

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 Poirrier, Douglas Bish, Diego Moran

Version Compilation date: February 1, 2024

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
3 Modeling: Linear Programming	21
3.1 Modeling and Assumptions in Linear Programming	22
3.1.1 General models	23
3.1.2 Assumptions	24
3.2 Examples	25
3.2.1 Knapsack Problem	31
3.2.2 Capital Investment	31
3.2.3 Work Scheduling	31
3.2.4 Assignment Problem	32
3.2.5 Multi period Models	33
3.2.5.1 Production Planning	33
3.2.5.2 Crop Planning	33
3.2.6 Mixing Problems	33
3.2.7 Financial Planning	33
3.2.8 Network Flow	34
3.2.8.1 Graphs	34
3.2.8.2 Maximum Flow Problem	35
3.2.8.3 Minimum Cost Network Flow	37
3.2.9 Multi-Commodity Network Flow	38

6 ■ CONTENTS

3.3	Modeling Tricks	39
3.3.1	Maximizing a minimum	39
3.4	Other examples	40
4	Graphically Solving Linear Programs	41
4.1	Nonempty and Bounded Problem	41
4.2	Infinitely Many Optimal Solutions	45
4.3	Problems with No Solution	48
4.4	Problems with Unbounded Feasible Regions	50
4.5	Formal Mathematical Statements	55
4.6	Graphical example	60
	Exercises	63
	Solutions	64
5	Software - Excel	67
5.0.1	Excel Solver	67
5.0.2	Videos	67
5.0.3	Links	67
6	Software - Python	69
6.1	Installing and Managing Python	69
6.2	NumPy Visual Guide	74
6.3	Plot Customization and Matplotlib Syntax Guide	77
6.4	Networkx - A Python Graph Algorithms Package	84
6.5	PuLP - An Optimization Modeling Tool for Python	84
6.5.1	Installation	85
6.5.2	Example Problem	85
6.5.2.1	Product Mix Problem	85
6.5.3	Things we can do	86
6.5.3.1	Exploring the variables	87
6.5.3.2	Other things you can do	88
6.5.4	Common issue	89
6.5.4.1	Transportation Problem	89
6.5.4.2	Optimization with PuLP	90
6.5.4.3	Optimization with PuLP: Round 2!	91
6.5.5	Changing details of the problem	93
6.5.6	Changing Constraint Coefficients	95
6.6	Multi Objective Optimization with PuLP	95
6.6.0.1	Transportation Problem	95
6.6.0.2	Initial Optimization with PuLP	96
6.6.1	Creating the Pareto Efficient Frontier	98
6.7	Comments	100
6.8	Jupyter Notebooks	100
6.9	Reading and Writing	101

6.10 Python Crash Course	101
6.11 Gurobi	101
6.11.1 Introductory Gurobi Examples	101
6.12 Plots, Pandas, and Geopandas	102
6.12.1 Matplotlib.pyplot	102
6.12.2 Pandas	102
6.12.3 Geopandas	102
6.13 Google OR Tools	102
7 CASE STUDY - Designing a campground - Simplex method	103
7.1 DESIGNING A CAMPGROUND - SIMPLEX METHOD	103
7.1.1 Case Study Description - Campground	103
7.1.2 References	104
7.1.3 Solution Approach - Two Variables	104
7.1.4 Assumptions made about linear programming problem	105
7.1.5 Graphical Simplex solution procedure	105
7.1.6 Stating the Solution	108
7.1.7 Refinements to the graphical solution	109
7.1.8 Solution Approach - Using Excel	115
8 Simplex Method	119
8.1 The Simplex Method	122
8.1.1 Pivoting	126
8.1.2 Termination and Reading the Dictionary	128
8.1.3 Exercises	129
8.1.4 2.5 Exercises	131
8.2 Finding Feasible Basis	132
9 Duality	137
9.1 The Dual of Linear Program	138
9.2 Linear programming duality	143
9.2.1 The dual problem	146
Exercises	148
Solutions	148
10 Sensitivity Analysis	151
11 Multi-Objective Optimization	153
11.1 Multi Objective Optimization and The Pareto Frontier	153
11.2 What points will the Scalarization method find if we vary λ ?	159
11.3 Political Redistricting [3]	159
11.4 Portfolio Optimization [5]	159
11.5 Simulated Portfolio Optimization based on Efficient Frontier	160
11.6 Aircraft Design [1]	160
11.7 Vehicle Dynamics [4]	161

11.8 Sustainable Constriction [2]	161
11.9 References	161

II Discrete Algorithms **163**

12 Graph Algorithms 165	
12.1 Graph Theory and Network Flows	165
12.2 Graphs	165
12.2.1 Drawing Graphs	165
12.3 Definitions	168
12.4 Shortest Path	171
12.5 Spanning Trees	180
12.6 Exercise Answers	184
12.7 Prim's Algorithm	186
12.8 Additional Exercises	186
12.8.1 Notes	195

III Integer Programming **197**

13 Integer Programming Formulations 199	
13.1 Knapsack Problem	199
13.2 Capital Budgeting	202
13.3 Set Covering	205
13.3.1 Covering (Generalizing Set Cover)	209
13.4 Assignment Problem	209
13.5 Facility Location	211
13.5.1 Capacitated Facility Location	212
13.5.2 Uncapacitated Facility Location	214
13.6 Graph Coloring	216
13.7 Basic Modeling Tricks - Using Binary Variables	218
13.7.1 Big M constraints - Activating/Deactivating Inequalities	220
13.7.2 Either Or Constraints	221
13.7.3 If then implications - opposite direction	221
13.7.4 Multi Term Disjunction with application to 2D packing	224
13.7.4.1 Strip Packing Problem	224
13.7.5 SOS1 Constraints	227
13.7.6 SOS2 Constraints	227
13.7.7 Piecewise linear functions with SOS2 constraint	228
13.7.7.1 SOS2 with binary variables	230
13.7.8 Maximizing a minimum	230
13.7.9 Relaxing (nonlinear) equality constraints	231
13.7.10 Exact absolute value	231

13.7.10.1 Exact 1 -norm	231
13.7.10.2 Maximum	232
13.8 Network Flow	233
13.8.1 Maximum flow	233
13.8.2 Minimum Cost Network Flow	234
13.8.3 Multi-Commodity Minimum Cost Network Flow with Integrality Constraints	235
13.9 Job Shop Scheduling	238
13.9.1 JSSP Components	238
13.9.2 Mathematical Model	240
13.9.3 Job Shop Scheduling Variations	242
13.10 Quadratic Assignment Problem (QAP)	243
13.11 Generalized Assignment Problem (GAP)	245
13.11.1 In special cases	245
13.11.2 Explanation of definition	245
13.12 Other material	246
13.12.1 Binary reformulation of integer variables	246
13.13 Literature and Resources	248
13.14 MIP Solvers and Modeling Tools	249
13.14.1 Tools for Solving Job Shop Scheduling Problems	249
14 Algorithms to Solve Integer Programs	251
14.1 Foundational Principle - LP is a relaxation for IP	252
14.1.1 Rounding LP Solution can be bad!	253
14.1.2 Rounding LP solution can be infeasible!	254
14.2 Branch and Bound	254
14.2.1 Binary Integer Programming	254
14.2.2 Branch and bound on general integer variables	256
14.3 Cutting Planes	259
14.3.1 Cover Inequalities	259
14.3.2 Chvátal Cuts	260
14.3.3 Cutting Planes Procedure	261
14.3.4 Gomory Cuts	264
14.4 Interpreting Output Information and Progress	266
14.5 Branching Decision Rules in Integer Programming	267
14.6 Decomposition Methods - Advanced approaches to integer programming	269
14.6.1 Lagrangian Relaxation	269
14.6.2 Dantzig-Wolfe Reformulation and Column Generation	270
14.6.3 Benders Decomposition	271
14.7 Literature and Resources	272
15 Exponential Size Formulations	273
15.1 Cutting Stock	273
15.1.1 Pattern formulation	276
15.1.2 Column Generation	278

15.1.3	Cutting Stock - Multiple widths	279
15.2	Traveling Salesman Problem	280
15.2.1	Miller Tucker Zemlin (MTZ) Model	282
15.2.2	Dantzig-Fulkerson-Johnson (DFJ) Model	289
15.2.3	Traveling Salesman Problem - Branching Solution	292
15.2.4	Traveling Salesman Problem Variants	292
15.2.4.1	Many salespersons (m-TSP)	292
15.2.4.2	TSP with order variants	294
15.2.5	Multi vehicle model with capacities	294
15.3	TSP with Time Windows	295
15.4	Case study - Traveling Salesman Problem Applications for Supply Chain Optimization	296
15.5	Vehicle Routing Problem (VRP)	299
15.5.1	The Clarke-Wright Saving Algorithm for VRP	300
15.5.2	Concept	300
15.5.3	Saving Calculation	300
15.5.4	Algorithm Steps	300
15.5.5	Advantages and Limitations	301
15.6	Tools to solve VRP	301
15.7	Literature and other notes	303
15.7.1	TSP In Excel	303
15.7.2	Resources	303

16 Algorithms and Complexity 305

16.1	Big-O Notation	305
16.2	Algorithms - Example with Bubble Sort	309
16.2.1	Sorting	309
16.3	Problem, instance, size	314
16.3.1	Problem, instance	314
16.3.2	Format and examples of problems/instances	314
16.3.3	Size of an instance	314
16.4	Complexity Classes	315
16.4.1	P	316
16.4.2	NP	317
16.4.3	Problem Reductions	318
16.4.4	Significance of Reductions	320
16.4.5	Examples of Problem Reductions	320
16.4.5.1	VERTEX COVER to SET COVER	320
16.4.5.2	3SAT to 3D MATCHING	320
16.4.5.3	Discussion	321
16.4.6	NP-Hard	321
16.4.7	NP-Complete	321
16.5	Problems and Algorithms	323
16.5.1	Matching Problem	323
16.5.1.1	Greedy Algorithm for Maximal Matching	324

16.5.1.2 Other algorithms to look at	325
16.5.2 Minimum Spanning Tree	325
16.5.3 Kruskal's algorithm	326
16.5.3.1 Prim's Algorithm	326
16.5.4 Traveling Salesman Problem	326
16.5.4.1 Nearest Neighbor - Construction Heuristic	326
16.5.4.2 Double Spanning Tree - 2-Apx	327
16.5.4.3 Christofides - Approximation Algorithm - (3/2)-Apx	328
16.6 Resources	328
16.6.1 Advanced - NP Completeness Reductions	329
16.7 Other material for Integer Linear Programming	330
Exercises	334
Solutions	335
16.7.1 Other discrete problems	335
16.7.2 Assignment Problem and the Hungarian Algorithm	335
16.7.3 History of Computation in Combinatorial Optimization	336
17 Heuristics for TSP	337
17.1 Construction Heuristics	337
17.1.1 Random Solution	337
17.1.2 Nearest Neighbor	337
17.1.3 Insertion Method	338
17.2 Improvement Heuristics	338
17.2.1 2-Opt (Subtour Reversal)	338
17.2.2 3-Opt	340
17.2.3 k -Opt	340
17.3 Meta-Heuristics	340
17.3.1 Hill Climbing (2-Opt for TSP)	340
17.3.2 Simulated Annealing	342
17.3.3 Tabu Search	343
17.3.4 Genetic Algorithms	344
17.3.5 Greedy randomized adaptive search procedure (GRASP)	344
17.3.6 Ant Colony Optimization	344
17.4 Computational Comparisons	344
17.4.1 VRP - Clark Wright Algorithm	345
IV Nonlinear Programming	347
18 Nonlinear Programming (NLP)	349
18.1 Definitions and Theorems	349
18.1.1 Calculus: Derivatives	351
18.1.2 Calculus: Second derivatives	352
18.1.3 Multivariate Calculus Examples	353

18.1.4 Taylor's Theorem	355
18.1.5 Constrained Minimization and the KKT Conditions	357
19 NLP Algorithms	361
19.1 Algorithms Introduction	361
19.2 1-Dimensional Algorithms	361
19.2.1 Golden Search Method - Derivative Free Algorithm	364
19.2.1.1 Example:	365
19.2.2 Bisection Method - 1st Order Method (using Derivative)	365
19.2.2.1 Minimization Interpretation	365
19.2.2.2 Root finding Interpretation	366
19.2.3 Gradient Descent - 1st Order Method (using Derivative)	366
19.2.4 Newton's Method - 2nd Order Method (using Derivative and Hessian)	369
19.3 Multi-Variate Unconstrained Optimizaiton	370
19.3.1 Descent Methods - Unconstrained Optimization - Gradient, Newton	370
19.3.1.1 Choice of α_t	371
19.3.1.2 Choice of d_t using $\nabla f(x)$	371
19.3.2 Stochastic Gradient Descent - The mother of all algorithms.	371
19.3.3 Neural Networks	373
19.3.4 Choice of Δ_k using the hessian $\nabla^2 f(x)$	373
19.4 Constrained Convex Nonlinear Programming	373
19.4.1 Barrier Method	374
20 Computational Issues with NLP	377
20.1 Irrational Solutions	377
20.2 Discrete Solutions	377
20.3 Convex NLP Harder than LP	377
20.4 NLP is harder than IP	378
20.5 Resources	379
21 Fairness in Algorithms	381
22 One-dimensional Optimization	383
23 Gradient Descent Methods	393
A Linear Algebra	399
A.1 Contributors	399
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 desired. 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 guarantees 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 guarantees 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

MIXED INTEGER NONLINEAR PROGRAMMING

- Mixed-Integer Nonlinear Optimization Pietro Belotti, Christian Kirches, Sven Leyffer, Jeff Linderoth, Jim Luedtke, and Ashutosh Mahajan

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. $\{1, 2\} \cup \{3, 5\} = \{1, 2, 3, 5\}$)

8 ■ Resources and Notation

- \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 $[6, 2, -3, -11]$)
 - \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

- *Identify reasons for studying operations research*
- *Define "Mathematical Programming"*
- *Learn about different applications of the tools in this book*
- *Explore the different types of optimization models and what types we will see in this book.*

2.1 Why study operations research?

2.2 What is Mathematical Programming?

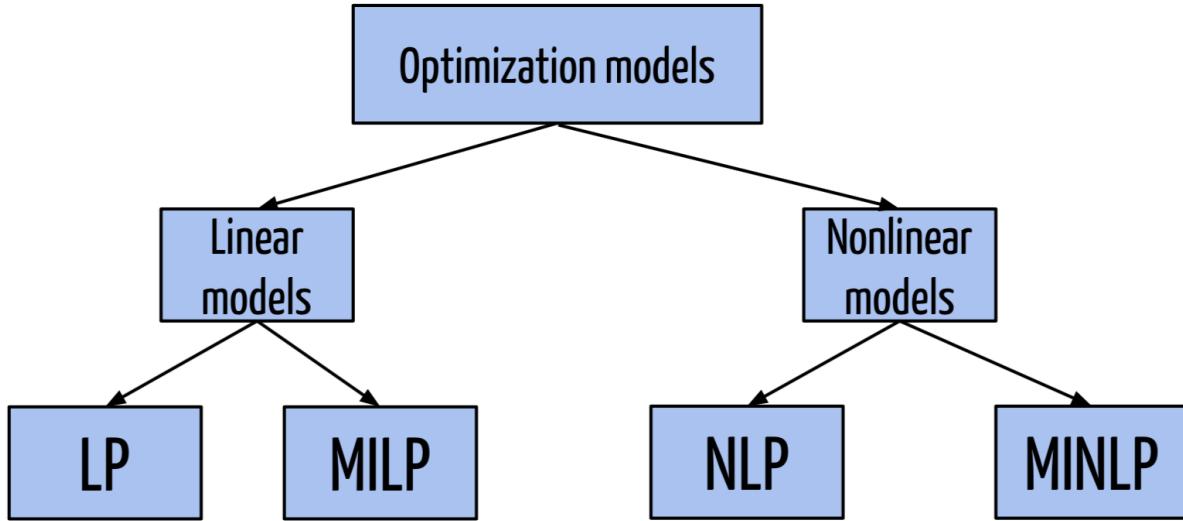
2.3 Applications

2.4 Types of Optimization problems

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).

