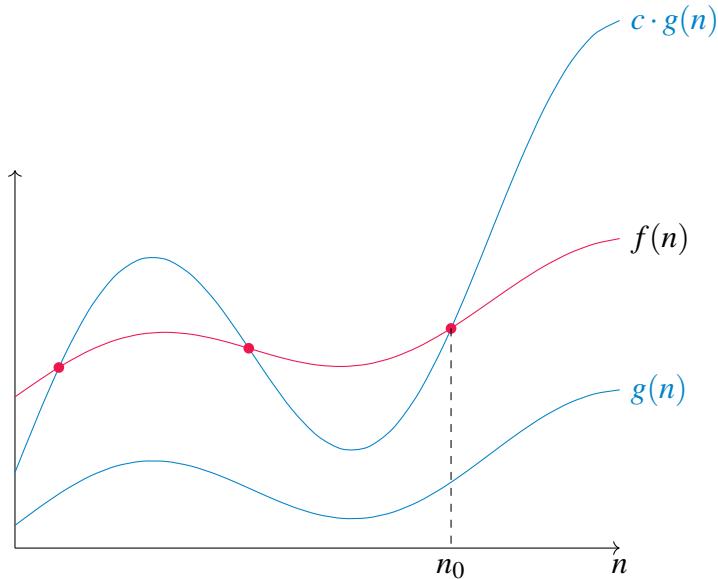


Figure 6.1: Example of Big-O notation: $f(n) = O(g(n))$. We see that for all $n \geq n_0$, we have $c \cdot g(n) \geq f(n)$.

Since all inequalities 1,2, and 3 are valid for $n \geq 1$, by adding them, we obtain a new inequality that is also valid for $n \geq 1$, which is

$$5n^2 + 10n + 7 \leq 5n^2 + 10n^2 + 7n^2 \quad \text{for all } n \geq 1, \quad (6.4)$$

$$\Rightarrow 5n^2 + 10n + 7 \leq 22n^2 \quad \text{for all } n \geq 1. \quad (6.5)$$

Hence, we have shown that Equation 6.3 holds for $c = 22$ and $n_0 = 1$. Hence $f(n) = O(g(n))$.

Correct uses:

- $2^n + n^5 + \sin(n) = O(2^n)$
- $2^n = O(n!)$
- $n! + 2^n + 5n = O(n!)$
- $n^2 + n = O(n^3)$
- $n^2 + n = O(n^2)$
- $\log(n) = O(n)$
- $10\log(n) + 5 = O(n)$

Notice that not all examples above give a tight bound on the asymptotic growth. For instance, $n^2 + n = O(n^3)$ is true, but a tighter bound is $n^2 + n = O(n^2)$.

In particular, the goal of big O notation is to give an upper bound on the asymptotic growth of a function. But we would prefer to give a strong upper bound as opposed to a weak upper bound. For instance, if

you order a package online, you will typically be given a bound on the latest date that it will arrive. For example, if it will arrive within a week, you might be guaranteed that it will arrive by next Tuesday. This sounds like a reasonable bound. But if instead, they tell you it will arrive before 1 year from today, this may not be as useful information. In the case of big O notation, we would like to give a least upper bound that most simply describes the growth behavior of the function.

In that example, $n^2 + n = O(n^2)$, this literally means that there is some number c and some value n_0 that $n^2 + n \leq cn^2$ for all $n \geq n_0$, that is, for all values of n_0 larger than n , the function cn^2 dominates $n^2 + n$.

For example, a valid choice is $c = 2$ and $n_0 = 1$. Then it is true that $n^2 + n \leq 2n^2$ for all $n \geq 1$.

But it is also true that $n^2 + n = O(n^3)$. For example, a valid choice is again $c = 2$ and $n_0 = 1$, then

$$n^2 + n \leq 2n^3 \text{ for all } n \geq 1.$$

In this example, $O(n^3)$ is the case where the internet tells you the package will arrive before 1 year from today. The bound is true, but it is not as useful information as we would like to have. Let's compare these upper bounds. Let

$$\begin{aligned} f(n) &= n^2 + n, \\ g(n) &= 2n^2, \\ h(n) &= 2n^3. \end{aligned}$$

Then we have:

	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
$f(n) = n^2 + n$	110	10,100	1,001,000	100,010,000
$g(n) = 2n^2$	200	20,000	2,000,000	200,000,000
$h(n) = 2n^3$	2,000	2,000,000	2,000,000,000	2,000,000,000,000

So, here we see that $g(n)$ and $h(n)$ are both upper bounds on $f(n)$, but the nice part about $g(n)$ is that is growing at a similar rate to $f(n)$. In particular, it is always within a factor of 2 of $f(n)$.

Alternatively, the bound $h(n)$ is true, but it grows so much faster than $f(n)$ that is doesn't give a good idea of the asymptotic growth of $f(n)$.

Some common classes of functions:

$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n^c)$ (for $c > 1$)	Polynomial
$O(c^n)$ (for $c > 1$)	Exponential

¹time-of-algorithms, from time-of-algorithms. time-of-algorithms, time-of-algorithms.

Exponential Time Algorithms do not currently solve reasonable-sized problems in reasonable time.

Polynomial	$\log n$	3	4
	n	10	20
	$n \log n$	33	86
	n^2	100	400
	n^3	1,000	8,000
	n^5	100,000	3,200,000
	n^{10}	10,000,000,000	10,240,000,000,000
Exponential	n	10	20
	$n^{\log n}$	2,099	419,718
	2^n	1,024	1,048,576
	5^n	9,765,625	95,367,431,640,625
	$n!$	3,628,800	2,432,902,008,176,640,000
	n^n	10,000,000,000	104,857,600,000,000,000,000,000,000

© time-of-algorithms¹

Figure 6.2: time-of-algorithms

6.2 Algorithms - Example with Bubble Sort

The following definition comes from Merriam-Webster's dictionary.

Definition 6.5: Algorithm

An algorithm is a procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end.

6.2.1. Sorting

The problem of sorting a list is a simple problem to think about that has many algorithms to consider. We will describe one such algorithm: Bubble Sort.

Sorting Problem:

Polynomial time (P)

Given a list of numbers (x_1, \dots, x_n) sort them into increasing order.

Example 6.6: Sorting Problem

Suppose you have the list of numbers $(10, 35, 9, 4, 15, 22)$.

The sorted list of numbers is $(4, 9, 10, 15, 22, 35)$.

What process or algorithm should we use to compute the sorted list?

Bubble sort algorithm:

The *Bubble Sort* algorithm works as follows:

1. Compare numbers in position 1 and 2. If numbers are out of order, then swap them.
 2. Next, compare numbers in position 2 and 3. If numbers are out of order, then swap them.
 3. Continue this process of comparing subsequent numbers until you get to the end of the list (and compare numbers in position $n - 1$ and n).
- Now the largest number should be in last position!
4. If no swaps had to be made, then the whole list is sorted!
 5. Otherwise, if any swaps were needed, then set the last number aside, and start over from the beginning and sort the remaining list.

Example: Bubble Sort

Let try using Bubble Sort to sort this list.

First pass through the list.

Step 1:

$$(10, 35, 9, 4, 15, 22) \rightarrow (10, 35, 9, 4, 15, 22)$$

Step 2:

$$(10, 35, 9, 4, 15, 22) \rightarrow (10, 9, 35, 4, 15, 22)$$

Step 3:

$$(10, 9, 35, 4, 15, 22) \rightarrow (10, 9, 4, 35, 15, 22)$$

Step 4:

$$(10, 9, 4, 35, 15, 22) \rightarrow (10, 9, 4, 15, 35, 22)$$

Step 5:

$$(10, 9, 4, 15, 35, 22) \rightarrow (10, 9, 4, 15, 22, 35)$$

Now 35 is in the last spot!

Second pass through the list

Step 1:

$$(10, 9, 4, 15, 22 | 35) \rightarrow (9, 10, 4, 15, 22 | 35)$$

Step 2:

$$(9, 10, 4, 15, 22 | 35) \rightarrow (9, 4, 10, 15, 22 | 35)$$

Step 3:

$$(9, 4, 10, 15, 22 | 35) \rightarrow (9, 4, 10, 15, 22 | 35)$$

Step 4:

$$(9, 4, 10, 15, 22 | 35) \rightarrow (9, 4, 10, 15, 22 | 35)$$

Now 22 is in the correct spot!

Third pass through the list

Step 1:

$$(9, 4, 10, 15, | 22, 35) \rightarrow (4, 9, 10, 15, | 22, 35)$$

Step 2:

$$(4, 9, 10, 15, | 22, 35) \rightarrow (4, 9, 10, 15, | 22, 35)$$

Step 3:

$$(4, 9, 10, 15, | 22, 35) \rightarrow (4, 9, 10, 15, | 22, 35)$$

Fourth pass through the list

Step 1:

$$(4, 9, 10, | 15, 22, 35) \rightarrow (4, 9, 10, | 15, 22, 35)$$

Step 2:

$$(4, 9, 10, | 15, 22, 35) \rightarrow (4, 9, 10, | 15, 22, 35)$$

No swaps were necessary! We must be done!

How many comparisons were needed?

- In the first pass, we needed 5 comparisions
- In the second pass, we needed 4 comparisions
- In the third pass, we needed 3 comparisions
- In the fourth pass, we needed 2 comparisions

Thus we used

$$5 + 4 + 3 + 2 = 14$$

comparisons.

Example: Worst Case Analysis**What is the worst case number of comparisons?**For a list of n numbers, the worst case would be

$$(n-1) + (n-2) + \cdots + 2 + 1.$$

Notice that we can compute this sum exactly in a shorter form. To do so, let's count the number of pairs that we can get to add up to n . Suppose that n is an even number.

$$\begin{aligned} (n-1) + 1 &= n \\ (n-2) + 2 &= n \\ (n-3) + 3 &= n \\ &\vdots \\ (n/2+1) + (n/2-1) &= n \end{aligned}$$

Then we also have the number $n/2$ left over.Adding all this up, we have $(n/2+1)$ pairs that add up to n , plus one $n/2$ left over.