Along with each problem class, we will associate a complexity class for the general version of the problem. See chapter 16 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.



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

2.5 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

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

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

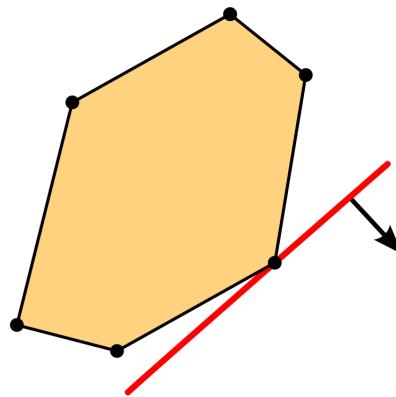
Linear Programming (LP) Standard Form:

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

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



© wiki/File/linear-programming.png²

Figure 2.2: Linear programming constraints and objective.

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.

2.6 Mixed-Integer Linear Programming (MILP)

Mixed-integer linear programming will be the focus of Sections 13, 15, 14, 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):

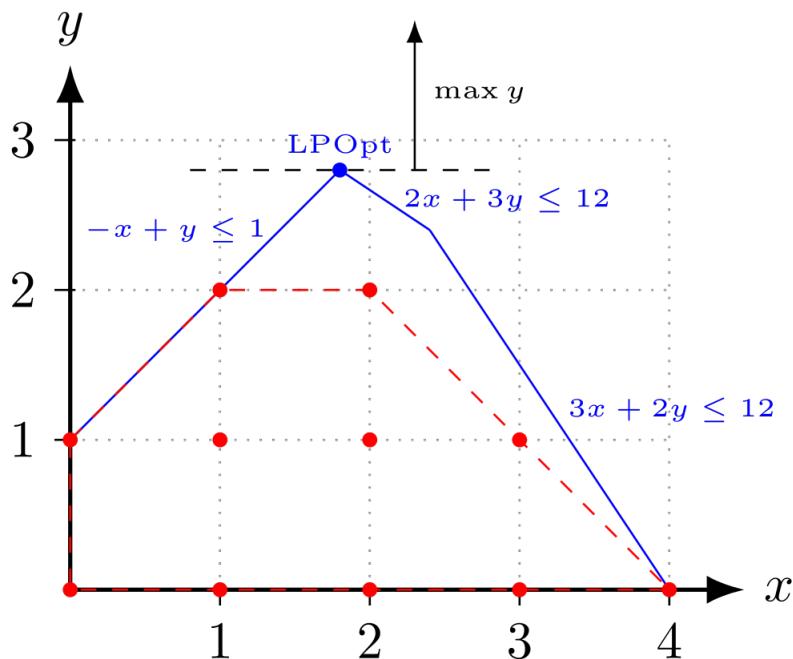
NP-Complete

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

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.



© wiki/File/integer-programming.png³
Figure 2.3: Comparing the LP relaxation to the IP solutions.

Figure 2.3

Integer Linear Programming (ILP):

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 *integer linear programming* problem

³wiki/File/integer-programming.png, from wiki/File/integer-programming.png. wiki/File/integer-programming.png, wiki/File/integer-programming.png.

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 programming* problem is

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

2.7 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.

2.7.1. 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 $f_i(x) = A_i x - b_i$.

2.7.2. 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

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

Alternatively, consider the transformation where $x_i \in \{-1, 1\}$.

$$\begin{aligned} & \min c^\top x \\ \text{s.t. } & Ax \leq b \\ & x \in \{-1, 1\}^n \end{aligned}$$

This can be reformulated with a single nonconvex constraint as

$$\begin{aligned} \min & \quad c^\top x \\ \text{s.t. } & Ax \leq b \\ & -1 \leq x_j \leq 1, \quad 1 \leq j \leq n, \\ & \|x\|^2 \geq n. \end{aligned}$$

2.7.3. 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.

2.8 Mixed Integer Non-Linear Programming (MINLP)

MINLP:

NP-Hard

Mixed Integer Nonlinear Programming (MINLP) combines elements of integer programming and nonlinear programming. In MINLP, some or all of the decision variables are constrained to be integers, and the objective function or constraints are nonlinear. A general MINLP problem can be formulated as:

$$\begin{aligned} \min \quad & f(x, y) \\ \text{s.t.} \quad & g_i(x, y) \leq 0 \quad \text{for } i = 1, \dots, m \\ & h_j(x, y) = 0 \quad \text{for } j = 1, \dots, p \\ & x \in \mathbb{R}^n, y \in \mathbb{Z}^k \end{aligned} \tag{2.1}$$

where f is the objective function, g_i and h_j are constraint functions, x is a vector of continuous variables, and y is a vector of integer variables.

MINLP is particularly challenging due to the nonlinearity in the objective and/or constraints and the discrete nature of some decision variables. It finds applications in various fields such as industry for process optimization, computational geometry, and machine learning for hyperparameter tuning.

2.8.1. Convex MINLP

In the convex case, both the objective function and the constraints are convex functions. This subclass is easier to solve compared to its non-convex counterpart.

Convex MINLP:

NP-Hard, but *Polynomial time (P)* in fixed dimension (typically)

For a convex MINLP, the problem is defined as:

$$\begin{aligned} \min \quad & f(x, y) \quad (\text{convex}) \\ \text{s.t.} \quad & g_i(x, y) \leq 0 \quad (\text{convex constraints}) \\ & x \in \mathbb{R}^n, y \in \mathbb{Z}^k \end{aligned} \tag{2.2}$$

Convex MINLPs, while still challenging, are typically more tractable due to the properties of convexity, which allow for more efficient solution methods.

2.8.2. Non-Convex MINLP

Non-convex MINLPs are significantly harder due to the presence of non-convex functions, which can lead to multiple local minima.

Non-Convex MINLP:

NP-Hard(in fact, undecidable)

The non-convex MINLP problem is formulated as:

$$\begin{aligned} \min \quad & f(x, y) \quad (\text{non-convex}) \\ \text{s.t.} \quad & g_i(x, y) \leq 0 \quad (\text{possibly non-convex constraints}) \\ & x \in \mathbb{R}^n, y \in \mathbb{Z}^k \end{aligned} \tag{2.3}$$

Non-convex MINLPs pose significant computational challenges due to the possibility of multiple local optima and the inherent complexity of integer constraints. These problems are common in real-world applications where decisions are discrete, and the system behavior is non-linear and complex.

Complexity and Applications

MINLP problems are known for their computational complexity, primarily due to the combination of non-linearity and integrality. The non-convex variants, in particular, are *NP-Hard*, making them some of the most challenging problems in optimization.

In practice, MINLP models find extensive applications across various domains. In industry, they are used for complex decision-making processes like supply chain optimization and production planning. In computational geometry, MINLP techniques help in solving problems like optimal packing or layout design. Furthermore, in the realm of machine learning, MINLPs are crucial for tasks like feature selection and hyperparameter optimization where discrete choices and nonlinear relationships are involved.

The versatility of MINLP models, combined with their inherent complexity, makes them a fascinating and essential area of study in the field of optimization.

Part I

Linear Programming

2.8.3. 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

$$\underset{x}{\text{Maximize}} \quad \mathbf{c}^\top \mathbf{x} \quad (2.4a)$$

$$\text{s.t.} \quad \mathbf{a}^\top \mathbf{x} \leq b \quad (2.4b)$$

$$\mathbf{x} \in [0, 1]^n \quad (2.4c)$$

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,

$$\mathbf{x}^* = (1, \dots, 1, \frac{\bar{b}}{a_k}, 0, \dots, 0), \quad (2.5)$$

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.

3. Modeling: Linear Programming

Outcomes

1. Define what a linear program is
2. Understand how to model a linear program
3. View many examples and get a sense of what types of problems can be modeled as linear programs.

Linear Programming, also known as Linear Optimization, is the starting point for most forms of optimization. It is the problem of optimization a linear function over linear constraints.

In this section, we will define what this means, how to setup a linear program, and discuss many examples. Examples will be connected with code in Excel and Python (using with PuLP or Gurobipy modeling tools) so that you can easily start solving optimization problems. Tutorials on these tools will come in later chapters.

We begin this section with a simple example.

Example: Toy Maker

Excel PuLP Gurobipy

Consider the problem of a toy company that produces toy planes and toy boats. The toy company can sell its planes for \$10 and its boats for \$8 dollars. It costs \$3 in raw materials to make a plane and \$2 in raw materials to make a boat. A plane requires 3 hours to make and 1 hour to finish while a boat requires 1 hour to make and 2 hours to finish. The toy company knows it will not sell anymore than 35 planes per week. Further, given the number of workers, the company cannot spend anymore than 160 hours per week finishing toys and 120 hours per week making toys. The company wishes to maximize the profit it makes by choosing how much of each toy to produce.

We can represent the profit maximization problem of the company as a linear programming problem. Let x_1 be the number of planes the company will produce and let x_2 be the number of boats the company will produce. The profit for each plane is $\$10 - \$3 = \$7$ per plane and the profit for each boat is $\$8 - \$2 = \$6$ per boat. Thus the total profit the company will make is:

$$z(x_1, x_2) = 7x_1 + 6x_2 \quad (3.1)$$

The company can spend no more than 120 hours per week making toys and since a plane takes 3 hours to make and a boat takes 1 hour to make we have:

$$3x_1 + x_2 \leq 120 \quad (3.2)$$

Likewise, the company can spend no more than 160 hours per week finishing toys and since it takes 1 hour to finish a plane and 2 hour to finish a boat we have:

$$x_1 + 2x_2 \leq 160 \quad (3.3)$$

Finally, we know that $x_1 \leq 35$, since the company will make no more than 35 planes per week. Thus the complete linear programming problem is given as:

$$\left\{ \begin{array}{l} \max z(x_1, x_2) = 7x_1 + 6x_2 \\ \text{s.t. } 3x_1 + x_2 \leq 120 \\ \quad x_1 + 2x_2 \leq 160 \\ \quad x_1 \leq 35 \\ \quad x_1 \geq 0 \\ \quad x_2 \geq 0 \end{array} \right. \quad (3.4)$$

Exercise 3.1: Chemical Manufacturing

A chemical manufacturer produces three chemicals: A, B and C. These chemicals are produced by two processes: 1 and 2. Running process 1 for 1 hour costs \$4 and yields 3 units of chemical A, 1 unit of chemical B and 1 unit of chemical C. Running process 2 for 1 hour costs \$1 and produces 1 unit of chemical A, and 1 unit of chemical B (but none of Chemical C). To meet customer demand, at least 10 units of chemical A, 5 units of chemical B and 3 units of chemical C must be produced daily. Assume that the chemical manufacturer wants to minimize the cost of production. Develop a linear programming problem describing the constraints and objectives of the chemical manufacturer. [Hint: Let x_1 be the amount of time Process 1 is executed and let x_2 be amount of time Process 2 is executed. Use the coefficients above to express the cost of running Process 1 for x_1 time and Process 2 for x_2 time. Do the same to compute the amount of chemicals A, B, and C that are produced.]

3.1 Modeling and Assumptions in Linear Programming

Outcomes

1. Address crucial assumptions when choosing to model a problem with linear programming.

3.1.1. General models

A Generic Linear Program (LP)

Decision Variables:

x_i : continuous variables ($x_i \in \mathbb{R}$, i.e., a real number), $\forall i = 1, \dots, 3$.

Parameters (known input parameters):

c_i : cost coefficients $\forall i = 1, \dots, n$

a_{ij} : constraint coefficients $\forall i = 1, \dots, n, j = 1, \dots, m$

b_j : right hand side coefficient for constraint j , $j = 1, \dots, m$

The problem we will consider is

$$\begin{aligned} \max \quad & z = c_1x_1 + \dots + c_nx_n \\ \text{s.t.} \quad & a_{11}x_1 + \dots + a_{1n}x_n \leq b_1 \\ & \vdots \\ & a_{m1}x_1 + \dots + a_{mn}x_n \leq b_m \end{aligned} \tag{3.1}$$

For example, in 3 variables and 4 constraints this could look like the following. The following example considers other types of constraints, i.e., \geq and $=$. We will show how all these forms can be converted later.

Decision Variables:

x_i : continuous variables ($x_i \in \mathbb{R}$, i.e., a real number), $\forall i = 1, \dots, 3$.

Parameters (known input parameters):

c_i : cost coefficients $\forall i = 1, \dots, 3$

a_{ij} : constraint coefficients $\forall i = 1, \dots, 3, j = 1, \dots, 4$

b_j : right hand side coefficient for constraint j , $j = 1, \dots, 4$

$$\text{Min } z = c_1x_1 + c_2x_2 + c_3x_3 \tag{3.2}$$

$$\text{s.t. } a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \geq b_1 \tag{3.3}$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \leq b_2 \tag{3.4}$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \tag{3.5}$$

$$a_{41}x_1 + a_{42}x_2 + a_{43}x_3 \geq b_4 \tag{3.6}$$

$$x_1 \geq 0, x_2 \leq 0, x_3 \text{ urs.} \tag{3.7}$$

Definition 3.2: Linear Function

A function $z : \mathbb{R}^n \rightarrow \mathbb{R}$ is linear if there are constants $c_1, \dots, c_n \in \mathbb{R}$ so that:

$$z(x_1, \dots, x_n) = c_1x_1 + \dots + c_nx_n \quad (3.8)$$

For the time being, we will eschew the general form and focus exclusively on linear programming problems with two variables. Using this limited case, we will develop a graphical method for identifying optimal solutions, which we will generalize later to problems with arbitrary numbers of variables.

3.1.2. Assumptions

Inspecting Example 1 (or the more general Problem 3.1) we can see there are several assumptions that must be satisfied when using a linear programming model. We enumerate these below:

Proportionality Assumption A problem can be phrased as a linear program only if the contribution to the objective function *and* the left-hand-side of each constraint by each decision variable (x_1, \dots, x_n) is proportional to the value of the decision variable.

Additivity Assumption A problem can be phrased as a linear programming problem only if the contribution to the objective function *and* the left-hand-side of each constraint by any decision variable x_i ($i = 1, \dots, n$) is completely independent of any other decision variable x_j ($j \neq i$) and additive.

Divisibility Assumption A problem can be phrased as a linear programming problem only if the quantities represented by each decision variable are infinitely divisible (i.e., fractional answers make sense).

Certainty Assumption A problem can be phrased as a linear programming problem only if the coefficients in the objective function and constraints are known with certainty.

The first two assumptions simply assert (in English) that both the objective function and functions on the left-hand-side of the (in)equalities in the constraints are linear functions of the variables x_1, \dots, x_n .

The third assumption asserts that a valid optimal answer could contain fractional values for decision variables. It's important to understand how this assumption comes into play—even in the toy making example. Many quantities can be divided into non-integer values (ounces, pounds etc.) but many other quantities cannot be divided. For instance, can we really expect that it's reasonable to make $\frac{1}{2}$ a plane in the toy making example? When values must be constrained to true integer values, the linear programming problem is called an *integer programming problem*. There is a vast literature dealing with these problems [PS98, WN99]. For many problems, particularly when the values of the decision variables may become large, a fractional optimal answer could be obtained and then rounded to the nearest integer to obtain a reasonable answer. For example, if our toy problem were re-written so that the optimal answer was to make 1045.3 planes, then we could round down to 1045.

The final assumption asserts that the coefficients (e.g., profit per plane or boat) is known with absolute

certainty. In traditional linear programming, there is no lack of knowledge about the make up of the objective function, the coefficients in the left-hand-side of the constraints or the bounds on the right-hand-sides of the constraints. There is a literature on *stochastic programming* [KW94, BN02] that relaxes some of these assumptions, but this too is outside the scope of the course.

Exercise 3.3

In a short sentence or two, discuss whether the problem given in Example 1 meets all of the assumptions of a scenario that can be modeled by a linear programming problem. Do the same for Exercise 3. [Hint: Can you make $\frac{2}{3}$ of a toy? Can you run a process for $\frac{1}{3}$ of an hour?]

3.2 Examples

Outcomes

- A. Learn how to format a linear optimization problem.
- B. Identify and understand common classes of linear optimization problems.

We will begin with a few examples, and then discuss specific problem types that occur often.

Example: Production with welding robot

Excel PuLP Gurobipy

You have 21 units of transparent aluminum alloy (TAA), LazWeld1, a joining robot leased for 23 hours, and CrumCut1, a cutting robot leased for 17 hours of aluminum cutting. You also have production code for a bookcase, desk, and cabinet, along with commitments to buy any of these you can produce for \$18, \$16, and \$10 apiece, respectively. A bookcase requires 2 units of TAA, 3 hours of joining, and 1 hour of cutting, a desk requires 2 units of TAA, 2 hours of joining, and 2 hour of cutting, and a cabinet requires 1 unit of TAA, 2 hours of joining, and 1 hour of cutting. Formulate an LP to maximize your revenue given your current resources.

Solution.**Sets:**

- The types of objects = { bookcase, desk, cabinet}.

Parameters:

- Purchase cost of each object
- Units of TAA needed for each object
- Hours of joining needed for each object
- Hours of cutting needed for each object
- Hours of TAA, Joining, and Cutting available on robots

Decision variables:

x_i : number of units of product i to produce,
for all i =bookcase, desk, cabinet.

Objective and Constraints:

$$\begin{aligned}
 \max \quad & z = 18x_1 + 16x_2 + 10x_3 && \text{(profit)} \\
 \text{s.t.} \quad & 2x_1 + 2x_2 + 1x_3 \leq 21 && \text{(TAA)} \\
 & 3x_1 + 2x_2 + 2x_3 \leq 23 && \text{(LazWeld1)} \\
 & 1x_1 + 2x_2 + 1x_3 \leq 17 && \text{(CrumCut1)} \\
 & x_1, x_2, x_3 \geq 0.
 \end{aligned}$$



Example: The Diet Problem

Gurobipy

In the future (as envisioned in a bad 70's science fiction film) all food is in tablet form, and there are four types, green, blue, yellow, and red. A balanced, futuristic diet requires, at least 20 units of Iron, 25 units of Vitamin B, 30 units of Vitamin C, and 15 units of Vitamin D. Formulate an LP that ensures a balanced diet at the minimum possible cost.

Tablet	Iron	B	C	D	Cost (\$)
green (1)	6	6	7	4	1.25
blue (2)	4	5	4	9	1.05
yellow (3)	5	2	5	6	0.85
red (4)	3	6	3	2	0.65

Solution. Now we formulate the problem:

Sets:

- Set of tablets $\{1, 2, 3, 4\}$

Parameters:

- Iron in each tablet
- Vitamin B in each tablet
- Vitamin C in each tablet
- Vitamin D in each tablet
- Cost of each tablet

Decision variables:

x_i : number of tablet of type i to include in the diet, $\forall i \in \{1, 2, 3, 4\}$.

Objective and Constraints:

$$\begin{aligned}
 \text{Min } z &= 1.25x_1 + 1.05x_2 + 0.85x_3 + 0.65x_4 \\
 \text{s.t. } 6x_1 + 4x_2 + 5x_3 + 3x_4 &\geq 20 \\
 6x_1 + 5x_2 + 2x_3 + 6x_4 &\geq 25 \\
 7x_1 + 4x_2 + 5x_3 + 3x_4 &\geq 30 \\
 4x_1 + 9x_2 + 6x_3 + 2x_4 &\geq 15 \\
 x_1, x_2, x_3, x_4 &\geq 0.
 \end{aligned}$$



Example: The Next Diet Problem

Gurobipy

Progress is important, and our last problem had too many tablets, so we are going to produce a single, purple, 10 gram tablet for our futuristic diet requires, which are at least 20 units of Iron, 25 units of Vitamin B, 30 units of Vitamin C, and 15 units of Vitamin D, and 2000 calories. The tablet is made from blending 4 nutritious chemicals; the following table shows the units of our nutrients per, and cost of, grams of each chemical. Formulate an LP that ensures a balanced diet at the minimum possible cost.

Tablet	Iron	B	C	D	Calories	Cost (\$)
Chem 1	6	6	7	4	1000	1.25
Chem 2	4	5	4	9	250	1.05
Chem 3	5	2	5	6	850	0.85
Chem 4	3	6	3	2	750	0.65

Solution.**Sets:**

- Set of chemicals {1,2,3,4}

Parameters:

- Iron in each chemical
- Vitamin B in each chemical
- Vitamin C in each chemical
- Vitamin D in each chemical
- Cost of each chemical

Decision variables:

x_i : grams of chemical i to include in the purple tablet, $\forall i = 1, 2, 3, 4$.

Objective and Constraints:

$$\begin{aligned} \min z &= 1.25x_1 + 1.05x_2 + 0.85x_3 + 0.65x_4 \\ s.t. \quad &6x_1 + 4x_2 + 5x_3 + 3x_4 \geq 20 \\ &6x_1 + 5x_2 + 2x_3 + 6x_4 \geq 25 \\ &7x_1 + 4x_2 + 5x_3 + 3x_4 \geq 30 \\ &4x_1 + 9x_2 + 6x_3 + 2x_4 \geq 15 \\ &1000x_1 + 250x_2 + 850x_3 + 750x_4 \geq 2000 \\ &x_1 + x_2 + x_3 + x_4 = 10 \\ &x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$



Example: Work Scheduling Problem

Gurobipy

You are the manager of LP Burger. The following table shows the minimum number of employees required to staff the restaurant on each day of the week. Each employee must work for five consecutive days. Formulate an LP to find the minimum number of employees required to staff the restaurant.

Day of Week	Workers Required
1 = Monday	6
2 = Tuesday	4
3 = Wednesday	5
4 = Thursday	4
5 = Friday	3
6 = Saturday	7
7 = Sunday	7

Solution. Decision variables:

Decision variables:

x_i : the number of workers that start 5 consecutive days of work on day i , $i = 1, \dots, 7$

$$\begin{aligned}
 \text{Min } z &= x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \\
 \text{s.t. } x_1 + x_4 + x_5 + x_6 + x_7 &\geq 6 \\
 x_2 + x_5 + x_6 + x_7 + x_1 &\geq 4 \\
 x_3 + x_6 + x_7 + x_1 + x_2 &\geq 5 \\
 x_4 + x_7 + x_1 + x_2 + x_3 &\geq 4 \\
 x_5 + x_1 + x_2 + x_3 + x_4 &\geq 3 \\
 x_6 + x_2 + x_3 + x_4 + x_5 &\geq 7 \\
 x_7 + x_3 + x_4 + x_5 + x_6 &\geq 7 \\
 x_1, x_2, x_3, x_4, x_5, x_6, x_7 &\geq 0.
 \end{aligned}$$

The solution is as follows:

LP Solution	IP Solution
$z_{LP} = 7.333$	$z_I = 8.0$
$x_1 = 0$	$x_1 = 0$
$x_2 = 0.333$	$x_2 = 0$
$x_3 = 1$	$x_3 = 0$
$x_4 = 2.333$	$x_4 = 3$
$x_5 = 0$	$x_5 = 0$
$x_6 = 3.333$	$x_6 = 4$
$x_7 = 0.333$	$x_7 = 1$



Example: LP Burger - extended

Gurobipy

LP Burger has changed its policy, and allows, at most, two part time workers, who work for two consecutive days in a week. Formulate this problem.

Solution. Decision variables:

x_i : the number of workers that start 5 consecutive days of work on day i , $i = 1, \dots, 7$

y_i : the number of workers that start 2 consecutive days of work on day i , $i = 1, \dots, 7$.

$$\begin{aligned} \text{Min } z &= 5(x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7) \\ &\quad + 2(y_1 + y_2 + y_3 + y_4 + y_5 + y_6 + y_7) \\ \text{s.t. } x_1 + x_4 + x_5 + x_6 + x_7 + y_1 + y_7 &\geq 6 \\ x_2 + x_5 + x_6 + x_7 + x_1 + y_2 + y_1 &\geq 4 \\ x_3 + x_6 + x_7 + x_1 + x_2 + y_3 + y_2 &\geq 5 \\ x_4 + x_7 + x_1 + x_2 + x_3 + y_4 + y_3 &\geq 4 \\ x_5 + x_1 + x_2 + x_3 + x_4 + y_5 + y_4 &\geq 3 \\ x_6 + x_2 + x_3 + x_4 + x_5 + y_6 + y_5 &\geq 7 \\ x_7 + x_3 + x_4 + x_5 + x_6 + y_7 + y_6 &\geq 7 \\ y_1 + y_2 + y_3 + y_4 + y_5 + y_6 + y_7 &\leq 2 \\ x_i &\geq 0, y_i \geq 0, \forall i = 1, \dots, 7. \end{aligned}$$

**3.2.1. Knapsack Problem****Example: Capital allocation**

Excel PuLP Gurobipy

3.2.2. Capital Investment**Example: Capital Investment¹**

Excel PuLP Gurobipy

3.2.3. Work Scheduling

3.2.4. Assignment Problem

Consider the assignment of n teams to n projects, where each team ranks the projects, where their favorite project is given a rank of n , their next favorite $n - 1$, and their least favorite project is given a rank of 1. The assignment problem is formulated as follows (we denote ranks using the R -parameter):

Variables:

x_{ij} : 1 if project i assigned to team j , else 0.

$$\begin{aligned} \text{Max } z &= \sum_{i=1}^n \sum_{j=1}^n R_{ij} x_{ij} \\ \text{s.t. } \sum_{i=1}^n x_{ij} &= 1, \quad \forall j = 1, \dots, n \\ \sum_{j=1}^n x_{ij} &= 1, \quad \forall i = 1, \dots, n \\ x_{ij} &\geq 0, \quad \forall i = 1, \dots, n, j = 1, \dots, n. \end{aligned}$$

Example: Hiring for tasks

Excel PuLP Gurobipy

In this assignment problem, we need to hire three people (Person 1, Person 2, Person 3) to three tasks (Task 1, Task 2, Task 3). In the table below, we list the cost of hiring each person for each task, in dollars. Since each person has a different cost for each task, we must make an assignment to minimize

	Cost	Task 1	Task 2	Task 3
our total cost.				
Person 1	40	47	80	
Person 2	72	36	58	
Person 3	24	61	71	

The assignment problem has an integrality property, such that if we remove the binary restriction on the x variables (now just non-negative, i.e., $x_{ij} \geq 0$) then we still get binary assignments, despite the fact that it is now an LP. This property is very interesting and useful. Of course, the objective function might not quite what we want, we might be interested ensuring that the team with the worst assignment is as good as possible (a fairness criteria). One way of doing this is to modify the assignment problem using a max-min objective:

Max-min Assignment-like Formulation

$$\begin{aligned}
 & \text{Max} \quad z \\
 \text{s.t.} \quad & \sum_{i=1}^n x_{ij} = 1, \quad \forall j = 1, \dots, n \\
 & \sum_{j=1}^n x_{ij} = 1, \quad \forall i = 1, \dots, n \\
 & x_{ij} \geq 0, \quad \forall i = 1, \dots, n, J = 1, \dots, n \\
 & z \leq \sum_{i=1}^n R_{ij} x_{ij}, \quad \forall j = 1, \dots, n.
 \end{aligned}$$

Does this formulation have the integrality property (it is not an assignment problem)? Consider a very simple example where two teams are to be assigned to two projects and the teams give the projects the following rankings: Both teams prefer Project 2. For both problems, if we remove the binary restriction on

	Project 1	Project 2
Team 1	2	1
Team 2	2	1

the x -variable, they can take values between (and including) zero and one. For the assignment problem the optimal solution will have $z = 3$, and fractional x -values will not improve z . For the max-min assignment problem this is not the case, the optimal solution will have $z = 1.5$, which occurs when each team is assigned half of each project (i.e., for Team 1 we have $x_{11} = 0.5$ and $x_{21} = 0.5$).

3.2.5. Multi period Models

3.2.5.1. Production Planning

3.2.5.2. Crop Planning

3.2.6. Mixing Problems

3.2.7. Financial Planning

3.2.8. Network Flow

Resources

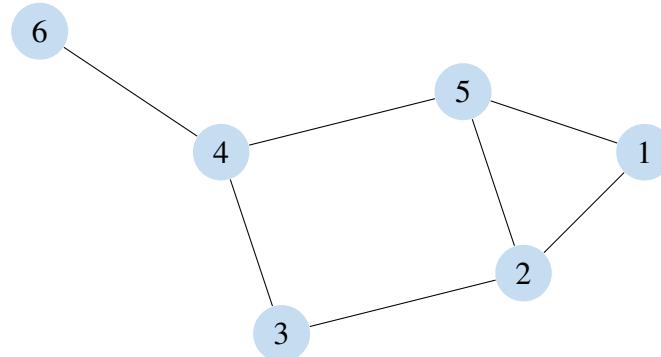
- MIT - CC BY NC SA 4.0 license
- Slides for Algorithms book by Kleinberg-Tardos

To begin a discussion on Network flow, we first need to discuss graphs.

3.2.8.1. Graphs

A graph $G = (V, E)$ is defined by a set of vertices V and a set of edges E that contains pairs of vertices.

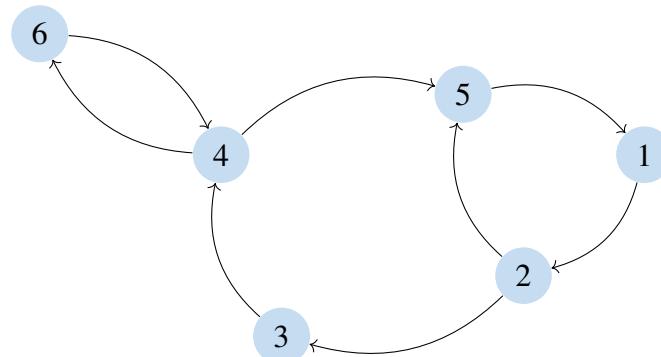
For example, the following graph G can be described by the vertex set $V = \{1, 2, 3, 4, 5, 6\}$ and the edge set $E = \{(4, 6), (4, 5), (5, 1), (1, 2), (2, 5), (2, 3), (3, 4)\}$.



In an undirected graph, we do not distinguish the direction of the edge. That is, for two vertices $i, j \in V$, we can equivalently write (i, j) or (j, i) to represent the edge.

Alternatively, we will want to consider directed graphs. We denote these as $G = (V, \mathcal{A})$ where \mathcal{A} is a set of arcs where an arc is a directed edge.

For example, the following directed graph G can be described by the vertex set $V = \{1, 2, 3, 4, 5, 6\}$ and the edge set $\mathcal{A} = \{(4, 6), (6, 4), (4, 5), (5, 1), (1, 2), (2, 5), (2, 3), (3, 4)\}$.



SETS A finite network G is described by a finite set of vertices V and a finite set \mathcal{A} of arcs. Each arc (i, j) has two key attributes, namely its tail $j \in V$ and its head $i \in V$.

We think of a (single) commodity as being allowed to "flow" along each arc, from its tail to its head.

VARIABLES Indeed, we have "flow" variables

$$x_{ij} := \text{amount of flow on arc}(i, j) \text{ from vertex } i \text{ to vertex } j,$$

for all $(i, j) \in \mathcal{A}$.