Hence, the sum is

$$n(\frac{n}{2} - 1) + \frac{n}{2} = \frac{n(n-1)}{2}.$$

Hence, we have proved that

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

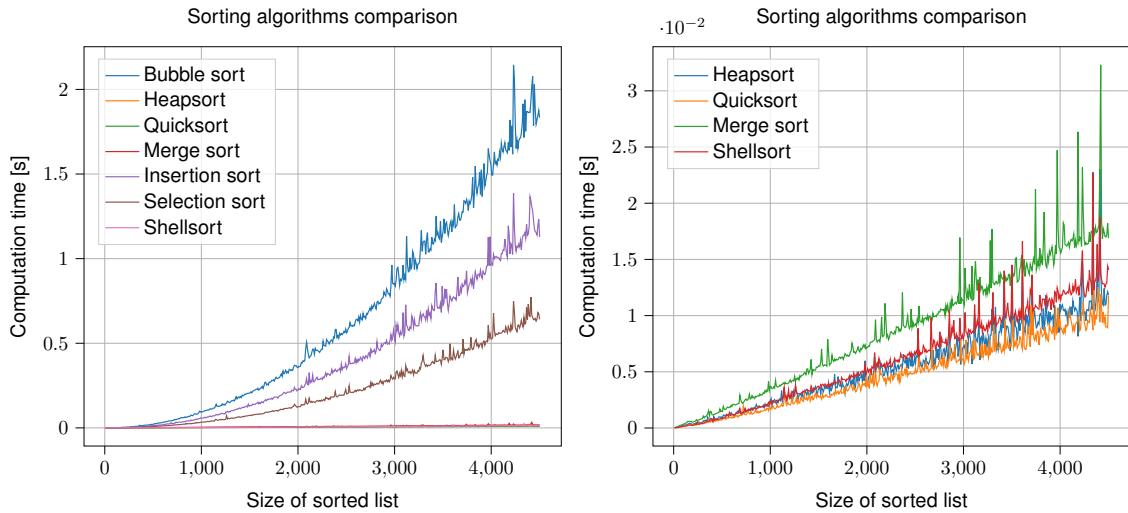
Since we just care about the Big-O expression, we can upper bound this by $O(n^2)$.

Hence, we will say that

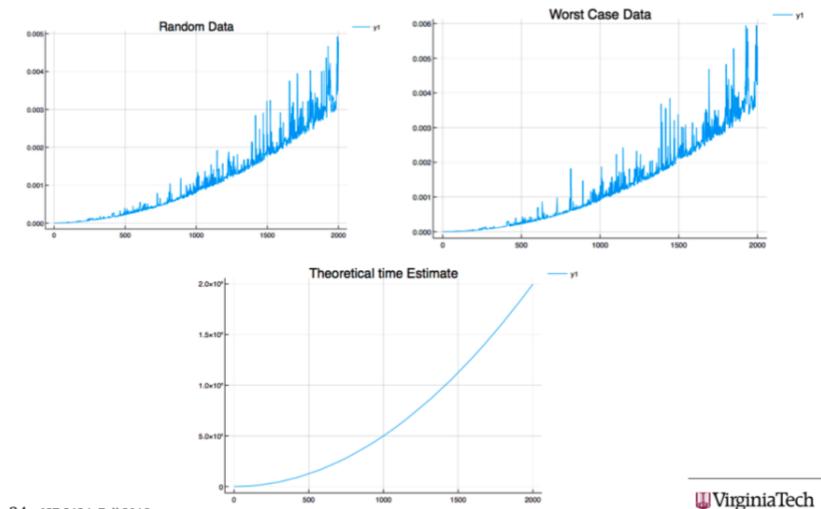
These can be verified experimentally as seen in the following plot. The random case grows quadratically just as the worst case does.

There are some other relations that hold:

²[bubble-sort-computational-example](#), from [bubble-sort-computational-example](#). [bubble-sort-computational-example](#), [bubble-sort-computational-example](#).

**Figure 6.3: Comparison of runtimes of sorting algorithms.**

Time elapsed in computer for bubble sort

**Figure 6.4: bubble-sort-computational-example**

Theorem 6.7: Summations

- $\sum_{i=1}^n i = \frac{n(n+1)}{2}$.
- $\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$.
- $\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$.

There are other formulas, but they get more complicated. In general, we know that

- $\sum_{i=1}^n i^k = O(n^{k+1})$.

6.3 Problem, instance, size

6.3.1. Problem, instance

Definition 6.8: Problem

Is a generic question/task that needs to be answered/solved.

A problem is a “collection of instances” (see below).

A particular realization of a problem is defined next.

Definition 6.9: Instance

An instance is a specific case of a problem. For example, for the problem of sorting, an instance we saw already is (4, 9, 10, 15, 22, 35).

6.3.2. Format and examples of problems/instances

A problem is an abstract concept. We will write problems in the following format:

- **INPUT:** Generic data/instance.
- **OUTPUT:** Question to be answered and/or task to be performed with the data.

Examples of problems/instances:

- Typical problems: optimization problems, decision problems, feasibility problems.
- LP and IP feasibility, TSP, IP minimization, Maximum cardinality independent set.

6.3.3. Size of an instance

The size of an instance is the *amount of information* required to represent the instance (in the computer). Typically, this information is bounded by the quantity of numbers in the problem and the size of the numbers.

Example 6.10: Size of Sorting Problem

Most of the time, we will think of the size of the sorting problem as

n ,

which is the number of numbers taht we need to sort.

However, we should also keep in mind that the size of the numbers is also important. That is, if the numbers we are asked to sort take up 1 gigabyte of space to write down, then merely comparing these numbers could take a long time.

So to be more precise, the size of the problem is

of bits to encode the problem

which can be upper bounded by

$$n\phi_{\max}$$

where ϕ_{\max} is the maximum encoding size of a number given in the data.

For the sake of simplicity, we will typically ignore the size ϕ_{\max} in our complexity discussion. A more rigorous discussion of complexity will be given in later (advanced) parts of the book.

Example 6.11: Size of Matching Problem

The matching problem is presented as a graph $G = (V, E)$ and a set of costs c_e for all $e \in E$. Thus, the size of the problem can be described as $|V| + |E|$, that is, in terms of the number of nodes and the number of edges in the graph.

6.4 Complexity Classes

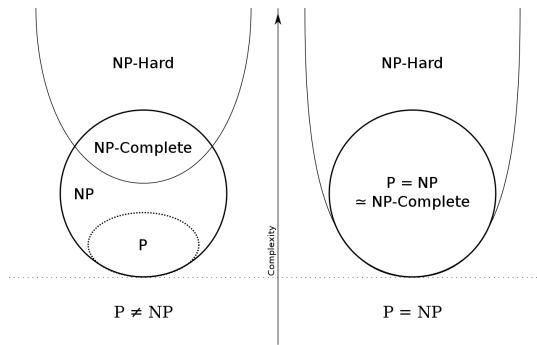
In this subsection we will discuss the complexity classes P, NP, NP-Complete, and NP-Hard. These classes help measure how difficult a problem is. **Informally**, these classes can be thought of as

- P - the class of efficiently solvable problems
- NP - the class of efficiently checkable problems
- NP-Hard - the class of problems that can solve any problem in NP
- NP-Complete - the class of problems that are in both NP and are NP-Hard.

It is not known if P is the same as NP, but it is conjectured that these are very different classes. This would mean that the NP-Hard problems (and NP-Complete problems) are necessarily much more difficult than the problems in P. See Figure 6.5 .

We will now discuss these classes more formally.

³wiki/File/complexity-classes.png, from wiki/File/complexity-classes.png. wiki/File/complexity-classes.png, wiki/File/complexity-classes.png.



© wiki/File/complexity-classes.png³

Figure 6.5: Complexity class possibilities. Most academics agree that the case $P \neq NP$ is more likely.

6.4.1. P

Definition 6.12: P

P is the class of polynomially solvable problems. *P* contains all problems for which there exists an algorithm that solves the problem in a run time bounded by a polynomial of input size. That is, $O(n^c)$ for some constant c .

Example 6.13: Sorting

The sorting problem can be solved in $O(n^2)$ time. Thus, this problem is in *P*

Example 6.14: Complexity Minimum Spanning Tree

The minimum size spanning tree problem is in *P*. It can be solved, for instance, by Prim's algorithm, which runs in time $O(m \log n)$, where m is the number of edges in the graph and n is the number of nodes in the graph.

Example 6.15: Complexity Linear Programming

Linear programming is in *P*. It can be solved by interior point methods in $O(n^{3.5}\phi)$ where ϕ represents the number of binary bits that are required to encode the problem. These bits describe the matrix *A*, and vectors *c* and *b* that define the linear program.

6.4.2. NP

In this section, we will be more specific about the types of problems we want to consider. In particular, we will consider *decisions problems*. These are problems where we only request an answer of "yes" or "no".

We can rephrase maximization problems as problems that ask "does there exists a solution with objective value greater than some number?"

Example 6.16: Maximum Matching as a decisions problem

Input: A graph $G = (V, E)$ with weights w_e for $e \in E$ and an objective goal W .

Does there exists a matching with objective value greater than W ?

Output: Either "yes" or "no".

We can now define the class of NP.

Definition 6.17: The class NP

Is the set of all decision problems for which a YES answer for a particular instance can be verified in polytime when provided a certificate.

A certificate can be any additional information to help convince someone of a solution. This should be describable in a compact way (polynomial in the size of the data). Typically the certificate is simply a feasible solution.

Examples:

- All problems in \mathcal{P}
- Integer programming
- TSP
- Binary knapsack
- Maximum independent set
- Subset sum
- Partition
- SAT, k -SAT
- Clique

Thus, to show that a problem is in NP, you must do the following:

1. Describe a format for a certificate to the problem.
2. Show that given such a certificate, it is easy to verify the solution to the problem.

Example 6.18

Integer Linear Programming is in NP. More explicitly, the feasibility question of
"Does there exists an integer vector x that satisfies $Ax \leq b$ "

is in NP.

Although it turns out to be difficult to find such an x or even prove that one exists, this problem is in NP for the following reason: if you are given a particular x and someone claims to you that it is a feasible solution to the problem, then you can easily check if they are correct. In this case, the vector x that you were given is called a certificate.

Note that it is easy to verify if x is a solution to the problem because you just have to

1. Check if x is integer.
2. Use matrix multiplication to check if $Ax \leq b$ holds.

6.4.3. Problem Reductions

We can compare different types of problems by showing that we can use one to solve the other. As an example, we will discuss how sorting can be solved as an integer program.

SOLVING THE SORTING PROBLEM USING INTEGER PROGRAMMING Integer programming (IP) offers a powerful modeling framework capable of representing a wide range of optimization problems, including fundamental ones like the sorting problem. In this section, we will delve into an IP formulation to sort a given list of numbers.

SORTING PROBLEM DEFINITION Given a list $\mathbf{a} = [a_1, a_2, \dots, a_n]$ of n distinct numbers, our goal is to find a permutation $\mathbf{b} = [b_1, b_2, \dots, b_n]$ such that $b_1 \leq b_2 \leq \dots \leq b_n$.

IP FORMULATION To cast the sorting problem as an IP, we introduce binary decision variables:

$$x_{ij} = \begin{cases} 1 & \text{if } a_i \text{ is placed in position } j \\ 0 & \text{otherwise} \end{cases}$$

for $i, j \in \{1, \dots, n\}$.

Additionally, we define:

$$b_j = \sum_{i=1}^n a_i x_{ij}$$

which represents the value in position j of the sorted list.

Objective Function: Since our main goal is to sort the list, we can use a trivial objective, such as:

$$\min \sum_{i=1}^n \sum_{j=1}^n x_{ij}$$

Constraints:

1. Ensure that each number is assigned to exactly one position:

$$\sum_{j=1}^n x_{ij} = 1 \quad \forall i \in \{1, \dots, n\}$$

2. Ensure that each position receives exactly one number:

$$\sum_{i=1}^n x_{ij} = 1 \quad \forall j \in \{1, \dots, n\}$$

3. Define b_j variables:

$$b_j = \sum_{i=1}^n a_i x_{ij} \quad \forall j \in \{1, \dots, n\}$$

4. Ensure the numbers are ordered:

$$b_j \leq b_{j+1} \quad \forall j \in \{1, \dots, n-1\}$$

DISCUSSION This IP formulation aptly captures the sorting problem. However, as before, it's pivotal to understand that sorting through IP is substantially less efficient than employing classical sorting algorithms like QuickSort or MergeSort. This IP representation primarily demonstrates the versatility of integer programming. In real-world scenarios, integer programming would not typically be the first choice for sorting. Nonetheless, understanding this approach paves the way for modeling more intricate combinatorial optimization challenges with IP.

PROBLEM REDUCTIONS Problem reductions are a central concept in the realm of computational complexity theory and algorithm design. Reductions allow us to understand the relative difficulty of problems by transforming or "reducing" one problem to another. In essence, if we can reduce problem A to problem B in polynomial time, then B is at least as hard as A .

Definition 6.19: Reduction

Given two problems \mathcal{A}, \mathcal{B} , we say \mathcal{A} is reduced to \mathcal{B} (and we write $\mathcal{A} \leq \mathcal{B}$ when we can assert that if we can solve \mathcal{B} in polynomial time, then we can also solve \mathcal{A} in polynomial time).

Perhaps more formally, we can

Definition 6.20: Polynomial-time reducible

A problem A is polynomial-time reducible to a problem B , denoted as $A \leq_p B$, if there exists a polynomial-time computable function f such that for every instance i of A :

$$i \text{ is a yes-instance of } A \iff f(i) \text{ is a yes-instance of } B$$

If $A \leq_p B$ and $B \leq_p A$, we say that A and B are polynomial-time equivalent.

6.4.4. Significance of Reductions

1. **Algorithmic Perspective:** If we have an efficient algorithm for solving B and if A can be polynomially reduced to B , then we can solve A efficiently as well.
2. **Complexity Perspective:** If A can be polynomially reduced to B and if B is known to be hard (for instance, NP-hard), then A is hard as well.

6.4.5. Examples of Problem Reductions

6.4.5.1. VERTEX COVER to SET COVER

Problem Descriptions:

VERTEX COVER (VC): Given an undirected graph $G(V, E)$ and an integer k , does there exist a subset of vertices $V' \subseteq V$ such that every edge in E has at least one endpoint in V' , and $|V'| \leq k$?

SET COVER (SC): Given a universe U of n elements, a collection of subsets S_1, S_2, \dots, S_m of U , and an integer k , is there a collection C of k or fewer sets such that the union of the sets in C covers all elements of U ?

Reduction: 1. For each edge $e = (u, v) \in E$ in the graph G , create an element e in the universe U . 2. For each vertex $v \in V$ in G , create a set S_v in SC where S_v contains all edges incident to v . 3. Set k in SC to be the same as k in VC.

The intuition is that if there is a vertex cover of size k , then there is a set cover of size k which covers all edges.

6.4.5.2. 3SAT to 3D MATCHING

Problem Descriptions:

3SAT: Given a Boolean formula ϕ consisting of clauses with exactly 3 literals each, is there a satisfying assignment?

3D MATCHING (3DM): Given three disjoint sets X, Y, Z each of size n and a set $T \subseteq X \times Y \times Z$ of triples, does there exist a perfect matching? A perfect matching M is a subset of T such that each element of $X \cup Y \cup Z$ is contained in exactly one triple of M .

Reduction: Construct the 3DM instance from a 3SAT instance as follows: 1. For each variable x_i , create two elements: x_i and \bar{x}_i in X . 2. For each clause C_j in ϕ , create an element C_j in Y . 3. For each literal in each clause, create a corresponding element in Z . 4. For each clause $C_j = (l_{j1} \vee l_{j2} \vee l_{j3})$, add triples $(l_{j1}, C_j, l_{j1}), (l_{j2}, C_j, l_{j2}),$ and (l_{j3}, C_j, l_{j3}) to T .

The key idea is that a perfect matching in the 3D matching problem corresponds to a satisfying assignment

in the 3SAT problem.

6.4.5.3. Discussion

Both of these reductions establish the NP-hardness of the target problems, assuming the source problems are NP-hard. They provide insights into the relationships between seemingly unrelated computational problems and demonstrate the versatility and importance of reductions in complexity theory.

6.4.6. NP-Hard

The class of problems that are called *NP-Hard* are those that can be used to solve any other problem in the NP class. That is, problem A is NP-Hard provided that for any problem B in NP there is a transformation of problem B that preserves the size of the problem, up to a polynomial factor, into a new problem that problem A can be used to solve.

Here we think of “if problem A could be solved efficiently, then all problems in NP could be solved efficiently”.

More specifically, we assume that we have an oracle for problem A that runs in polynomial time. An oracle is an algorithm that for the problem that returns the solution of the problem in a time polynomial in the input. This oracle can be thought of as a magic computer that gives us the answer to the problem. Thus, we say that problem A is NP-Complete provided that given an oracle for problem A, one can solve any other problem B in NP in polynomial time.

Note: These problems are not necessarily in NP.

6.4.7. NP-Complete

The class of problems that are call *NP-Complete* are those which are in NP and also NP-Hard.

We know of many problems that are NP-Complete. For example, binary integer programming feasibility is NP-Complete. One can show that another problem is NP-complete by

1. showing that it can be used to solve binary integer programming feasibility,
2. showing that the problem is in NP.

The first problem proven to be NP-Complete is called *3-SAT* [1]. 3-SAT is a special case of the *satisfiability problem*. In a satisfiability problem, we have variables X_1, \dots, X_n and we want to assign them values as either `true` or `false`. The problem is described with *AND* operations, denoted as \wedge , with *OR* operations, denoted as \vee , and with *NOT* operations, denoted as \neg . The *AND* operation $X_1 \wedge X_2$ returns `true` if BOTH X_1 and X_2 are true. The *OR* operation $X_1 \vee X_2$ returns `true` if AT LEAST ONE OF X_1 and X_2 are true. Lastly, the *NOT* operation $\neg X_1$ returns the opposite of the value of X_1 .

These can be described in the following table

$$\text{true} \wedge \text{true} = \text{true} \quad (6.1)$$

$$\text{true} \wedge \text{false} = \text{false} \quad (6.2)$$

$$\text{false} \wedge \text{false} = \text{false} \quad (6.3)$$

$$\text{false} \wedge \text{true} = \text{false} \quad (6.4)$$

$$\text{true} \vee \text{true} = \text{true} \quad (6.5)$$

$$\text{true} \vee \text{false} = \text{true} \quad (6.6)$$

$$\text{false} \vee \text{false} = \text{false} \quad (6.7)$$

$$\text{false} \vee \text{true} = \text{true} \quad (6.8)$$

$$\neg \text{true} = \text{false} \quad (6.9)$$

$$\neg \text{false} = \text{true} \quad (6.10)$$

For example, **Missing code here** A *logical expression* is a sequence of logical operations on variables X_1, \dots, X_n , such that

$$(X_1 \wedge \neg X_2 \vee X_3) \wedge (X_1 \vee \neg X_3) \vee (X_1 \wedge X_2 \wedge X_3). \quad (6.11)$$

A *clause* is a logical expression that only contains the operations \vee and \neg and is not nested (with parentheses), such as

$$X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4. \quad (6.12)$$

A fundamental result about logical expressions is that they can always be reduced to a sequence of clauses that are joined by \wedge operations, such as

$$(X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4) \wedge (X_1 \vee X_2 \vee X_3) \wedge (X_2 \vee \neg X_3 \vee \neg X_4 \vee X_5). \quad (6.13)$$

The satisfiability problem takes as input a logical expression in this format and asks if there is an assignment of `true` or `false` to each variable X_i that makes the expression `true`. The 3-SAT problem is a special case where the clauses have only three variables in them.