3.2.8.2. Maximum Flow Problem

$$\max \sum_{(s,i) \in \mathcal{A}} x_{si} \quad \text{max total flow from source} \quad (3.1)$$

$$\text{s.t.} \quad \sum_{i:(i,v) \in \mathcal{A}} x_{iv} - \sum_{j:(v,j) \in \mathcal{A}} x_{vj} = 0 \quad v \in V \setminus \{s, t\} \quad (3.2)$$

$$0 \leq x_{ij} \leq u_{ij} \quad \forall (i, j) \in \mathcal{A} \quad (3.3)$$

SHORTEST PATH PROBLEM

$$\begin{aligned} & \text{minimize} && \sum_{u \rightarrow v} \ell_{u \rightarrow v} \cdot x_{u \rightarrow v} \\ & \text{subject to} && \sum_u x_{u \rightarrow s} - \sum_w x_{s \rightarrow w} = 1 \\ & && \sum_u x_{u \rightarrow t} - \sum_w x_{t \rightarrow w} = -1 \\ & && \sum_u x_{u \rightarrow v} - \sum_w x_{v \rightarrow w} = 0 \quad \text{for every vertex } v \neq s, t \\ & && x_{u \rightarrow v} \geq 0 \quad \text{for every edge } u \rightarrow v \end{aligned}$$

Or maybe write it like this:

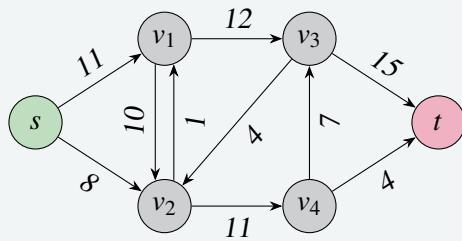
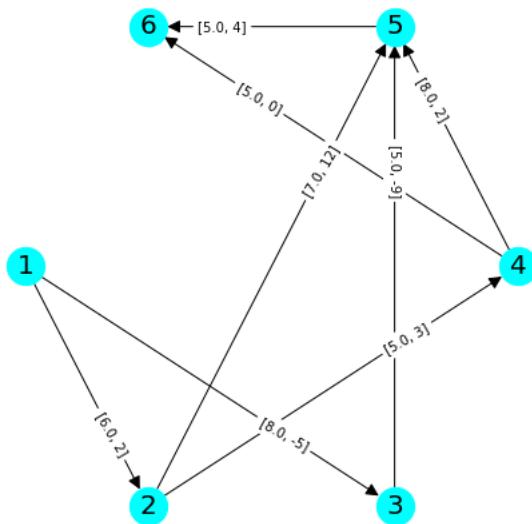
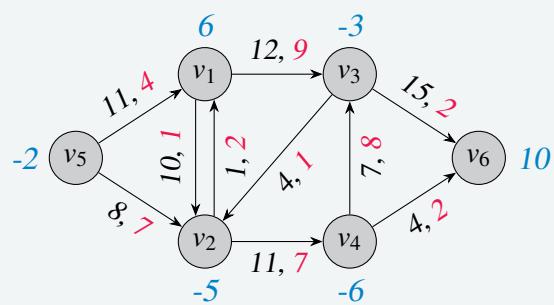
$$\min \sum_{(i,j) \in A} c_{ij} x_{ij} \quad (3.4)$$

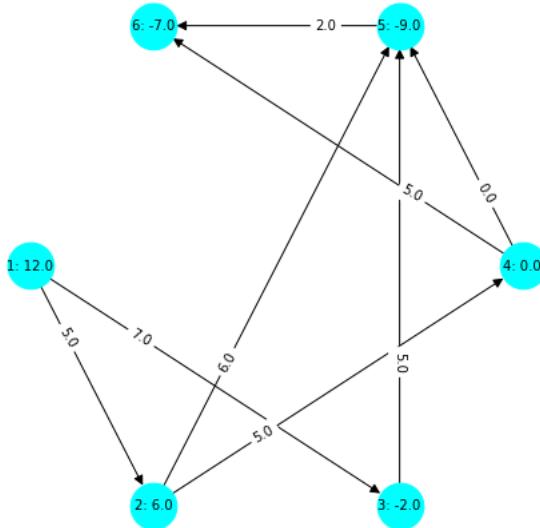
$$\text{s.t.} \quad \sum_{(i,j) \in \delta^+(i)} x_{ij} = 0 \quad \forall i \in V \setminus \{s, t\} \quad (3.5)$$

$$\sum_{(i,j) \in \delta^+(s)} x_{ij} = -1 \quad (3.6)$$

$$\sum_{(i,j) \in \delta^+(t)} x_{ij} = 1 \quad (3.7)$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in A, \quad (3.8)$$

Example 3.4: Max flow example**Example 3.5: Min Cost Network Flow**© network-flow²**Figure 3.1: network-flow**²network-flow, from [network-flow](#). [network-flow](#), [network-flow](#).³network-flow-solution, from [network-flow-solution](#). [network-flow-solution](#), [network-flow-solution](#).



© network-flow-solution³
Figure 3.2: network-flow-solution

3.2.8.3. Minimum Cost Network Flow

PARAMETERS We assume that flow on arc (i, j) should be non-negative and should not exceed

$$u_{ij} := \text{the flow upper bound on arc } (i, j),$$

for $(i, j) \in \mathcal{A}$. Associated with each arc (i, j) is a cost

$$c_{ij} := \text{cost per-unit-flow on arc } (i, j),$$

for $(i, j) \in \mathcal{A}$. The (total) cost of the flow x is defined to be

$$\sum_{(i,j) \in \mathcal{A}} c_{ij} x_{ij}.$$

We assume that we have further data for the nodes. Namely,

$$b_v := \text{the net supply at node } v,$$

for $v \in V$.

A flow is conservative if the net flow out of node v , minus the net flow into node v , is equal to the net supply at node v , for all nodes $v \in V$.

The (single-commodity min-cost) network-flow problem is to find a minimumcost conservative flow that is non-negative and respects the flow upper bounds on the arcs.

OBJECTIVE AND CONSTRAINTS We can formulate this as follows:

$$\begin{aligned}
 \min \sum_{(i,j) \in \mathcal{A}} c_{ij} x_{ij} & \quad \text{minimize cost} \\
 \sum_{(i,v) \in \mathcal{A}} x_{iv} - \sum_{(v,i) \in \mathcal{A}} x_{vi} = b_v, \quad \text{for all } v \in V, & \quad \text{flow conservation} \\
 0 \leq x_{ij} \leq u_{ij}, \quad \text{for all } (i,j) \in \mathcal{A}. &
 \end{aligned}$$

Theorem 3.6: Integrality of Network Flow

If the capacities and demands are all integer values, then there always exists an optimal solution to the LP that has integer values.

3.2.9. Multi-Commodity Network Flow

In the same vein as the Network Flow Problem

$$\begin{aligned}
 \min \sum_{k=1}^K \sum_{e \in \mathcal{A}} c_e^k x_e^k \\
 \sum_{e \in \mathcal{A} : t(e)=v} x_e^k - \sum_{e \in \mathcal{A} : h(e)=v} x_e^k = b_v^k, \quad \text{for } v \in \mathcal{N}, k = 1, 2, \dots, K; \\
 \sum_{k=0}^K x_e^k \leq u_e, \quad \text{for } e \in \mathcal{A}; \\
 x_e^k \geq 0, \quad \text{for } e \in \mathcal{A}, k = 1, 2, \dots, K
 \end{aligned}$$

Notes:

$K=1$ is ordinary single-commodity network flow. Integer solutions for free when node-supplies and arc capacities are integer. $K=2$ example below with integer data gives a fractional basic optimum. This example doesn't have any feasible integer flow at all.

Remark 3.7

Unfortunately, the same integrality theorem does not hold in the multi-commodity network flow problem. Nonetheless, if the quantities in each flow are very large, then the LP solution will likely be very close to an integer valued solution.

3.3 Modeling Tricks

3.3.1. 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

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

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

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

Example 3.8: Minimizing an Absolute Value

Note that

$$|t| = \max(t, -t),$$

Thus, if we need to minimize $|t|$ we can instead write

$$\min z \tag{3.1}$$

$$s.t. \tag{3.2}$$

$$t \leq z - t \leq z \tag{3.3}$$

3.4 Other examples

Food manfacturing - GUROBI

Optimization Methods in Finance - CorneuJols, Tütüncü

4. Graphically Solving Linear Programs

Outcomes

- A. Learn how to plot the feasible region and the objective function.
- B. Identify and compute extreme points of the feasible region.
- C. Find the optimal solution(s) to a linear program graphically.
- D. Classify the type of result of the problem as infeasible, unbounded, unique optimal solution, or infinitely many optimal solutions.

Linear Programs (LP's) with two variables can be solved graphically by plotting the feasible region along with the level curves of the objective function.¹ We will show that we can find a point in the feasible region that maximizes the objective function using the level curves of the objective function.

We will begin with an easy example that is bounded and investigate the structure of the feasible region. We will then explore other examples.

4.1 Nonempty and Bounded Problem

Consider the problem

$$\begin{aligned} \max \quad & 2X + 5Y \\ \text{s.t.} \quad & X + 2Y \leq 16 \\ & 5X + 3Y \leq 45 \\ & X, Y \geq 0 \end{aligned}$$

We want to start by plotting the *feasible region*, that is, the set points (X, Y) that satisfy all the constraints.

We can plot this by first plotting the four lines

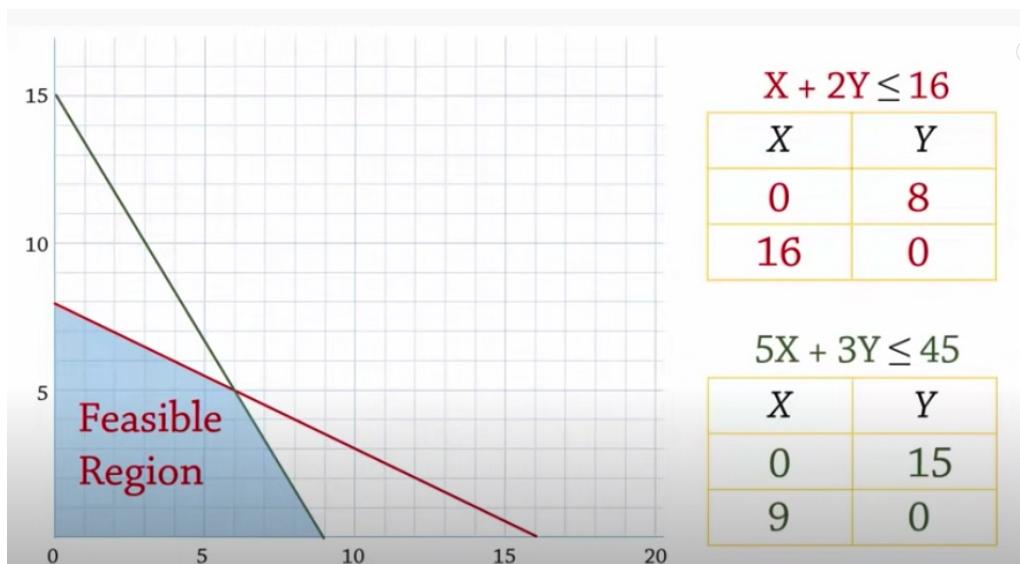
- $X + 2Y = 16$
- $5X + 3Y = 45$
- $X = 0$
- $Y = 0$

¹Special thanks to Joshua Emmanuel and Christopher Griffin for sharing their content to help put this section together. Proper citations and references are forthcoming.

and then shading in the side of the space cut out by the corresponding inequality.



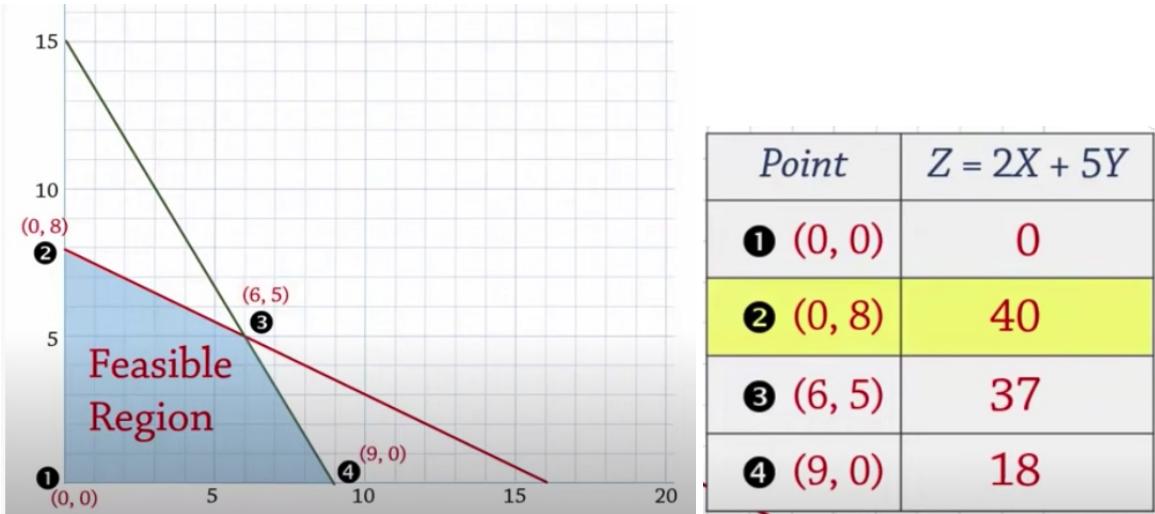
The resulting feasible region can then be shaded in as the region that satisfies all the inequalities.



Notice that the feasible region is nonempty (it has points that satisfy all the inequalities) and also that it is bounded (the feasible points don't continue infinitely in any direction).

We want to identify the *extreme points* (i.e., the corners) of the feasible region. Understanding these points will be critical to understanding the optimal solutions of the model. Notice that all extreme points can be computed by finding the intersection of 2 of the lines. But! Not all intersections of any two lines are feasible.

We will later use the terminology *basic feasible solution* for an extreme point of the feasible region, and *basic solution* as a point that is the intersection of 2 lines, but is actually infeasible (does not satisfy all the constraints).



Theorem 4.1: Optimal Extreme Point

If the feasible region is nonempty and bounded, then there exists an optimal solution at an extreme point of the feasible region.

We will explore why this theorem is true, and also what happens when the feasible region does not satisfy the assumptions of either nonempty or bounded. We illustrate the idea first using the problem from Example 1.

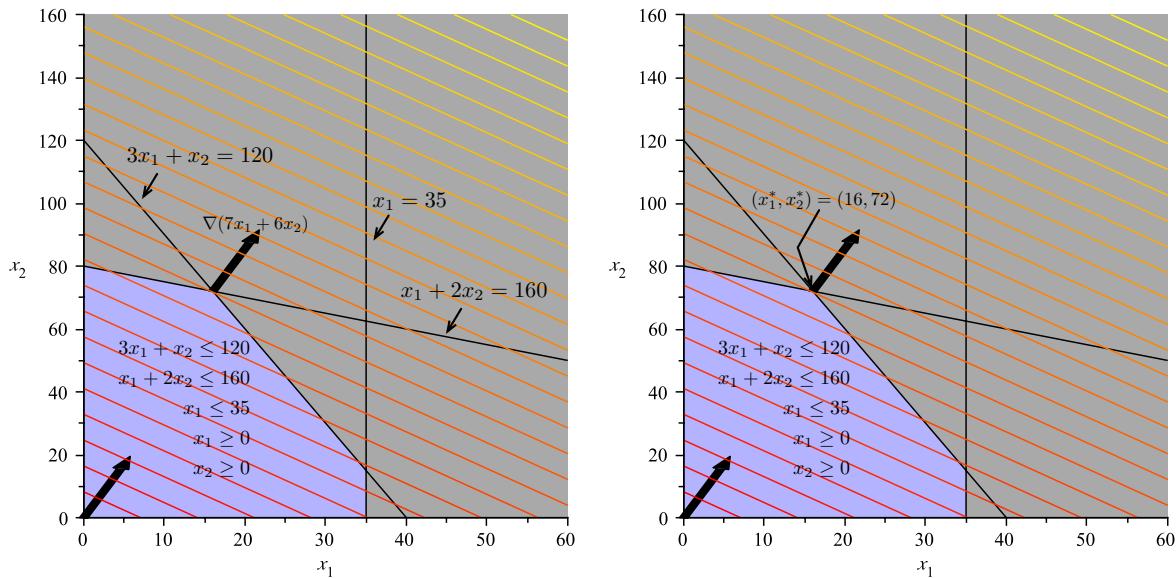


Figure 4.1: Feasible Region and Level Curves of the Objective Function: The shaded region in the plot is the feasible region and represents the intersection of the five inequalities constraining the values of x_1 and x_2 . On the right, we see the optimal solution is the “last” point in the feasible region that intersects a level set as we move in the direction of increasing profit.

Example 4.2: Continuation of Example

Let's continue the example of the Toy Maker begin in Example 1. Solve this problem graphically.

Solution. To solve the linear programming problem graphically, begin by drawing the feasible region. This is shown in the blue shaded region of Figure 4.1.

After plotting the feasible region, the next step is to plot the level curves of the objective function. In our problem, the level sets will have the form:

$$7x_1 + 6x_2 = c \implies x_2 = \frac{-7}{6}x_1 + \frac{c}{6}$$

This is a set of parallel lines with slope $-7/6$ and intercept $c/6$ where c can be varied as needed. The level curves for various values of c are parallel lines. In Figure 4.1 they are shown in colors ranging from red to yellow depending upon the value of c . Larger values of c are more yellow.

To solve the linear programming problem, follow the level sets along the gradient (shown as the black arrow) until the last level set (line) intersects the feasible region. If you are doing this by hand, you can draw a single line of the form $7x_1 + 6x_2 = c$ and then simply draw parallel lines in the direction of the gradient $(7, 6)$. At some point, these lines will fail to intersect the feasible region. The last line to intersect the feasible region will do so at a point that maximizes the profit. In this case, the point that maximizes $z(x_1, x_2) = 7x_1 + 6x_2$, subject to the constraints given, is $(x_1^*, x_2^*) = (16, 72)$.

Note the point of optimality $(x_1^*, x_2^*) = (16, 72)$ is at a corner of the feasible region. This corner is formed by the intersection of the two lines: $3x_1 + x_2 = 120$ and $x_1 + 2x_2 = 160$. In this case, the constraints

$$\begin{aligned} 3x_1 + x_2 &\leq 120 \\ x_1 + 2x_2 &\leq 160 \end{aligned}$$

are both *binding*, while the other constraints are non-binding. In general, we will see that when an optimal solution to a linear programming problem exists, it will always be at the intersection of several binding constraints; that is, it will occur at a corner of a higher-dimensional polyhedron. ♠

We can now define an algorithm for identifying the solution to a linear programming problem in two variables with a *bounded* feasible region (see Algorithm 1):

Algorithm 1 Algorithm for Solving a Two Variable Linear Programming Problem Graphically–Bounded Feasible Region, Unique Solution Case

Algorithm for Solving a Linear Programming Problem Graphically

Bounded Feasible Region, Unique Solution

1. Plot the feasible region defined by the constraints.
 2. Plot the level sets of the objective function.
 3. For a maximization problem, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
 4. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem.
-

The example linear programming problem presented in the previous section has a single optimal solution. In general, the following outcomes can occur in solving a linear programming problem:

1. The linear programming problem has a unique solution. (We've already seen this.)
2. There are infinitely many alternative optimal solutions.
3. There is no solution and the problem's objective function can grow to positive infinity for maximization problems (or negative infinity for minimization problems).
4. There is no solution to the problem at all.

Case 3 above can only occur when the feasible region is unbounded; that is, it cannot be surrounded by a ball with finite radius. We will illustrate each of these possible outcomes in the next four sections. We will prove that this is true in a later chapter.

4.2 Infinitely Many Optimal Solutions

It can happen that there is more than one solution. In fact, in this case, there are infinitely many optimal solutions. We'll study a specific linear programming problem with an infinite number of solutions by modifying the objective function in Example 1.

Example 4.3: Toy Maker Alternative Solutions

Suppose the toy maker in Example 1 finds that it can sell planes for a profit of \$18 each instead of \$7 each. The new linear programming problem becomes:

$$\left\{ \begin{array}{l} \max z(x_1, x_2) = 18x_1 + 6x_2 \\ \text{s.t. } 3x_1 + x_2 \leq 120 \\ \quad x_1 + 2x_2 \leq 160 \\ \quad x_1 \leq 35 \\ \quad x_1 \geq 0 \\ \quad x_2 \geq 0 \end{array} \right. \quad (4.1)$$

Solution. Applying our graphical method for finding optimal solutions to linear programming problems yields the plot shown in Figure 4.2. The level curves for the function $z(x_1, x_2) = 18x_1 + 6x_2$ are *parallel* to one face of the polygon boundary of the feasible region. Hence, as we move further up and to the right in the direction of the gradient (corresponding to larger and larger values of $z(x_1, x_2)$) we see that there is not *one* point on the boundary of the feasible region that intersects that level set with greatest value, but instead a side of the polygon boundary described by the line $3x_1 + x_2 = 120$ where $x_1 \in [16, 35]$. Let:

$$S = \{(x_1, x_2) | 3x_1 + x_2 \leq 120, x_1 + 2x_2 \leq 160, x_1 \leq 35, x_1, x_2 \geq 0\}$$

that is, S is the feasible region of the problem. Then for any value of $x_1^* \in [16, 35]$ and any value x_2^* so that $3x_1^* + x_2^* = 120$, we will have $z(x_1^*, x_2^*) \geq z(x_1, x_2)$ for all $(x_1, x_2) \in S$. Since there are infinitely many values that x_1 and x_2 may take on, we see this problem has an infinite number of alternative optimal solutions.

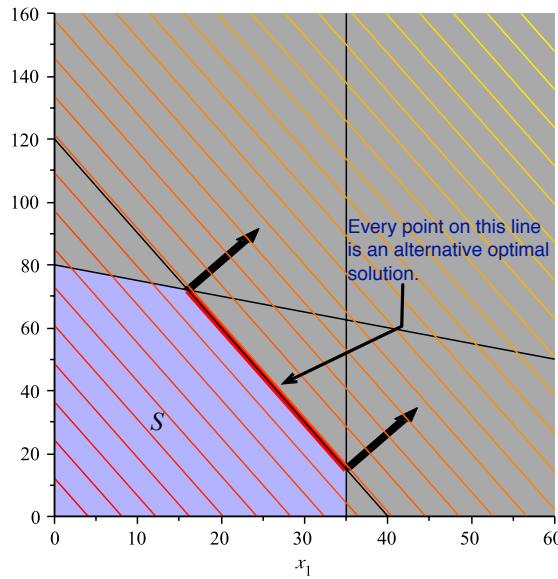


Figure 4.2: An example of infinitely many alternative optimal solutions in a linear programming problem. The level curves for $z(x_1, x_2) = 18x_1 + 6x_2$ are parallel to one face of the polygon boundary of the feasible region. Moreover, this side contains the points of greatest value for $z(x_1, x_2)$ inside the feasible region. Any combination of (x_1, x_2) on the line $3x_1 + x_2 = 120$ for $x_1 \in [16, 35]$ will provide the largest possible value $z(x_1, x_2)$ can take in the feasible region S .

Exercise 4.4

Use the graphical method for solving linear programming problems to solve the linear programming problem you defined in Exercise 3.

Based on the example in this section, we can modify our algorithm for finding the solution to a linear programming problem graphically to deal with situations with an infinite set of alternative optimal solutions (see Algorithm 2):

Algorithm 2 Algorithm for Solving a Two Variable Linear Programming Problem Graphically–Bounded Feasible Region Case

Algorithm for Solving a Linear Programming Problem Graphically

Bounded Feasible Region

1. Plot the feasible region defined by the constraints.
 2. Plot the level sets of the objective function.
 3. For a maximization problem, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
 4. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem.
 5. **If the level set corresponding to the greatest (least) objective function value is parallel to a side of the polygon boundary next to the corner identified, then there are infinitely many alternative optimal solutions and any point on this side may be chosen as an optimal solution.**
-

Exercise 4.5

Modify the linear programming problem from Exercise 3 to obtain a linear programming problem with an infinite number of alternative optimal solutions. Solve the new problem and obtain a description for the set of alternative optimal solutions. [Hint: Just as in the example, x_1 will be bound between two values corresponding to a side of the polygon. Find those values and the constraint that is binding. This will provide you with a description of the form for any $x_1^ \in [a, b]$ and x_2^* is chosen so that $cx_1^* + dx_2^* = v$, the point (x_1^*, x_2^*) is an alternative optimal solution to the problem. Now you fill in values for a, b, c, d and v .]*

4.3 Problems with No Solution

Recall for *any* mathematical programming problem, the feasible set or region is simply a subset of \mathbb{R}^n . If this region is empty, then there is no solution to the mathematical programming problem and the problem is said to be *over constrained*. In this case, we say that the problem is *infeasible*. We illustrate this case for linear programming problems with the following example.

Example 4.6: Infeasible Problem

Consider the following linear programming problem:

$$\left\{ \begin{array}{l} \max z(x_1, x_2) = 3x_1 + 2x_2 \\ \text{s.t. } \frac{1}{40}x_1 + \frac{1}{60}x_2 \leq 1 \\ \quad \frac{1}{50}x_1 + \frac{1}{50}x_2 \leq 1 \\ \quad x_1 \geq 30 \\ \quad x_2 \geq 20 \end{array} \right. \quad (4.1)$$

Solution. The level sets of the objective and the constraints are shown in Figure 4.3.

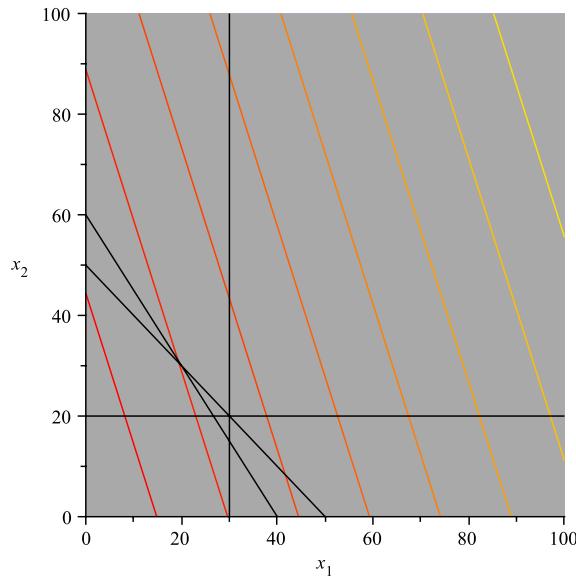


Figure 4.3: A Linear Programming Problem with no solution. The feasible region of the linear programming problem is empty; that is, there are no values for x_1 and x_2 that can simultaneously satisfy all the constraints. Thus, no solution exists.

The fact that the feasible region is empty is shown by the fact that in Figure 4.3 there is no blue region—i.e., all the regions are gray indicating that the constraints are not satisfiable. ♠

Based on this example, we can modify our previous algorithm for finding the solution to linear programming problems graphically (see Algorithm 3):

Algorithm 3 Algorithm for Solving a Two Variable Linear Programming Problem Graphically–Bounded Feasible Region Case

Algorithm for Solving a Linear Programming Problem Graphically

Bounded Feasible Region

1. Plot the feasible region defined by the constraints.
2. **If the feasible region is empty, then no solution exists.**
3. Plot the level sets of the objective function.
4. For a maximization problem, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
5. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem.
6. **If the level set corresponding to the greatest (least) objective function value is parallel to a side of the polygon boundary next to the corner identified, then there are infinitely many alternative optimal solutions and any point on this side may be chosen as an optimal solution.**

4.4 Problems with Unbounded Feasible Regions

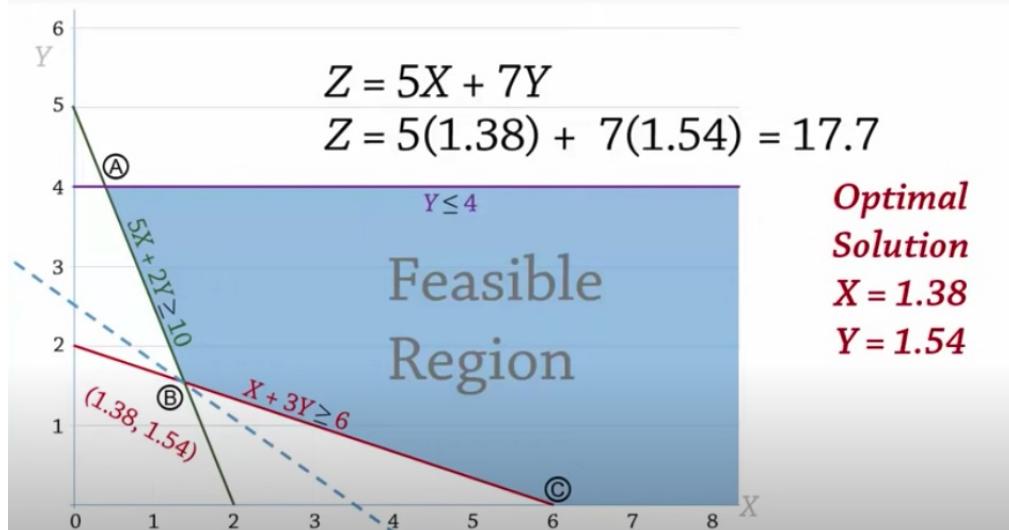
Consider the problem

$$\begin{aligned} \min \quad & Z = 5X + 7Y \\ \text{s.t.} \quad & X + 3Y \geq 6 \\ & 5X + 2Y \geq 10 \\ & Y \leq 4 \\ & X, Y \geq 0 \end{aligned}$$



As you can see, the feasible region is *unbounded*. In particular, from any point in the feasible region, one can always find another feasible point by increasing the X coordinate (i.e., move to the right in the picture). However, this does not necessarily mean that the optimization problem is unbounded.

Indeed, the optimal solution is at the B, the extreme point in the lower left hand corner.



Consider however, if we consider a different problem where we try to maximize the objective

$$\begin{aligned} \max \quad & Z = 5X + 7Y \\ \text{s.t.} \quad & X + 3Y \geq 6 \\ & 5X + 2Y \geq 10 \\ & Y \leq 4 \\ & X, Y \geq 0 \end{aligned}$$

Solution. This optimization problem is unbounded! For example, notice that the point $(X, Y) = (n, 0)$ is feasible for all $n = 1, 2, 3, \dots$. Then the objective function $Z = 5n + 0$ follows the sequence $5, 10, 15, \dots$, which diverges to infinity. ♠

Again, we'll tackle the issue of linear programming problems with unbounded feasible regions by illustrating the possible outcomes using examples.

Example 4.7

Consider the linear programming problem below:

$$\left\{ \begin{array}{l} \max z(x_1, x_2) = 2x_1 - x_2 \\ \text{s.t. } x_1 - x_2 \leq 1 \\ \quad 2x_1 + x_2 \geq 6 \\ \quad x_1, x_2 \geq 0 \end{array} \right. \quad (4.1)$$

Solution. The feasible region and level curves of the objective function are shown in Figure 4.4.

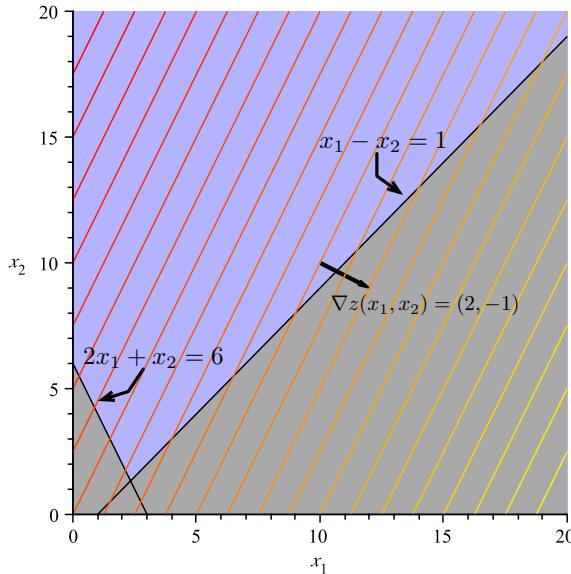


Figure 4.4: A Linear Programming Problem with Unbounded Feasible Region: Note that we can continue to make level curves of $z(x_1, x_2)$ corresponding to larger and larger values as we move down and to the right. These curves will continue to intersect the feasible region for any value of $v = z(x_1, x_2)$ we choose. Thus, we can make $z(x_1, x_2)$ as large as we want and still find a point in the feasible region that will provide this value. Hence, the optimal value of $z(x_1, x_2)$ subject to the constraints $+\infty$. That is, the problem is unbounded.