3-SAT:

NP-Complete

Given a logical expression in n variables where each clause has only 3 variables, decide if there is an assignment to the variables that makes the expression `true`.

Binary Integer Programming:*NP-Complete*

Binary Integer Programming can easily be shown to be in NP, since verifying solutions to BIP can be done by checking a linear system of inequalities.

Furthermore, it can be shown to be NP-Complete since it can be used to solve 3-SAT. That is, given an oracle for BIP, since 3-SAT can be modeled as a BIP, the 3-SAT could be solved in oracle-polynomial time.

6.5 Problems and Algorithms

We will discuss the following concepts:

- Feasible solutions
- Optimal solutions
- Approximate solutions
- Heuristics
- Exact Algorithms
- Approximation Algorithms
- Complexity class relations

6.5.1. Matching Problem

Definition 6.21: Matching

Given a graph $G = (V, E)$, a matching is a subset $E' \subseteq E$ such that no vertex $v \in V$ is contained in more than one edge in E' .

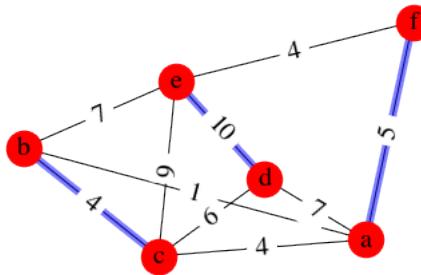
A perfect matching is a matching were every vertex is connected to an edges in E' .

A maximal matching is a matching E' such that there is no matching E'' that strictly contains it.

Figure 6.6: Two possible matchings. On the left, we have a perfect matchings (all nodes are matched). On the right, a feasible matching, but not a perfect matching since not all nodes are matched.

Definition 6.22: Maximum Weight Matching

Given a graph $G = (V, E)$, with associated weights $w_e \geq 0$ for all $e \in E$, a maximum weight matching is a matching that maximizes the sum of the weights in the matching.



© graph-for-matching-maximal⁴

Figure 6.7: graph-for-matching-maximal

6.5.1.1. Greedy Algorithm for Maximal Matching

The greedy algorithm iteratively adds the edge with largest weight that is feasible to add.

Greedy Algorithm for Maximal Matching:

Complexity: $O(|E| \log(|V|))$

1. Begin with an empty graph ($M = \emptyset$)
2. Label the edges in the graph such that $w(e_1) \geq w(e_2) \geq \dots \geq w(e_m)$
3. For $i = 1, \dots, m$
If $M \cup \{e_i\}$ is a valid matching (i.e., no vertex is incident with two edges), then set $M \leftarrow M \cup \{e_i\}$
(i.e., add edge e_i to the graph M)
4. Return M

Theorem 6.23: Greedy algorithm gives a 2-approximation [[Avis83]]

The greedy algorithm finds a 2-approximation of the maximum weighted matching problem. That is, $w(M_{greedy}) \geq \frac{1}{2}w(M^*)$.

⁴graph-for-matching-maximal, from graph-for-matching-maximal. graph-for-matching-maximal, graph-for-matching-maximal.

6.5.1.2. Other algorithms to look at

1. Improved Algorithm [[DRAKE2003211](#)]
2. Blossom Algorithm [Wikipedia](#)

6.5.2. Minimum Spanning Tree

Definition 6.24: Spanning Tree

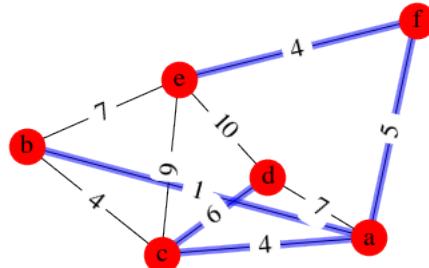
Given a graph $G = (V, E)$, a spanning tree connected, acyclic subgraph T that contains every node in V .

© spanning-tree⁵

Figure 6.8: spanning-tree

Definition 6.25: Max weight spanning tree

Given a graph $G = (V, E)$, with associated weights $w_e \geq 0$ for all $e \in E$, a maximum weight spanning tree is a spanning tree maximizes the sum of the edge weights.



© spanning-tree-MST⁶

Figure 6.9: spanning-tree-MST

Lemma 6.26: Edges and Spanning Trees

Let G be a connected graph with n vertices.

1. T is a spanning tree of G if and only if T has $n - 1$ edges and is connected.
2. Any subgraph S of G with more than $n - 1$ edges contains a cycle.

See Section ?? for integer programming formulations of this problem.

⁵spanning-tree, from [spanning-tree](#). [spanning-tree](#), [spanning-tree](#).

⁶spanning-tree-MST, from [spanning-tree-MST](#). [spanning-tree-MST](#), [spanning-tree-MST](#).

6.5.3. Kruskal's algorithm

Kruskal - for Minimum Spanning tree:

Complexity: $O(|E| \log(|V|))$

1. Begin with an empty tree ($T = \emptyset$)
2. Label the edges in the graph such that $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$
3. For $i = 1, \dots, m$
If $T \cup \{e_i\}$ is acyclic, then set $T \leftarrow T \cup \{e_i\}$
4. Return T

6.5.3.1. Prim's Algorithm

6.5.4. Traveling Salesman Problem

See Section 5.2 for integer programming formulations of this problem. Also, hill climbing algorithms for this problem such as 2-Opt, simulated annealing, and tabu search will be discussed in Section ??.

6.5.4.1. Nearest Neighbor - Construction Heuristic

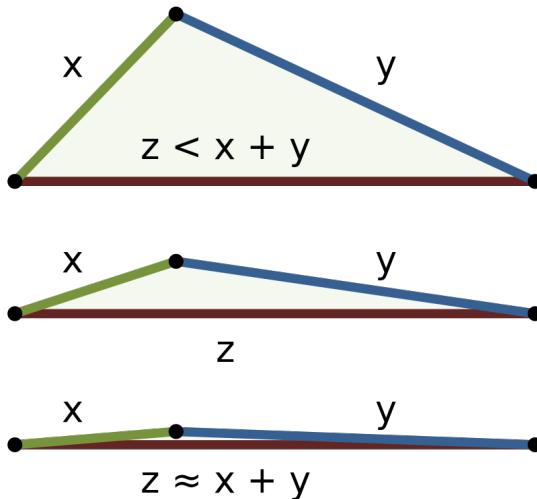
We will discuss heuristics more later in this book. For now, present this construction heuristic as a simple algorithmic example.

Starting from any node, add the edge to the next closest node. Continue this process.

Nearest Neighbor:

Complexity: $O(n^2)$

1. Start from any node (lets call this node 1) and label this as your current node.
2. Pick the next current node as the one that is closest to the current node that has not yet been visited.
3. Repeat step 2 until all nodes are in the tour.



© wiki/File/triangle_inequality.png⁷

Figure 6.10: wiki/File/triangle_inequality.png

6.5.4.2. Double Spanning Tree - 2-Apx

We can use a minimum spanning tree algorithm to find a provably okay solution to the TSP, provided certain properties of the graph are satisfied.

Graphs with nice properties are often easier to handle and typically graphs found in the real world have some nice properties. The *triangle inequality* comes from the idea of a triangle that the sum of the lengths of two sides always are longer than the length of the third side. See Figure 6.10

Definition 6.27: Triangle Inequality on a Graph

A complete, weighted graph G (i.e., a graph that has all possible edges and a weight assigned to each edge) satisfies the triangle inequality provided that for every triple of vertices a, b, c and edges e_{ab}, e_{bc}, e_{ac} , we have that

$$w(e_{ab}) + w(e_{bc}) \geq w(e_{ac}).$$

Let S be the resulting tour and let S^* be an optimal tour. Since the resulting tour is feasible, it will satisfy

$$w(S^*) \leq w(S).$$

But we also know that the weight of a minimum spanning tree T is less than that of the optimal tour, hence

$$w(T) \leq w(S^*).$$

Lastly, due to the triangle inequality we know that

$$w(S) \leq 2w(T),$$

⁷wiki/File/triangle_inequality.png, from wiki/File/triangle_inequality.png. wiki/File/triangle_inequality.png, wiki/File/triangle_inequality.png.

Algorithm 4 Double Spanning Tree

Require: A graph $G = (V, E)$ that satisfies the triangle inequality

Ensure: A tour that is a 2-Apx of the optimal solution

- 1: Compute a minimum spanning tree T of G .
- 2: Double each edge in the minimum spanning tree (i.e., if edge e_{ab} is in T , add edge e_{ba}).
- 3: Compute an Eulerian Tour using these edges.
- 4: Return tour that visits vertices in the order the Eulerian Tour visits them, but without repeating any vertices.

since replacing any edge in the Eulerian tour with a more direct edge only reduces the total weight.

Putting this together, we have

$$w(S) \leq 2w(T) \leq 2w(S^*)$$

and hence, S is a 2-approximation of the optimal solution.

6.5.4.3. Christofides - Approximation Algorithm - (3/2)-Apx

If we combine algorithms for minimum spanning tree and matching, we can find a better approximation algorithm. This is given by Christofides. Again, this is in the case where the graph satisfies the triangle inequality. See Wikipedia - Christofides Algorithm or Ola Svensson Lecture Slides for more detail.

6.6 Resources

Resources

- [MIT Lecture Notes - Big O](#)
- [Youtube! - P versus NP](#)

Resources

[Bubble Sort](#)

- [Wikipedia](#)

Resources

[Kruskal Wikipedia](#)

Resources

Prim's Algorithm

- [Wikipedia](#)
- [TeXample - Figure for min spanning tree](#)

Resources

Nearest Neighbor for TSP [Wikipedia](#)

6.6.1. Advanced - NP Completeness Reductions

Problem: Subset Sum

Instance: A set of n positive integers S and a target positive integer t .