The feasible region in Figure 4.4 is clearly unbounded since it stretches upward along the x_2 axis infinitely far and also stretches rightward along the x_1 axis infinitely far, bounded below by the line $x_1 - x_2 = 1$. There is no way to enclose this region by a disk of finite radius, hence the feasible region is not bounded.

We can draw more level curves of $z(x_1, x_2)$ in the direction of increase (down and to the right) as long as we wish. There will always be an intersection point with the feasible region because it is infinite. That is, these curves will continue to intersect the feasible region for any value of $v = z(x_1, x_2)$ we choose. Thus, we can make $z(x_1, x_2)$ as large as we want and still find a point in the feasible region that will provide this value. Hence, the largest value $z(x_1, x_2)$ can take when (x_1, x_2) are in the feasible region is $+\infty$. That is, the problem is unbounded. ♠

Just because a linear programming problem has an unbounded feasible region does not imply that there is not a finite solution. We illustrate this case by modifying example 4.4.

Example 4.8: Continuation of Example 4.4

Consider the linear programming problem from Example 4.4 with the new objective function:

$z(x_1, x_2) = (1/2)x_1 - x_2$. Then we have the new problem:

$$\begin{cases} \max z(x_1, x_2) = \frac{1}{2}x_1 - x_2 \\ \text{s.t. } x_1 - x_2 \leq 1 \\ \quad 2x_1 + x_2 \geq 6 \\ \quad x_1, x_2 \geq 0 \end{cases} \quad (4.2)$$

Solution. The feasible region, level sets of $z(x_1, x_2)$ and gradients are shown in Figure 4.5. In this case note, that the direction of increase of the objective function is *away* from the direction in which the feasible region is unbounded (i.e., downward). As a result, the point in the feasible region with the largest $z(x_1, x_2)$ value is $(7/3, 4/3)$. Again this is a vertex: the binding constraints are $x_1 - x_2 = 1$ and $2x_1 + x_2 = 6$ and the solution occurs at the point these two lines intersect.

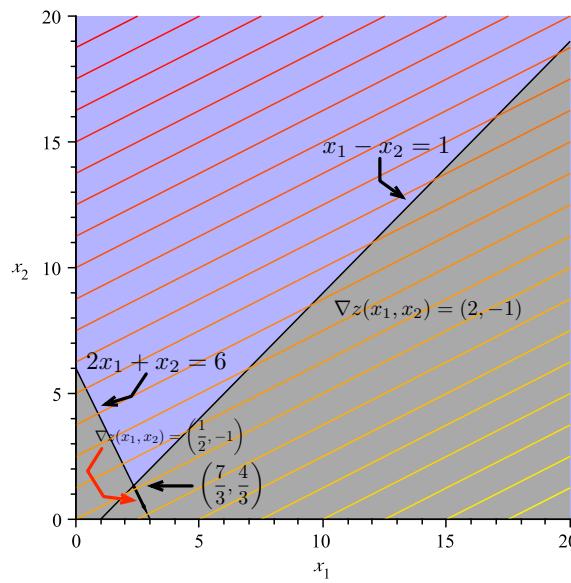


Figure 4.5: A Linear Programming Problem with Unbounded Feasible Region and Finite Solution:
In this problem, the level curves of $z(x_1, x_2)$ increase in a more “southerly” direction than in Example 4.4—that is, *away* from the direction in which the feasible region increases without bound. The point in the feasible region with largest $z(x_1, x_2)$ value is $(7/3, 4/3)$. Note again, this is a vertex.



Based on these two examples, we can modify our algorithm for graphically solving a two variable linear programming problems to deal with the case when the feasible region is unbounded.

Algorithm 4 Algorithm for Solving a Linear Programming Problem Graphically–Bounded and Unbounded Case

Algorithm for Solving a Two Variable Linear Programming Problem Graphically

1. Plot the feasible region defined by the constraints.
 2. If the feasible region is empty, then no solution exists.
 3. If the feasible region is unbounded goto Line 8. Otherwise, Goto Line 4.
 4. Plot the level sets of the objective function.
 5. For a maximization problem, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
 6. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem.
 7. **If the level set corresponding to the greatest (least) objective function value is parallel to a side of the polygon boundary next to the corner identified, then there are infinitely many alternative optimal solutions and any point on this side may be chosen as an optimal solution.**
 8. (The feasible region is unbounded): Plot the level sets of the objective function.
 9. If the level sets intersect the feasible region at larger and larger (smaller and smaller for a minimization problem), then the problem is unbounded and the solution is $+\infty$ ($-\infty$ for minimization problems).
 10. Otherwise, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
 11. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem. **If the level set corresponding to the greatest (least) objective function value is parallel to a side of the polygon boundary next to the corner identified, then there are infinitely many alternative optimal solutions and any point on this side may be chosen as an optimal solution.**
-

Exercise 4.9

Does the following problem have a bounded solution? Why?

$$\left\{ \begin{array}{l} \min z(x_1, x_2) = 2x_1 - x_2 \\ \text{s.t. } x_1 - x_2 \leq 1 \\ \quad 2x_1 + x_2 \geq 6 \\ \quad x_1, x_2 \geq 0 \end{array} \right. \quad (4.3)$$

[Hint: Use Figure 4.5 and Algorithm 4.]

Exercise 4.10

Modify the objective function in Example 4.4 or Example 4.4 to produce a problem with an infinite number of solutions.

Exercise 4.11

Modify the objective function in Exercise 4.4 to produce a **minimization** problem that has a finite solution. Draw the feasible region and level curves of the objective to “prove” your example works. [Hint: Think about what direction of increase is required for the level sets of $z(x_1, x_2)$ (or find a trick using Exercise ??).]

4.5 Formal Mathematical Statements

Vectors and Linear and Convex Combinations

Vectors: Vector \mathbf{n} has n -elements and represents a point (or an arrow from the origin to the point, denoting a direction) in \mathcal{R}^n space (Euclidean or real space). Vectors can be expressed as either row or column vectors.

Vector Addition: Two vectors of the same size can be added, componentwise, e.g., for vectors $\mathbf{a} = (2, 3)$ and $\mathbf{b} = (3, 2)$, $\mathbf{a} + \mathbf{b} = (2 + 3, 3 + 2) = (5, 5)$.

Scalar Multiplication: A vector can be multiplied by a scalar k (constant) component-wise. If $k > 0$ then this does not change the direction represented by the vector, it just scales the vector.

Inner or Dot Product: Two vectors of the same size can be multiplied to produce a real number. For example, $\mathbf{ab} = 2 * 3 + 3 * 2 = 10$.

Linear Combination: The vector \mathbf{b} is a **linear combination** of $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$ if $\mathbf{b} = \sum_{i=1}^k \lambda_i \mathbf{a}_i$ for $\lambda_1, \lambda_2, \dots, \lambda_k \in \mathcal{R}$. If $\lambda_1, \lambda_2, \dots, \lambda_k \in \mathcal{R}_{\geq 0}$ then \mathbf{b} is a *non-negative linear combination* of $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$.

Convex Combination: The vector \mathbf{b} is a **convex combination** of $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$ if $\mathbf{b} = \sum_{i=1}^k \lambda_i \mathbf{a}_i$, for $\lambda_1, \lambda_2, \dots, \lambda_k \in \mathcal{R}_{\geq 0}$ and $\sum_{i=1}^k \lambda_i = 1$. For example, any convex combination of two points will lie on the line segment between the points.

Linear Independence: Vectors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$ are *linearly independent* if the following linear combination $\sum_{i=1}^k \lambda_i \mathbf{a}_i = 0$ implies that $\lambda_i = 0$, $i = 1, 2, \dots, k$. In \mathcal{R}^2 two vectors are only linearly dependent if they lie on the same line. Can you have three linearly independent vectors in \mathcal{R}^2 ?

Spanning Set: Vectors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$ span \mathcal{R}^m if any vector in \mathcal{R}^m can be represented as a linear combination of $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$, i.e., $\sum_{i=1}^m \lambda_i \mathbf{a}_i$ can represent any vector in \mathcal{R}^m .

Basis: Vectors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$ form a basis of \mathcal{R}^m if they span \mathcal{R}^m and any smaller subset of these vectors does not span \mathcal{R}^m . Vectors $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$ can only form a basis of \mathcal{R}^m if $k = m$ and they are linearly independent.

Convex and Polyhedral Sets

Convex Set: Set \mathcal{S} in \mathbb{R}^n is a *convex set* if a line segment joining any pair of points \mathbf{a}_1 and \mathbf{a}_2 in \mathcal{S} is completely contained in \mathcal{S} , that is, $\lambda\mathbf{a}_1 + (1 - \lambda)\mathbf{a}_2 \in \mathcal{S}, \forall \lambda \in [0, 1]$.

Hyperplanes and Half-Spaces: A hyperplane in \mathbb{R}^n divides \mathbb{R}^n into 2 half-spaces (like a line does in \mathbb{R}^2). A hyperplane is the set $\{\mathbf{x} : \mathbf{p}\mathbf{x} = k\}$, where \mathbf{p} is the gradient to the hyperplane (i.e., the coefficients of our linear expression). The corresponding half-spaces is the set of points $\{\mathbf{x} : \mathbf{p}\mathbf{x} \geq k\}$ and $\{\mathbf{x} : \mathbf{p}\mathbf{x} \leq k\}$.

Polyhedral Set: A *polyhedral set* (or polyhedron) is the set of points in the intersection of a finite set of half-spaces. Set $\mathcal{S} = \{\mathbf{x} : \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0\}$, where \mathbf{A} is an $m \times n$ matrix, \mathbf{x} is an n -vector, and \mathbf{b} is an m -vector, is a *polyhedral set* defined by $m + n$ hyperplanes (i.e., the intersection of $m + n$ half-spaces).

- Polyhedral sets are convex.
- A polytope is a bounded polyhedral set.
- A polyhedral cone is a polyhedral set where the hyperplanes (that define the half-spaces) pass through the origin, thus $\mathcal{C} = \{\mathbf{x} : \mathbf{A}\mathbf{x} \leq 0\}$ is a polyhedral cone.

Edges and Faces: An *edge* of a polyhedral set \mathcal{S} is defined by $n - 1$ hyperplanes, and a *face* of \mathcal{S} by one or more defining hyperplanes of \mathcal{S} , thus an extreme point and an edge are faces (an extreme point is a zero-dimensional face and an edge a one-dimensional face). In \mathbb{R}^2 faces are only edges and extreme points, but in \mathbb{R}^3 there is a third type of face, and so on...

Extreme Points: $\mathbf{x} \in \mathcal{S}$ is an extreme point of \mathcal{S} if:

Definition 1: \mathbf{x} is not a convex combination of two other points in \mathcal{S} , that is, all line segments that are completely in \mathcal{S} that contain \mathbf{x} must have \mathbf{x} as an endpoint.

Definition 2: \mathbf{x} lies on n linearly independent defining hyperplanes of \mathcal{S} .

If more than n hyperplanes pass through an extreme points then it is a degenerate extreme point, and the polyhedral set is considered degenerate. This just adds a bit of complexity to the algorithms we will study, but it is quite common.

Unbounded Sets:

Rays: A ray in \mathbb{R}^n is the set of points $\{\mathbf{x} : \mathbf{x}_0 + \lambda\mathbf{d}, \lambda \geq 0\}$, where \mathbf{x}_0 is the vertex and \mathbf{d} is the direction of the ray.

Convex Cone: A *Convex Cone* is a convex set that consists of rays emanating from the origin. A convex cone is completely specified by its extreme directions. If \mathcal{C} is convex cone, then for any $\mathbf{x} \in \mathcal{C}$ we have

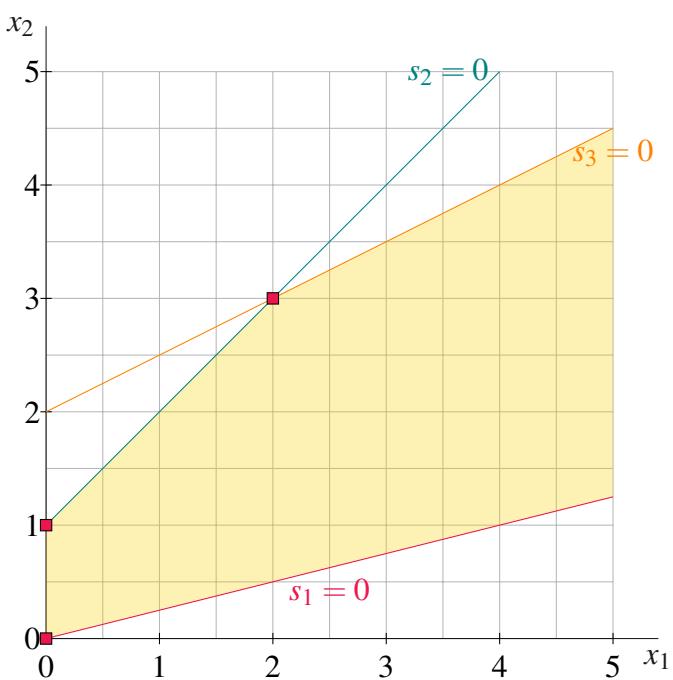
$$\lambda \mathbf{x} \in \mathcal{C}, \lambda \geq 0.$$

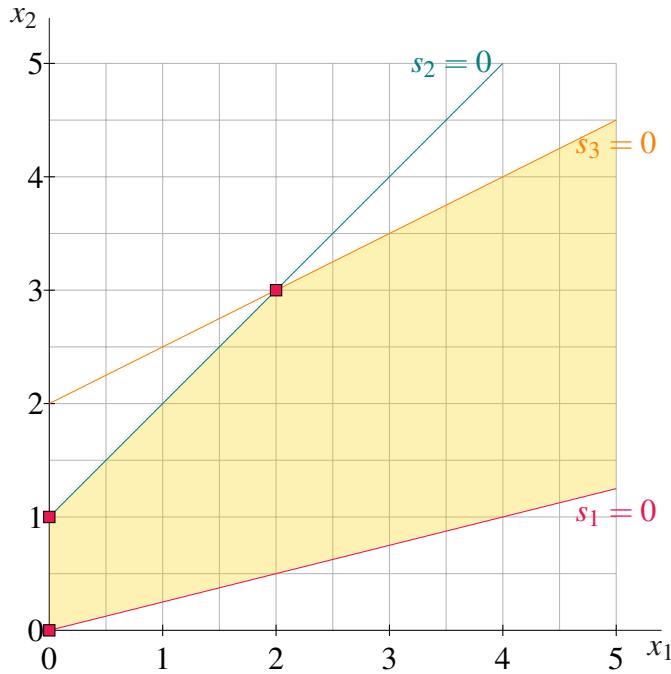
Unbounded Polyhedral Sets: If \mathcal{S} is unbounded, it will have *directions*. \mathbf{d} is a direction of \mathcal{S} only if $\mathbf{Ax} + \lambda \mathbf{d} \leq \mathbf{b}, \mathbf{x} + \lambda \mathbf{d} \geq 0$ for all $\lambda \geq 0$ and all $\mathbf{x} \in \mathcal{S}$. In other words, consider the ray $\{\mathbf{x} : \mathbf{x}_0 + \lambda \mathbf{d}, \lambda \geq 0\}$ in \mathbb{R}^n , where \mathbf{x}_0 is the vertex and \mathbf{d} is the direction of the ray. $\mathbf{d} \neq 0$ is a **direction** of set \mathcal{S} if for each \mathbf{x}_0 in \mathcal{S} the ray $\{\mathbf{x}_0 + \lambda \mathbf{d}, \lambda \geq 0\}$ also belongs to \mathcal{S} .

Extreme Directions: An *extreme direction* of \mathcal{S} is a direction that *cannot* be represented as positive linear combination of other directions of \mathcal{S} . A non-negative linear combination of extreme directions can be used to represent all other directions of \mathcal{S} . A polyhedral cone is completely specified by its extreme directions.

Let's define a procedure for finding the extreme directions, using the following LP's feasible region. Graphically, we can see that the extreme directions should follow the the $s_1 = 0$ (red) line and the $s_3 = 0$ (orange) line.

$$\begin{aligned} \max \quad & z = -5x_1 - x_2 \\ \text{s.t.} \quad & x_1 - 4x_2 + s_1 = 0 \\ & -x_1 + x_2 + s_2 = 1 \\ & -x_1 + 2x_2 + s_3 = 4 \\ & x_1, x_2, s_1, s_2, s_3 \geq 0. \end{aligned}$$



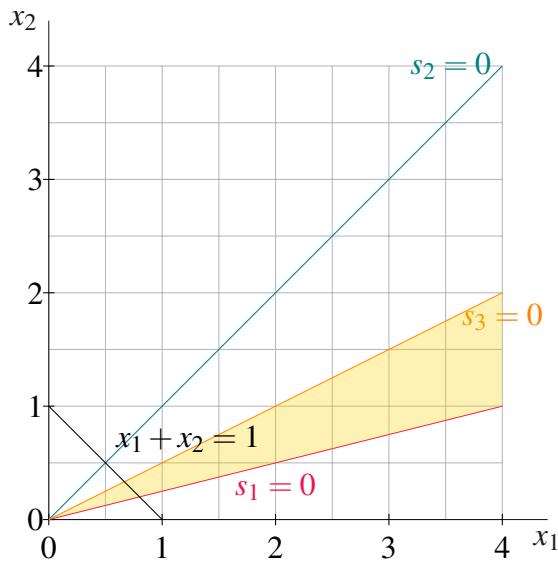


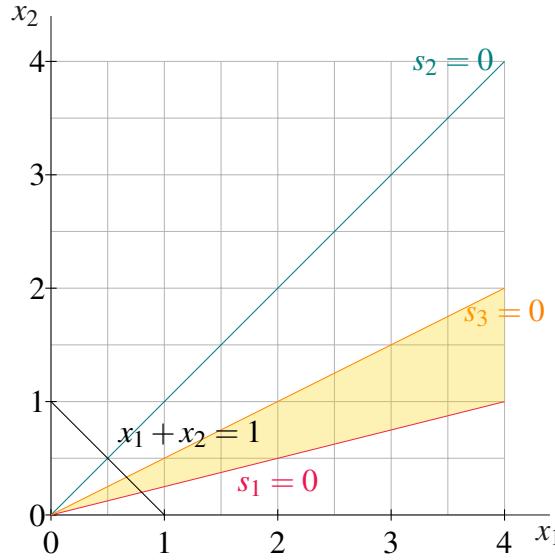
E.g., consider the $s_3 = 0$ (orange) line, to find the extreme direction start at extreme point $(2,3)$ and find another feasible point on the orange line, say $(4,4)$ and subtract $(2,3)$ from $(4,4)$, which yields $(2,1)$.

This is related to the slope in two-dimensions, as discussed in class, the rise is 1 and the run is 2. So this direction has a slope of $1/2$, but this does not carry over easily to higher dimensions where directions cannot be defined by a single number.

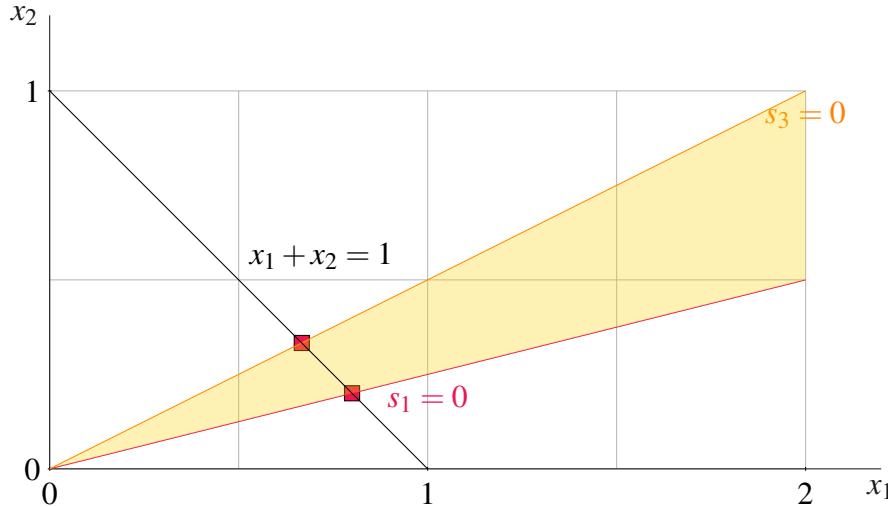
To find the extreme directions we can change the right-hand-side to $\mathbf{b} = 0$, which forms a polyhedral cone (in yellow), and then add the constraint $x_1 + x_2 = 1$. The intersection of the cone and $x_1 + x_2 = 1$ form a line segment.

$$\begin{aligned} \max \quad & z = -5x_1 - x_2 \\ \text{s.t.} \quad & x_1 - 4x_2 + s_1 = 0 \\ & -x_1 + x_2 + s_2 = 0 \\ & -x_1 + 2x_2 + s_3 = 0 \\ & x_1 + x_2 = 1 \\ & x_1, x_2, s_1, s_2, s_3 \geq 0. \end{aligned}$$





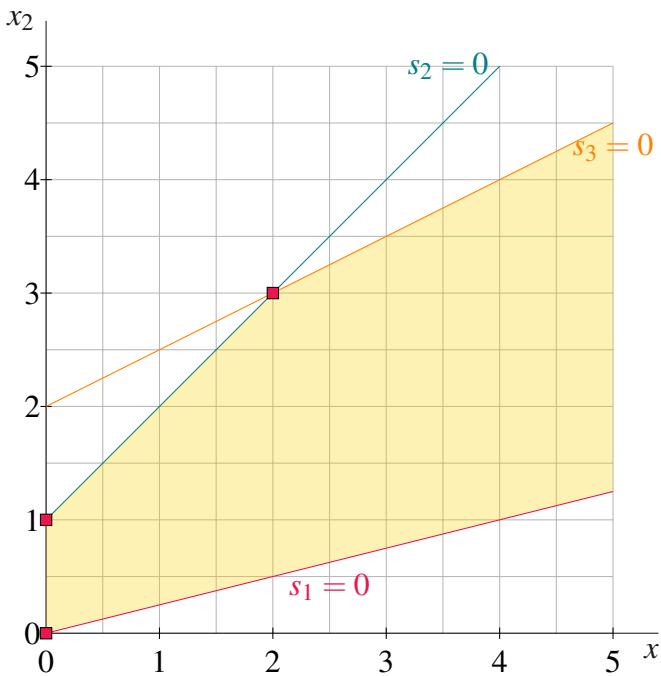
Magnifying for clarity, and removing the $s_2 = 0$ (teal) line, as it is redundant, and marking the extreme points of the new feasible region, $(4/5, 1/5)$ and $(2/3, 1/3)$, with red boxes, we have:



The extreme directions are thus $(4/5, 1/5)$ and $(2/3, 1/3)$.

Representation Theorem: Let $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k$ be the set of extreme points of \mathcal{S} , and if \mathcal{S} is unbounded, $\mathbf{d}_1, \mathbf{d}_2, \dots, \mathbf{d}_l$ be the set of extreme directions. Then any $\mathbf{x} \in \mathcal{S}$ is equal to a convex combination of the extreme points and a non-negative linear combination of the extreme directions: $\mathbf{x} = \sum_{j=1}^k \lambda_j \mathbf{x}_j + \sum_{j=1}^l \mu_j \mathbf{d}_j$, where $\sum_{j=1}^k \lambda_j = 1$, $\lambda_j \geq 0$, $\forall j = 1, 2, \dots, k$, and $\mu_j \geq 0$, $\forall j = 1, 2, \dots, l$.

$$\begin{aligned}
 \max \quad & z = -5x_1 - x_2 \\
 \text{s.t.} \quad & x_1 - 4x_2 + s_1 = 0 \\
 & -x_1 + x_2 + s_2 = 1 \\
 & -x_1 + 2x_2 + s_3 = 4 \\
 & x_1, x_2, s_1, s_2, s_3 \geq 0.
 \end{aligned}$$



Represent point $(1/2, 1)$ as a convex combination of the extreme points of the above LP. Find λ s to solve the following system of equations:

$$\lambda_1 \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \lambda_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \lambda_3 \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1 \end{bmatrix}$$

4.6 Graphical example

To motivate the subject of linear programming (LP), we begin with a planning problem that can be solved graphically.

Example 4.12: Lemonade Vendor

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 three dollars. Each unit of lemon juice gives a profit of two dollars. You have 6 lemons and 4 litres of water available. How many units of lemonade and lemon juice should you make to maximize profit?

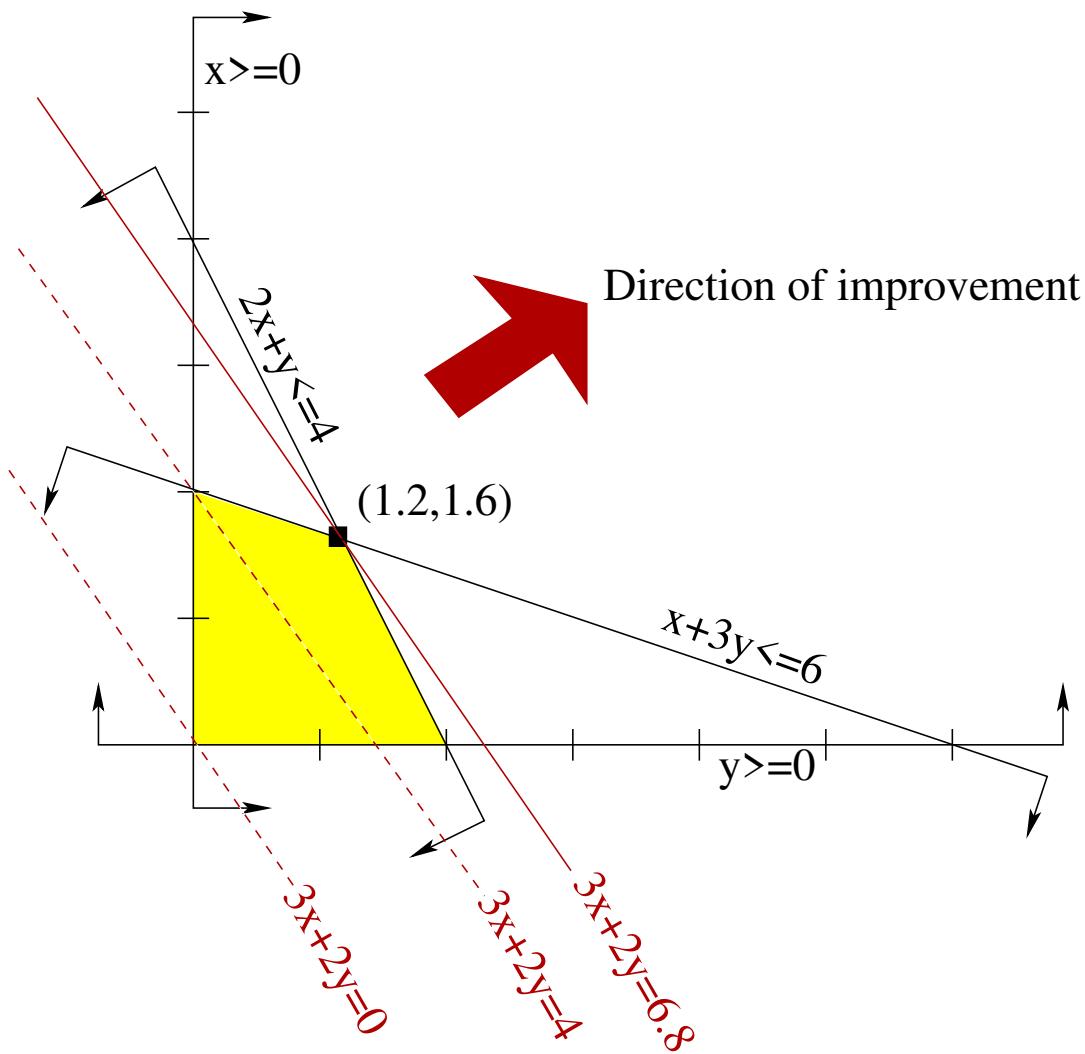
If we let x denote the number of units of lemonade to be made and let y denote the number of units of lemon juice to be made, then the profit is given by $3x + 2y$ dollars. We call $3x + 2y$ the objective function. Note that there are a number of constraints that x and y must satisfy. First of all, x and y should be nonnegative. The number of lemons needed to make x units of lemonade and y units of lemon juice is $x + 3y$ and cannot exceed 6. The number of litres of water needed to make

x units of lemonade and y units of lemon juice is $2x + y$ and cannot exceed 4. Hence, to determine the maximum profit, we need to maximize $3x + 2y$ subject to x and y satisfying the constraints $x + 3y \leq 6$, $2x + y \leq 4$, $x \geq 0$, and $y \geq 0$.

A more compact way to write the problem is as follows:

$$\begin{array}{ll} \text{maximize} & 3x + 2y \\ \text{subject to} & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x \geq 0 \\ & y \geq 0. \end{array}$$

We can solve this maximization problem graphically as follows. We first sketch the set of $\begin{bmatrix} x \\ y \end{bmatrix}$ satisfying the constraints, called the feasible region, on the (x, y) -plane. We then take the objective function $3x + 2y$ and turn it into an equation of a line $3x + 2y = z$ where z is a parameter. Note that as the value of z increases, the line defined by the equation $3x + 2y = z$ moves in the direction of the normal vector $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$. We call this direction the direction of improvement. Determining the maximum value of the objective function, called the optimal value, subject to the constraints amounts to finding the maximum value of z so that the line defined by the equation $3x + 2y = z$ still intersects the feasible region.



In the figure above, the lines with z at 0, 4 and 6.8 have been drawn. From the picture, we can see that if z is greater than 6.8, the line defined by $3x + 2y = z$ will not intersect the feasible region. Hence, the profit cannot exceed 6.8 dollars.

As the line $3x + 2y = 6.8$ does intersect the feasible region, 6.8 is the maximum value for the objective function. Note that there is only one point in the feasible region that intersects the line $3x + 2y = 6.8$, namely $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1.2 \\ 1.6 \end{bmatrix}$. In other words, to maximize profit, we want to make 1.2 units of lemonade and 1.6 units of lemon juice.

The above solution method can hardly be regarded as rigorous because we relied on a picture to conclude that $3x + 2y \leq 6.8$ for all $\begin{bmatrix} x \\ y \end{bmatrix}$ satisfying the constraints. But we can actually show this *algebraically*.

Note that multiplying both sides of the constraint $x + 3y \leq 6$ gives $0.2x + 0.6y \leq 1.2$, and multiplying both sides of the constraint $2x + y \leq 4$ gives $2.8x + 1.4y \leq 5.6$. Hence, any $\begin{bmatrix} x \\ y \end{bmatrix}$ that satisfies both $x + 3y \leq 6$ and $2x + y \leq 4$ must also satisfy $(0.2x + 0.6y) + (2.8x + 1.4y) \leq 1.2 + 5.6$, which simplifies to $3x + 2y \leq 6.8$ as desired! (Here, we used the fact that if $a \leq b$ and $c \leq d$, then $a + c \leq b + d$.)