Question: Is there a subset of S that adds up to t ?

Reduction: We'll reduce the problem "3SAT" to "Subset Sum".

Given an instance of 3SAT with m clauses and n variables, we'll create a corresponding instance of Subset Sum as follows:

For each variable, create two positive integers: $2^{(i-1)}$ and 2^i , where i is the index of the variable.

For each clause (x or y or z), create a target integer t equal to $2^{(m+n)} + 2^{(x-1)} + 2^{(y-1)} + 2^{(z-1)}$, where x, y, z are the indices of the variables in the clause.

The set S for Subset Sum will be the n positive integers created for the n variables, plus all the target integers created for the m clauses.

Claim: The 3SAT instance is satisfiable if and only if the corresponding Subset Sum instance has a solution.

Proof:

If the 3SAT instance is satisfiable, then for each clause (x or y or z), at least one of the variables x, y , or z must be set to true. Thus, we can include the corresponding positive integers in our solution for the Subset Sum instance, and the solution will sum up to the target t for that clause.

If the Subset Sum instance has a solution, then for each clause t , there must be a subset of S that sums up to t . We can set the variables corresponding to the positive integers included in the subset to true, and set all other variables to false. This will result in a satisfying assignment for the 3SAT instance.

Since 3SAT is NP-complete, and we have shown that Subset Sum can be reduced to 3SAT in polynomial time, it follows that Subset Sum is also NP-complete.

6.7 Other material for Integer Linear Programming

Recall the problem on lemonade and lemon juice from Chapter ??:

Problem. Say you are a vendor of lemonade and lemon juice. Each unit of lemonade requires 1 lemon and 2 litres of water. Each unit of lemon juice requires 3 lemons and 1 litre of water. Each unit of lemonade gives a profit of \$3. Each unit of lemon juice gives a profit of \$2. You have 6 lemons and 4 litres of water available. How many units of lemonade and lemon juice should you make to maximize profit?

Letting x denote the number of units of lemonade to be made and letting y denote the number of units of lemon juice to be made, the problem could be formulated as the following linear programming problem:

$$\begin{aligned} \max \quad & 3x + 2y \\ \text{s.t.} \quad & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x \geq 0 \\ & y \geq 0. \end{aligned}$$

The problem has a unique optimal solution at $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1.2 \\ 1.6 \end{bmatrix}$ for a profit of 6.8. But this solution requires us to make fractional units of lemonade and lemon juice. What if we require the number of units to be integers? In other words, we want to solve

$$\begin{aligned} \max \quad & 3x + 2y \\ \text{s.t.} \quad & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x \geq 0 \\ & y \geq 0 \\ & x, y \in \mathbb{Z}. \end{aligned}$$

This problem is no longer a linear programming problem. But rather, it is an integer linear programming problem.

A **mixed-integer linear programming problem** is a problem of minimizing or maximizing a linear function subject to finitely many linear constraints such that the number of variables are finite and at least one of which is required to take on integer values.

If all the variables are required to take on integer values, the problem is called a **pure integer linear programming problem** or simply an **integer linear programming problem**. Normally, we assume the problem data to be rational numbers to rule out some pathological cases.

Mixed-integer linear programming problems are in general difficult to solve yet they are too important to ignore because they have a wide range of applications (e.g. transportation planning, crew scheduling, circuit design, resource management etc.) Many solution methods for these problems have been devised and some of them first solve the **linear programming relaxation** of the original problem, which is the problem obtained from the original problem by dropping all the integer requirements on the variables.

Example 6.28

Let (MP) denote the following mixed-integer linear programming problem:

$$\begin{array}{lllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 \geq 1 \\ & -x_1 & - & x_2 & + & 2x_3 \geq 0 \\ & -x_1 & + & 5x_2 & - & x_3 = 3 \\ & x_1, & x_2, & x_3 & \geq 0 \\ & & & x_3 & \in \mathbb{Z}. \end{array}$$

The linear programming relaxation of (MP) is:

$$\begin{array}{lllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 \geq 1 \\ & -x_1 & - & x_2 & + & 2x_3 \geq 0 \\ & -x_1 & + & 5x_2 & - & x_3 = 3 \\ & x_1, & x_2, & x_3 & \geq 0. \end{array}$$

Let (P1) denote the linear programming relaxation of (MP). Observe that the optimal value of (P1) is a lower bound for the optimal value of (MP) since the feasible region of (P1) contains all the feasible solutions to (MP), thus making it possible to find a feasible solution to (P1) with objective function value better than the optimal value of (MP). Hence, if an optimal solution to the linear programming relaxation happens to be a feasible solution to the original problem, then it is also an optimal solution to the original problem. Otherwise, there is an integer variable having a nonintegral value v . What we then do is to create two new subproblems as follows: one requiring the variable to be at most the greatest integer less than v , the other requiring the variable to be at least the smallest integer greater than v . This is the basic idea behind the **branch-and-bound method**. We now illustrate these ideas on (MP).

Solving the linear programming relaxation (P1), we find that $\mathbf{x}' = \begin{bmatrix} 0 \\ \frac{2}{3} \\ \frac{1}{3} \end{bmatrix}$ is an optimal solution to (P1). Note

that \mathbf{x}' is not a feasible solution to (MP) because x'_3 is not an integer. We now create two subproblems (P2) and (P3) such that (P2) is obtained from (P1) by adding the constraint $x_3 \leq \lfloor x'_3 \rfloor$ and (P3) is obtained from (P1) by adding the constraint $x_3 \geq \lceil x'_3 \rceil$. (For a number a , $\lfloor a \rfloor$ denotes the greatest integer at most a and $\lceil a \rceil$ denotes the smallest integer at least a .) Hence, (P2) is the problem

$$\begin{array}{lllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 \geq 1 \\ & -x_1 & - & x_2 & + & 2x_3 \geq 0 \\ & -x_1 & + & 5x_2 & - & x_3 = 3 \\ & & & x_3 & \leq 0 \\ & x_1, & x_2, & x_3 & \geq 0, \end{array}$$

and (P3) is the problem

$$\begin{array}{lllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 + x_2 + x_3 & \geq & 1 \\ & -x_1 - x_2 + 2x_3 & \geq & 0 \\ & -x_1 + 5x_2 - x_3 & = & 3 \\ & & & x_3 & \geq 1 \\ & x_1, x_2, x_3 & \geq & 0. \end{array}$$

Note that any feasible solution to (MP) must be a feasible solution to either (P2) or (P3). Using the help of a solver, one sees that (P2) is infeasible. The problem (P3) has an optimal solution at $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$, which

is also feasible to (MP). Hence, $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$ is an optimal solution to (MP).

We now give a description of the method for a general mixed-integer linear programming problem (MIP). Suppose that (MIP) is a minimization problem and has n variables x_1, \dots, x_n . Let $\mathcal{I} \subseteq \{1, \dots, n\}$ denote the set of indices i such that x_i is required to be an integer in (MIP).

Branch-and-bound method

Input: The problem (MIP).

Steps:

1. Set $\text{bestbound} := \infty$, $\mathbf{x}_{\text{best}}^* := \text{N/A}$, $\text{activeproblems} := \{(LP)\}$ where (LP) denotes the linear programming relaxation of (MIP).
2. If there is no problem in activeproblems , then stop; if $\mathbf{x}_{\text{best}}^* \neq \text{N/A}$, then $\mathbf{x}_{\text{best}}^*$ is an optimal solution; otherwise, (MIP) has no optimal solution.
3. Select a problem P from activeproblems and remove it from activeproblems .
4. Solve P .
 - If P is unbounded, then stop and conclude that (MIP) does not have an optimal solution.
 - If P is infeasible, go to step 2.
 - If P has an optimal solution \mathbf{x}^* , then let z denote the objective function value of \mathbf{x}^* .
5. If $z \geq \text{bestbound}$, go to step 2.
6. If x_i^* is not an integer for some $i \in \mathcal{I}$, then create two subproblems P_1 and P_2 such that P_1 is the problem obtained from P by adding the constraint $x_i \leq \lfloor x_i^* \rfloor$ and P_2 is the problem obtained from P by adding the constraint $x_i \geq \lceil x_i^* \rceil$. Add the problems P_1 and P_2 to activeproblems and go to step 2.
7. Set $\mathbf{x}_{\text{best}}^* = \mathbf{x}^*$, $\text{bestbound} = z$ and go to step 2.

Remarks.

- Throughout the algorithm, `activeproblems` is a set of subproblems remained to be solved. Note that for each problem P in `activeproblems`, P is a linear programming problem and that any feasible solution to P satisfying the integrality requirements is a feasible solution to (MIP).
- x_{best}^* is the feasible solution to (MIP) that has the best objective function value found so far and `bestbound` is its objective function value. It is often called an **incumbent**.
- In practice, how a problem from `activeproblems` is selected in step 3 has an impact on the overall performance. However, there is no general rule for selection that guarantees good performance all the time.
- In step 5, the problem P is discarded since it cannot contain any feasible solution to (MIP) having a better objective function value than x_{best}^* .
- If step 7 is reached, then x^* is a feasible solution to (MIP) having objective function value better than `bestbound`. So it becomes the current best solution.
- It is possible for the algorithm to never terminate. Below is an example for which the algorithm will never stop:

$$\begin{aligned} \min \quad & x_1 \\ \text{s.t. } & x_1 + 2x_2 - 2x_3 = 1 \\ & x_1, x_2, x_3 \geq 0 \\ & x_1, x_2, x_3 \in \mathbb{Z}. \end{aligned}$$

However, it is easy to see that $\mathbf{x}^* = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ is an optimal solution because there is no feasible solution with $x_1 = 0$.

One way to keep track of the progress of the computations is to set up a progress chart with the following headings:

Iter	solved	status	branching	activeproblems	x_{best}^*	bestbound
------	--------	--------	-----------	----------------	---------------------	-----------

In a given iteration, the entry in the **solved** column denotes the subproblem that has been solved with the result in the **status** column. The **branching** column indicates the subproblems created from the solved subproblem with an optimal solution not feasible to (MIP). The entries in the remaining columns contain the latest information in the given iteration. For the example (MP) above, the chart could look like the following:

Iter	solved	status	branching	activeproblems	x_{best}^*	bestbound
1	(P1)	optimal $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{2}{3} \\ \frac{1}{3} \end{bmatrix}$	(P2): $x_3 \leq 0$, (P3): $x_3 \geq 1$	(P2), (P3)	N/A	∞
2	(P2)	infeasible	—	(P3)	N/A	∞

Iter	solved	status	branching	active problems	$\mathbf{x}_{\text{best}}^*$	best bound
3	(P3)	optimal $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ \frac{5}{5} \\ 1 \end{bmatrix}$	—	—	$\begin{bmatrix} 0 \\ \frac{4}{5} \\ \frac{5}{5} \\ 1 \end{bmatrix}$	1

Exercises

- Suppose that (MP) in Example 6.7 above has x_2 required to be an integer as well. Continue with the computations and determine an optimal solution to the modified problem.
- With the help of a solver, determine the optimal value of

$$\begin{aligned} \max \quad & 3x + 2y \\ \text{s.t.} \quad & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x, y \geq 0 \\ & x, y \in \mathbb{Z}. \end{aligned}$$

- Let $\mathbf{A} \in \mathbb{Q}^{m \times n}$ and $\mathbf{b} \in \mathbb{Q}^m$. Let S denote the system

$$\begin{aligned} \mathbf{Ax} &\geq \mathbf{b} \\ \mathbf{x} &\in \mathbb{Z}^n \end{aligned}$$

- Suppose that $\mathbf{d} \in \mathbb{Q}^m$ satisfies $\mathbf{d} \geq 0$ and $\mathbf{d}^T \mathbf{A} \in \mathbb{Z}^n$. Prove that every \mathbf{x} satisfying S also satisfies $\mathbf{d}^T \mathbf{Ax} \geq \lceil \mathbf{d}^T \mathbf{b} \rceil$. (This inequality is known as a **Chvátal-Gomory cutting plane**.)
- Suppose that $\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 5 & 3 \\ 7 & 6 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} 2 \\ 1 \\ 8 \end{bmatrix}$. Show that every \mathbf{x} satisfying S also satisfies $x_1 + x_2 \geq 2$.

Solutions

- An optimal solution to the modified problem is given by $x^* = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.
- An optimal solution is $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$. Thus, the optimal value is 6.

3. a. Since $\mathbf{d} \geq 0$ and $\mathbf{Ax} \geq \mathbf{b}$, we have $\mathbf{d}^T \mathbf{Ax} \geq \mathbf{d}^T \mathbf{b}$. If $\mathbf{d}^T \mathbf{b}$ is an integer, the result follows immediately. Otherwise, note that $\mathbf{d}^T \mathbf{A} \in \mathbb{Z}^n$ and $\mathbf{x} \in \mathbb{Z}^n$ imply that $\mathbf{d}^T \mathbf{Ax}$ is an integer. Thus, $\mathbf{d}^T \mathbf{Ax}$ must be greater than or equal to the least integer greater than $\mathbf{d}^T \mathbf{b}$.

- b. Take $\mathbf{d} = \begin{bmatrix} \frac{1}{9} \\ 0 \\ \frac{1}{9} \end{bmatrix}$ and apply the result in the previous part.

6.7.1. Other discrete problems

6.7.2. Assignment Problem and the Hungarian Algorithm

Assignment Problem:

Polynomial time (P)

$$\begin{aligned}
 & \min \langle C, X \rangle \\
 \text{s.t. } & \sum_i X_{ij} = 1 \text{ for all } j \\
 & \sum_j X_{ij} = 1 \text{ for all } i \\
 & X_{ij} \in \{0, 1\} \text{ for all } i = 1, \dots, n, j = 1, \dots, m
 \end{aligned} \tag{6.1}$$

This problem is efficiently solved by the Hungarian Algorithm.

6.7.3. History of Computation in Combinatorial Optimization

Book: Computing in Combinatorial Optimization by William Cook, 2019

7. Heuristics for TSP

In this section we will show how different heuristics can find good solutions to TSP. For convenience, we will focus on the *symmetric TSP* problem. That is, the distance d_{ij} from node i to node j is the same as the distance d_{ji} from node j to node i .

There are two general types of heuristics: construction heuristics and improvement heuristics. We will first discuss a few construction heuristics for TSP.

Then we will demonstrate three types of metaheuristics for improvement- Hill Climbing, Tabu Search, and Simulated Annealing. These are called *metaheuristics* because they are a general framework for a heuristic that can be applied to many different types of settings. Furthermore, Tabu Search and Simulated Annealing have parameters that can be adjusted to try to find better solutions.

7.1 Construction Heuristics

7.1.1. Random Solution

TSP is convenient in that choosing a random ordering of the nodes creates a feasible solution. It may not be a very good one, but it is at least a solution.

Random Construction:

Complexity: $O(n)$

For $i = 1, \dots, n$, randomly choose a node not yet in the tour and place it at the end of the tour.

7.1.2. Nearest Neighbor

Starting from any node, add the edge to the next closest node. Continue this process.

Nearest Neighbor:

Complexity: $O(n^2)$

1. Start from any node (lets call this node 1) and label this as your current node.
2. Pick the next current node as the one that is closest to the current node that has not yet been visited.
3. Repeat step 2 until all nodes are in the tour.

7.1.3. Insertion Method

Algorithm 5 Insertion Method

- 1: **Input:** Set of nodes, distance between nodes, etc.
 - 2: **Output:** Constructed tour.
 - 3: **Complexity:** $O(n^2)$
 - 4: Start from any 3 nodes (let's call the first one node 1) and label this as your current node.
 - 5: **while** there are unvisited nodes **do**
 - 6: Pick the next current node as the one closest to the current node that hasn't been visited yet.
 - 7: **return** the constructed tour.
-

7.2 Improvement Heuristics

There are many ways to generate improving steps. The key features of improving step to consider are

- What is the complexity of computing this improving step?
- How good this this improving step?

We will mention ways to find neighbors of a current solution for TSP. If the neighbor has a better objective value, the moving to this neighbor will be an improving step.

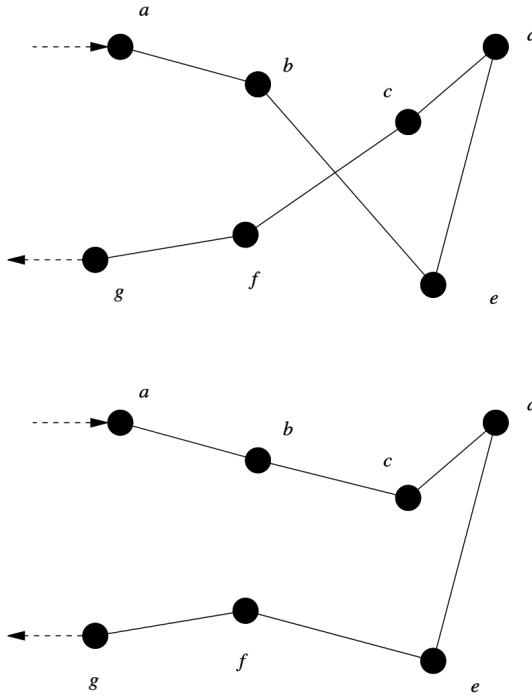
7.2.1. 2-Opt (Subtour Reversal)

We will assume that all tours start and end with then node 1.

2-Opt (Subtour reversal):

Input a tour $1 \rightarrow \dots \rightarrow 1$.

1. Pick distinct nodes $i, j \neq 1$.



© wiki/File/2-opt_wiki.png¹

Figure 7.1: wiki/File/2-opt_wiki.png

2. Let s, t and x_1, \dots, x_k be nodes in the tour such that it can be written as

$$1 \rightarrow \dots \rightarrow s \rightarrow i \rightarrow x_1 \rightarrow \dots \rightarrow x_k \rightarrow j \rightarrow t \rightarrow \dots \rightarrow 1.$$

3. Consider the subtour reversal

$$1 \rightarrow \dots \rightarrow s \rightarrow j \rightarrow x_k \rightarrow \dots \rightarrow x_1 \rightarrow i \rightarrow t \rightarrow \dots \rightarrow 1.$$

Thus, we reverse the order of $i \rightarrow x_1 \rightarrow \dots \rightarrow x_k \rightarrow j$.

4. In this process, we

- deleted the edges (s, i) and (j, t)
- added the edges (s, j) and (i, t)

Pictorially, this looks like the following

Figure 7.1

Computationally, we need to consider the costs on the edges of a graph.....

See [Englert2014] for an analysis of performance of this improvement.

¹wiki/File/2-opt_wiki.png, from wiki/File/2-opt_wiki.png. wiki/File/2-opt_wiki.png, wiki/File/2-opt_wiki.png.

7.2.2. 3-Opt

7.2.3. k -Opt

This is a generalization of 2-Opt and 3-Opt.

7.3 Meta-Heuristics

7.3.1. Hill Climbing (2-Opt for TSP)

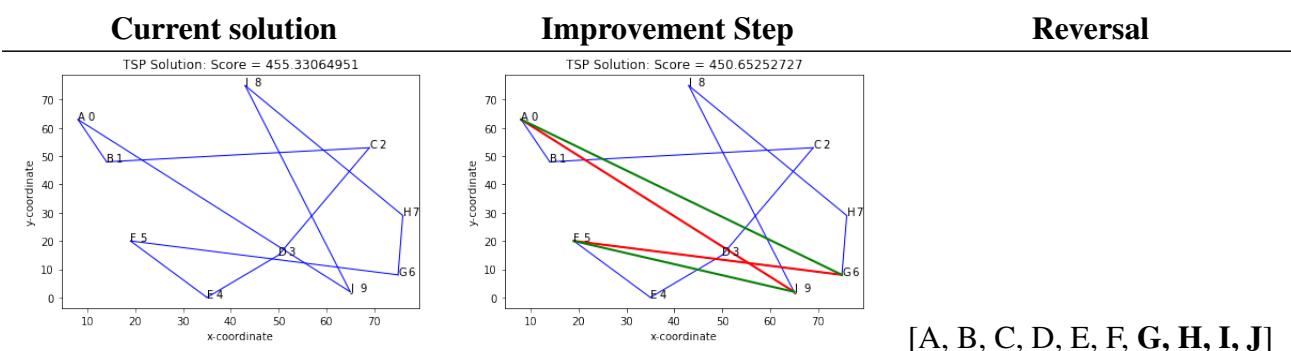
The *Hill Climbing* algorithm finds an improving neighboring solution and climbs in that direction. It continues this process until there is no other neighbor that is improving.

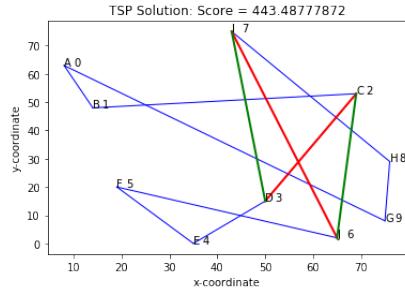
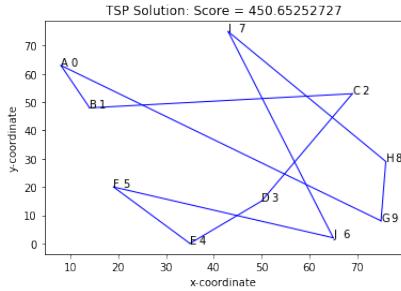
In the context of TSP, we will consider 2-Opt improving moves and the Hill Climbing algorithm for TSP in this case is referred to as the 2-Opt algorithm (also known as the Subtour Reversal Algorithm).

Algorithm 6 Hill Climbing

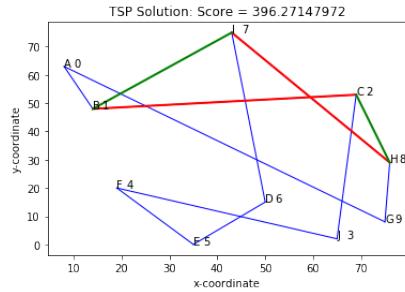
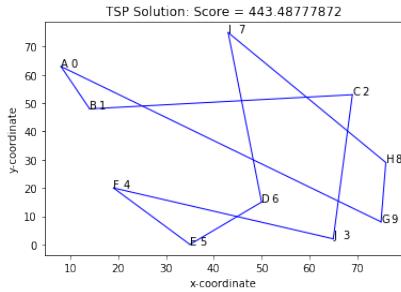
- 1: **Input:** Initial feasible solution, etc.
 - 2: **Output:** Best solution found.
 - 3: Start with an initial feasible solution, label it as the current solution.
 - 4: **repeat**
 - 5: List all neighbors of the current solution.
 - 6: Evaluate all neighbors to find the best one.
 - 7: **if** no neighbor is better **then**
 - 8: **break**
 - 9: Move to the best neighbor.
 - 10: **until** no better neighbor is found
 - 11: **return** the current solution.
-

Here is an example on the TSP problem with 2-Opt swaps:

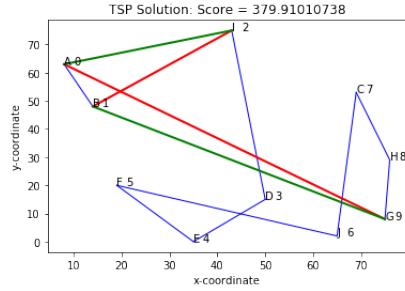
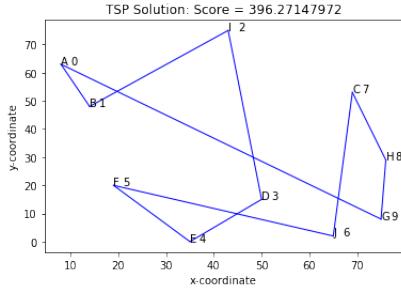




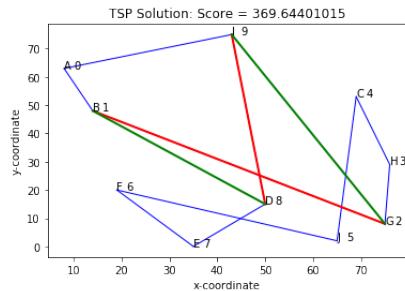
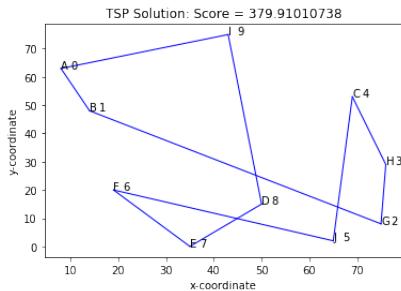
[A, B, C, D, E, F, J, I, H, G]



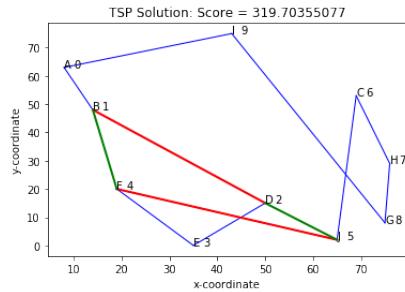
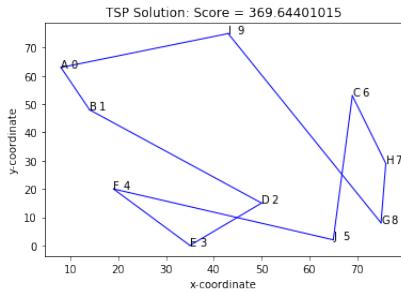
[A, B, C, J, F, E, D, I, H, G]



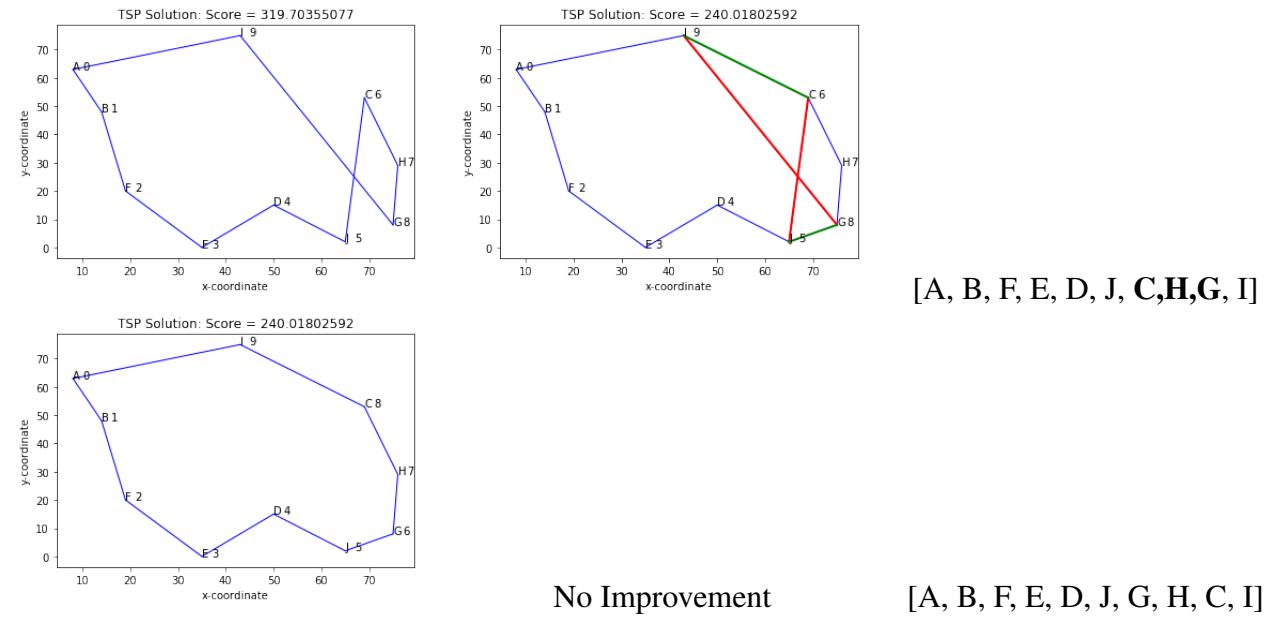
[A, B, I, D, E, F, J, C, H, G]



[A, B, G, H, C, J, F, E, D, I]



[A, B, D, E, F, J, C, H, G, I]



7.3.2. Simulated Annealing

Here is a great python package for TSP Simulated annealing: <https://pypi.org/project/satsp/>.

Simulated annealing is a randomized heuristic that randomly decides when to accept non-improving moves. For this, we use what is referred to as a *temperature schedule*. The temperature schedule guides a parameter T that goes into deciding the probability of accepting a non-improving move.

A typically temperature schedule starts at a value $T = 0.2 \times Z_c$, where Z_c is the objective value of an initial feasible solution. Then the temperature is decreased over time to smaller values.

Temperature schedule example:

- $T_1 = 0.2Z_c$
- $T_2 = 0.8T_1$
- $T_3 = 0.8T_2$
- $T_4 = 0.8T_3$
- $T_5 = 0.8T_4$

For instance, we could choose to run the algorithm for 10 iterations at each temperature value. The Simulated Annealing algorithm is the following:

Figure 7.2

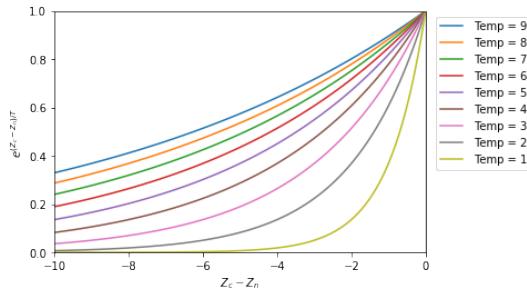
²[simulated_annealing_temperatures.png](#), [simulated_annealing_temperatures.png](#), [simulated_annealing_temperatures.png](#), [simulated_annealing_temperatures.png](#), [simulated_annealing_temperatures.png](#).

Algorithm 7 Simulated Annealing Outline [minimization version]

```

1: Input: Initial feasible solution, temperature schedule, etc.
2: Output: Best found solution during the algorithm.
3: Start with an initial feasible solution, label it as the current solution, and compute its objective value
    $Z_c$ .
4: while iterations left in the schedule do
5:   Select a neighbor of the current solution and compute its objective value  $Z_n$ .
6:   if  $Z_n < Z_c$  then
7:     accept the move and set the neighbor as the current solution.
8:   else
9:     Compute  $p = e^{\frac{Z_c - Z_n}{T}}$ 
10:    Generate a random number  $x \in [0, 1]$  from the computer.
11:    if  $x < p$  then
12:      accept the move
13:    else
14:      reject the move and stay at the current solution.
15:   Update the temperature  $T$ .
16: return the best found solution.

```



© simulated_annealing_temperatures.png²
Figure 7.2: simulated_annealing_temperatures.png

7.3.3. Tabu Search

Tabu Search Outline:

[minimization version]

1. Initialize a *Tabu List* as an empty set: $\text{Tabu} = \{\}$.
2. Start with an initial feasible solution, label it as the current solution.
3. List all neighbors of the current solution.
4. Choose the best neighbor that is not tabu to move too (the move should not be restricted by the

set Tabu.)

5. Add moves to the Tabu List.
6. If the Tabu List is longer than its designated maximal size S , then remove old moves in the list until it reaches the size S .
7. If no object improvement has been seen for K steps, then Stop.
8. Otherwise, Go to Step 3 and continue.

7.3.4. Genetic Algorithms

Genetic algorithms start with a set of possible solutions, and then slowly mutate them to better solutions. See Scikit-opt for an implementation for the TSP.

[Video explaining a genetic algorithm for TSP](#)

7.3.5. Greedy randomized adaptive search procedure (GRASP)

We currently do not cover this topic.

[Wikipedia - GRASP](#)

For an in depth (and recent) book, check out Optimization by GRASP Greedy Randomized Adaptive Search Procedures Authors: Resende, Mauricio, Ribeiro, Celso C..

7.3.6. Ant Colony Optimization

[Wikipedia - Ant Colony Optimization](#)

7.4 Computational Comparisons

Notice how the heuristics are generally faster and provide reasonable solutions, but the solvers provide the best solutions. This is a trade off to consider when deciding how fast you need a solution and how good of a solution it is that you actually need.

Table 7.2: Instance with 5 nodes

Algorithm	Value	Time (seconds)	Memory
Nearest Neighbor	494	0.000065	2.172 KiB
Farthest Insertion	494	0.000057	1.781 KiB
Simulated Annealing	494	0.000600	162.156 KiB
Math Programming Cbc	494.0	0.091290	1.460 MiB
Math Programming Gurobi	494.0	0.006610	78.797 KiB

Table 7.3: Instance with 20 nodes

Algorithm	Value	Time (seconds)	Memory
Nearest Neighbor	790	0.000162	6.406 KiB
Farthest Insertion	791	0.000128	2.734 KiB
Simulated Annealing	777	0.007818	2.601 MiB
Math Programming Cbc	773.0	2.738521	607.961 KiB
Math Programming Gurobi	773.0	0.238488	717.133 KiB

Table 7.4: Instance with 40 nodes

Algorithm	Value	Time (seconds)	Memory
Nearest Neighbor	1216	0.000288	15.141 KiB
Farthest Insertion	1281	0.000286	3.969 KiB
Simulated Annealing	1227	0.047512	10.387 MiB
Math Programming Cbc	1088.0	6.292632	2.111 MiB
Math Programming Gurobi	1088.0	1.349253	2.520 MiB

7.4.1. VRP - Clark Wright Algorithm

Resources and References

Resources

- Amazing video covering all TSP and topics in this section
- Interactive tutorial of TSP algorithms
- TSP Simulated Annealing Video with fun music.
- VRP Heuristic Approach Lecture by Flipkart Delivery Hub
- <https://github.com/Gurobi/pres-mipheur> - Gurobi coded heuristics for TSP with comparison.

8. Forecasting and Stochastic Programming

9. Decomposition Methods

A. Linear Algebra

A.1 Contributors



Champions of Access to Knowledge



OPEN TEXT

All digital forms of access to our high-quality open texts are entirely FREE! All content is reviewed for excellence and is wholly adaptable; custom editions are produced by Lyryx for those adopting Lyryx assessment. Access to the original source files is also open to anyone!



ONLINE ASSESSMENT

We have been developing superior online formative assessment for more than 15 years. Our questions are continuously adapted with the content and reviewed for quality and sound pedagogy. To enhance learning, students receive immediate personalized feedback. Student grade reports and performance statistics are also provided.



SUPPORT

Access to our in-house support team is available 7 days/week to provide prompt resolution to both student and instructor inquiries. In addition, we work one-on-one with instructors to provide a comprehensive system, customized for their course. This can include adapting the text, managing multiple sections, and more!

¹This book was not produced by Lyryx, but this book has made substantial use of their open source material. We leave this page in here as a tribute to Lyryx for sharing their content.



INSTRUCTOR SUPPLEMENTS

Additional instructor resources are also freely accessible. Product dependent, these supplements include: full sets of adaptable slides and lecture notes, solutions manuals, and multiple choice question banks with an exam building tool.

Contact Lyryx Today!

info@lyryx.com



BE A CHAMPION OF OER!

Contribute suggestions for improvements, new content, or errata:

A new topic

A new example

An interesting new question

A new or better proof to an existing theorem

Any other suggestions to improve the material

Contact Lyryx at info@lyryx.com with your ideas.

CONTRIBUTIONS

Ilijas Farah, York University

Ken Kuttler, Brigham Young University

Lyryx Learning Team

Foundations of Applied Mathematics

<https://github.com/Foundations-of-Applied-Mathematics>
CONTRIBUTIONS

List of Contributors

E. Evans
Brigham Young University
R. Evans
Brigham Young University
J. Grout
Drake University
J. Humpherys
Brigham Young University
T. Jarvis
Brigham Young University
J. Whitehead
Brigham Young University
J. Adams
Brigham Young University
J. Bejarano
Brigham Young University
Z. Boyd
Brigham Young University
M. Brown
Brigham Young University
A. Carr
Brigham Young University
C. Carter
Brigham Young University
T. Christensen
Brigham Young University
M. Cook
Brigham Young University
R. Dorff
Brigham Young University
B. Ehlert
Brigham Young University
M. Fabiano
Brigham Young University
K. Finlinson
Brigham Young University

J. Fisher
Brigham Young University
R. Flores
Brigham Young University
R. Fowers
Brigham Young University
A. Frandsen
Brigham Young University
R. Fuhriman
Brigham Young University
S. Giddens
Brigham Young University
C. Gigena
Brigham Young University
M. Graham
Brigham Young University
F. Glines
Brigham Young University
C. Glover
Brigham Young University
M. Goodwin
Brigham Young University
R. Grout
Brigham Young University
D. Grundvig
Brigham Young University
E. Hannesson
Brigham Young University
J. Hendricks
Brigham Young University
A. Henriksen
Brigham Young University
I. Henriksen
Brigham Young University
C. Hettinger
Brigham Young University

S. Horst	H. Ringer
<i>Brigham Young University</i>	<i>Brigham Young University</i>
K. Jacobson	C. Robertson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Leete	M. Russell
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Lytle	R. Sandberg
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. McMurray	C. Sawyer
<i>Brigham Young University</i>	<i>Brigham Young University</i>
S. McQuarrie	M. Stauffer
<i>Brigham Young University</i>	<i>Brigham Young University</i>
D. Miller	J. Stewart
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Morrise	S. Suggs
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Morrise	A. Tate
<i>Brigham Young University</i>	<i>Brigham Young University</i>
A. Morrow	T. Thompson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Murray	M. Victors
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Nelson	J. Webb
<i>Brigham Young University</i>	<i>Brigham Young University</i>
E. Parkinson	R. Webb
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Probst	J. West
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Proudfoot	A. Zaitzeff
<i>Brigham Young University</i>	<i>Brigham Young University</i>
D. Reber	
<i>Brigham Young University</i>	

This project is funded in part by the National Science Foundation, grant no. TUES Phase II DUE-1323785.

A.1.1. Graph Theory

Chapter on Graph Theory adapted from: CC-BY-SA 3.0 Math in Society A survey of mathematics for the liberal arts major Math in Society is a free, open textbook. This book is a survey of contemporary mathematical topics, most non-algebraic, appropriate for a college-level quantitative literacy topics course for liberal arts majors. The text is designed so that most chapters are independent, allowing the instructor to choose a selection of topics to be covered. Emphasis is placed on the applicability of the mathematics. Core material for each topic is covered in the main text, with additional depth available through exploration exercises appropriate for in-class, group, or individual investigation. This book is appropriate for Washington State Community Colleges' Math 107.

The current version is 2.5, released Dec 2017. <http://www.opentextbookstore.com/mathinsociety/2.5/GraphTheory.pdf>

Communicated by Tricia Muldoon Brown, Ph.D. Associate Professor of Mathematics Georgia Southern University Armstrong Campus Savannah, GA 31419 <http://math.armstrong.edu/faculty/brown/MATH1001.html>