Now, one might ask if it is always possible to find an algebraic proof like the one above for similar problems. If the answer is yes, how does one find such a proof? We will see answers to this question later on.

Before we end this segment, let us consider the following problem:

$$\begin{array}{ll} \text{minimize} & -2x + y \\ \text{subject to} & -x + y \leq 3 \\ & x - 2y \leq 2 \\ & x \geq 0 \\ & y \geq 0. \end{array}$$

Note that for any $t \geq 0$, $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} t \\ t \end{bmatrix}$ satisfies all the constraints. The value of the objective function at $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} t \\ t \end{bmatrix}$ is $-t$. As $t \rightarrow \infty$, the value of the objective function tends to $-\infty$. Therefore, there is no minimum value for the objective function. The problem is said to be unbounded. Later on, we will see how to detect unboundedness algorithmically.

As an exercise, check that unboundedness can also be established by using $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2t+2 \\ t \end{bmatrix}$ for $t \geq 0$.

Exercises

1. Sketch all $\begin{bmatrix} x \\ y \end{bmatrix}$ satisfying

$$x - 2y \leq 2$$

on the (x,y) -plane.

2. Determine the optimal value of

$$\begin{array}{ll} \text{Minimize} & x + y \\ \text{Subject to} & 2x + y \geq 4 \\ & x + 3y \geq 1. \end{array}$$

3. Show that the problem

$$\begin{array}{ll} \text{Minimize} & -x + y \\ \text{Subject to} & 2x - y \geq 0 \\ & x + 3y \geq 3 \end{array}$$

is unbounded.

4. Suppose that you are shopping for dietary supplements to satisfy your required daily intake of 0.40mg of nutrient M and 0.30mg of nutrient N . There are three popular products on the market. The costs and the amounts of the two nutrients are given in the following table:

	Product 1	Product 2	Product 3
Cost	\$27	\$31	\$24
Daily amount of M	0.16 mg	0.21 mg	0.11 mg
Daily amount of N	0.19 mg	0.13 mg	0.15 mg

You want to determine how much of each product you should buy so that the daily intake requirements of the two nutrients are satisfied at minimum cost. Formulate your problem as a linear programming problem, assuming that you can buy a fractional number of each product.

Solutions

1. The points (x,y) satisfying $x - 2y \leq 2$ are precisely those above the line passing through $(2,0)$ and $(0,-1)$.
2. We want to determine the minimum value z so that $x + y = z$ defines a line that has a nonempty intersection with the feasible region. However, we can avoid referring to a sketch by setting $x = z - y$ and substituting for x in the inequalities to obtain:

$$\begin{aligned} 2(z-y) + y &\geq 4 \\ (z-y) + 3y &\geq 1, \end{aligned}$$

or equivalently,

$$\begin{aligned} z &\geq 2 + \frac{1}{2}y \\ z &\geq 1 - 2y, \end{aligned}$$

Thus, the minimum value for z is $\min\{2 + \frac{1}{2}y, 1 - 2y\}$, which occurs at $y = -\frac{2}{5}$. Hence, the optimal value is $\frac{9}{5}$.

We can verify our work by doing the following. If our calculations above are correct, then an optimal solution is given by $x = \frac{11}{5}$, $y = -\frac{2}{5}$ since $x = z - y$. It is easy to check that this satisfies both inequalities and therefore is a feasible solution.

Now, taking $\frac{2}{5}$ times the first inequality and $\frac{1}{5}$ times the second inequality, we can infer the inequality $x + y \geq \frac{9}{5}$. The left-hand side of this inequality is precisely the objective function. Hence, no feasible solution can have objective function value less than $\frac{9}{5}$. But $x = \frac{11}{5}$, $y = -\frac{2}{5}$ is a feasible solution with objective function value equal to $\frac{9}{5}$. As a result, it must be an optimal solution.

Remark. We have not yet discussed how to obtain the multipliers $\frac{2}{5}$ and $\frac{1}{5}$ for inferring the inequality $x + y \geq \frac{9}{5}$. This is an issue that will be taken up later. In the meantime, think about how one could have obtained these multipliers for this particular exercise.

3. We could glean some insight by first making a sketch on the (x, y) -plane.

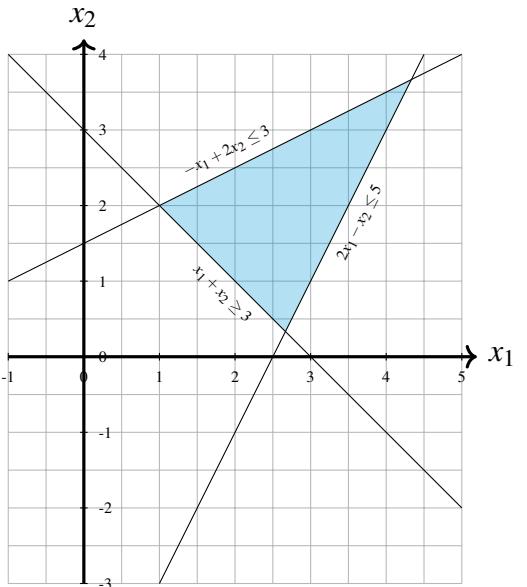
The line defined by $-x + y = z$ has x -intercept $-z$. Note that for $z \leq -3$, $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -z \\ 0 \end{bmatrix}$ satisfies both inequalities and the value of the objective function at $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -z \\ 0 \end{bmatrix}$ is z . Hence, there is no lower bound on the value of objective function.

4. Let x_i denote the amount of Product i to buy for $i = 1, 2, 3$. Then, the problem can be formulated as

$$\begin{aligned} & \text{minimize} && 27x_1 + 31x_2 + 24x_3 \\ & \text{subject to} && 0.16x_1 + 0.21x_2 + 0.11x_3 \geq 0.30 \\ & && 0.19x_1 + 0.13x_2 + 0.15x_3 \geq 0.40 \\ & && x_1, x_2, x_3 \geq 0. \end{aligned}$$

Remark. If one cannot buy fractional amounts of the products, the problem can be formulated as

$$\begin{aligned} & \text{minimize} && 27x_1 + 31x_2 + 24x_3 \\ & \text{subject to} && 0.16x_1 + 0.21x_2 + 0.11x_3 \geq 0.30 \\ & && 0.19x_1 + 0.13x_2 + 0.15x_3 \geq 0.40 \\ & && x_1, x_2, x_3 \geq 0. \\ & && x_1, x_2, x_3 \in \mathbb{Z}. \end{aligned}$$



2

²<https://tex.stackexchange.com/questions/75933/how-to-draw-the-region-of-inequality>

5. Software - Excel

Resources

- <https://www.excel-easy.com/data-analysis/solver.html>
- Excel Solver - Introduction on Youtube
- Some notes from MIT

5.0.1. Excel Solver

5.0.2. Videos

Solving a linear program Optimal product mix Set Cover

Introduction to Designing Optimization Models Using Excel Solver

Traveling Salesman Problem

Also Travelin Salesman Problem

Multiple Traveling Salesman Problem

Shortest Path

5.0.3. Links

Loan Example

Several Examples including TSP

6. Software - Python

Outcomes

- *Install and get python up and running in some form*
- *Introduce basic python skills that will be helpful*

Resources

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

6.1 Installing and Managing Python

Installing and Managing Python

Lab Objective: *One of the great advantages of Python is its lack of overhead: it is relatively easy to download, install, start up, and execute. This appendix introduces tools for installing and updating specific packages and gives an overview of possible environments for working efficiently in Python.*

Installing Python via Anaconda

A *Python distribution* is a single download containing everything needed to install and run Python, together with some common packages. For this curriculum, we **strongly** recommend using the *Anaconda* distribution to install Python. Anaconda includes IPython, a few other tools for developing in Python, and a large selection of packages that are common in applied mathematics, numerical computing, and data science. Anaconda is free and available for Windows, Mac, and Linux.

Follow these steps to install Anaconda.

1. Go to <https://www.anaconda.com/download/>.
2. Download the **Python 3.6** graphical installer specific to your machine.
3. Open the downloaded file and proceed with the default configurations.

For help with installation, see <https://docs.anaconda.com/anaconda/install/>. This page contains links to detailed step-by-step installation instructions for each operating system, as well as information for updating and uninstalling Anaconda.

ACHTUNG!

This curriculum uses Python 3.6, **not** Python 2.7. With the wrong version of Python, some example code within the labs may not execute as intended or result in an error.

Managing Packages

A *Python package manager* is a tool for installing or updating Python packages, which involves downloading the right source code files, placing those files in the correct location on the machine, and linking the files to the Python interpreter. **Never** try to install a Python package without using a package manager (see <https://xkcd.com/349/>).

Conda

Many packages are not included in the default Anaconda download but can be installed via Anaconda's package manager, conda. See <https://docs.anaconda.com/anaconda/packages/pkg-docs> for the complete list of available packages. When you need to update or install a package, **always** try using conda first.

Command	Description
conda install <package-name>	Install the specified package.
conda update <package-name>	Update the specified package.
conda update conda	Update conda itself.
conda update anaconda	Update all packages included in Anaconda.
conda --help	Display the documentation for conda.

For example, the following terminal commands attempt to install and update matplotlib.

```
$ conda update conda          # Make sure that conda is up to date.
$ conda install matplotlib    # Attempt to install matplotlib.
$ conda update matplotlib     # Attempt to update matplotlib.
```

See <https://conda.io/docs/user-guide/tasks/manage-pkgs.html> for more examples.

NOTE

The best way to ensure a package has been installed correctly is to try importing it in IPython.

```
# Start IPython from the command line.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

# Try to import matplotlib.
In [1]: from matplotlib import pyplot as plt      # Success!
```

ACHTUNG!

Be careful not to attempt to update a Python package while it is in use. It is safest to update packages while the Python interpreter is not running.

Pip

The most generic Python package manager is called pip. While it has a larger package list, conda is the cleaner and safer option. Only use pip to manage packages that are not available through conda.

Command	Description
pip install package-name	Install the specified package.
pip install --upgrade package-name	Update the specified package.
pip freeze	Display the version number on all installed packages.
pip --help	Display the documentation for pip.

See https://pip.pypa.io/en/stable/user_guide/ for more complete documentation.

Workflows

There are several different ways to write and execute programs in Python. Try a variety of workflows to find what works best for you.

Text Editor + Terminal

The most basic way of developing in Python is to write code in a text editor, then run it using either the Python or IPython interpreter in the terminal.

There are many different text editors available for code development. Many text editors are designed specifically for computer programming which contain features such as syntax highlighting and error detection, and are highly customizable. Try installing and using some of the popular text editors listed below.

- Atom: <https://atom.io/>
- Sublime Text: <https://www.sublimetext.com/>
- Notepad++ (Windows): <https://notepad-plus-plus.org/>
- Geany: <https://www.geany.org/>
- Vim: <https://www.vim.org/>
- Emacs: <https://www.gnu.org/software/emacs/>

Once Python code has been written in a text editor and saved to a file, that file can be executed in the terminal or command line.

```
$ ls                               # List the files in the current directory.
hello_world.py
$ cat hello_world.py               # Print the contents of the file to the terminal.
print("hello, world!")
$ python hello_world.py           # Execute the file.
hello, world!

# Alternatively, start IPython and run the file.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run hello_world.py
hello, world!
```

IPython is an enhanced version of Python that is more user-friendly and interactive. It has many features that cater to productivity such as tab completion and object introspection.

NOTE

While Mac and Linux computers come with a built-in bash terminal, Windows computers do not. Windows does come with *Powershell*, a terminal-like application, but some commands in Powershell are different than their bash analogs, and some bash commands are missing from Powershell

altogether. There are two good alternatives to the bash terminal for Windows:

- Windows subsystem for linux: docs.microsoft.com/en-us/windows/wsl/.
- Git bash: <https://gitforwindows.org/>.

Jupyter Notebook

The Jupyter Notebook (previously known as IPython Notebook) is a browser-based interface for Python that comes included as part of the Anaconda Python Distribution. It has an interface similar to the IPython interpreter, except that input is stored in cells and can be modified and re-evaluated as desired. See <https://github.com/jupyter/jupyter/wiki/> for some examples.

To begin using Jupyter Notebook, run the command `jupyter notebook` in the terminal. This will open your file system in a web browser in the Jupyter framework. To create a Jupyter Notebook, click the **New** drop down menu and choose **Python 3** under the **Notebooks** heading. A new tab will open with a new Jupyter Notebook.

Jupyter Notebooks differ from other forms of Python development in that notebook files contain not only the raw Python code, but also formatting information. As such, Jupyter Notebook files cannot be run in any other development environment. They also have the file extension `.ipynb` rather than the standard Python extension `.py`.

Jupyter Notebooks also support Markdown—a simple text formatting language—and \LaTeX , and can embed images, sound clips, videos, and more. This makes Jupyter Notebook the ideal platform for presenting code.

Integrated Development Environments

An *integrated development environment* (IDEs) is a program that provides a comprehensive environment with the tools necessary for development, all combined into a single application. Most IDEs have many tightly integrated tools that are easily accessible, but come with more overhead than a plain text editor. Consider trying out each of the following IDEs.

- JupyterLab: <http://jupyterlab.readthedocs.io/en/stable/>
- PyCharm: <https://www.jetbrains.com/pycharm/>
- Spyder: <http://code.google.com/p/spyderlib/>
- Eclipse with PyDev: <http://www.eclipse.org/>, <https://www.pydev.org/>

See <https://realpython.com/python-ides-code-editors-guide/> for a good overview of these (and other) workflow tools.

6.2 NumPy Visual Guide

NumPy Visual Guide I
are straightforward. The slicing syntax, stacking 1- or 2-dimensional arrays, and

Data Access

The entries of a 2-D array are indexed by a row index, a comma, and a column index.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$A[2,1] = \begin{bmatrix} \times \\ \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Slicing

A lone colon extracts an entire entry up to (but not including) the end. This means “everything up to (but not including) the end”.

$$A[1] = A[1,:] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$A[:,2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times \\ \times \end{bmatrix}$$

$$A[1:,:2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$A[1:-1,1:-1] = \begin{bmatrix} \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$A = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

$$B = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

$$\text{np.hstack}((A, B, A)) = \begin{bmatrix} \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \end{bmatrix}$$

$$\text{np.vstack}((A, B, A)) = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ * & * & * \\ * & * & * \\ * & * & * \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$x = [\times \quad \times \quad \times \quad \times]$$

$$y = [* \quad * \quad * \quad *]$$

$$\text{np.hstack}((x, y, x)) = [\times \quad \times \quad \times \quad \times \quad * \quad * \quad * \quad * \quad \times \quad \times \quad \times \quad \times]$$

$$\text{np.vstack}((x, y, x)) = \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix}$$

$$\text{np.column_stack}((x, y, x)) = \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}$$

The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See <http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html> for more in-depth examples and broadcasting rules.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad x = [10 \ 20 \ 30]$$

$$A + x = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$A + x.reshape((1, -1)) = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$A.sum(axis=0) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = [4 \ 8 \ 12 \ 16]$$

$$A.sum(axis=1) = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = [10 \ 10 \ 10 \ 10]$$

6.3 Plot Customization and Matplotlib Syntax Guide

Matplotlib Customization

Lab Objective: *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. For an introduction to Matplotlib, see lab ??.*

Colors

By default, every plot is assigned a different color specified by the “color cycle”. It can be overwritten by specifying what color is desired in a few different ways.

-

Matplotlib recognizes some basic built-in colors.

Code	Color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

The following displays how these colors can be implemented. The result is displayed in Figure 6.1.

```
1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 colors = np.array(["k", "g", "b", "r", "c", "m", "y", "w"])
5 x = np.linspace(0, 5, 1000)
6 y = np.ones(1000)
7
8 for i in xrange(8):
9     plt.plot(x, i*y, colors[i], linewidth=18)
10
11 plt.ylim([-1, 8])
12 plt.savefig("colors.pdf", format='pdf')
13 plt.clf()
```

colors.py

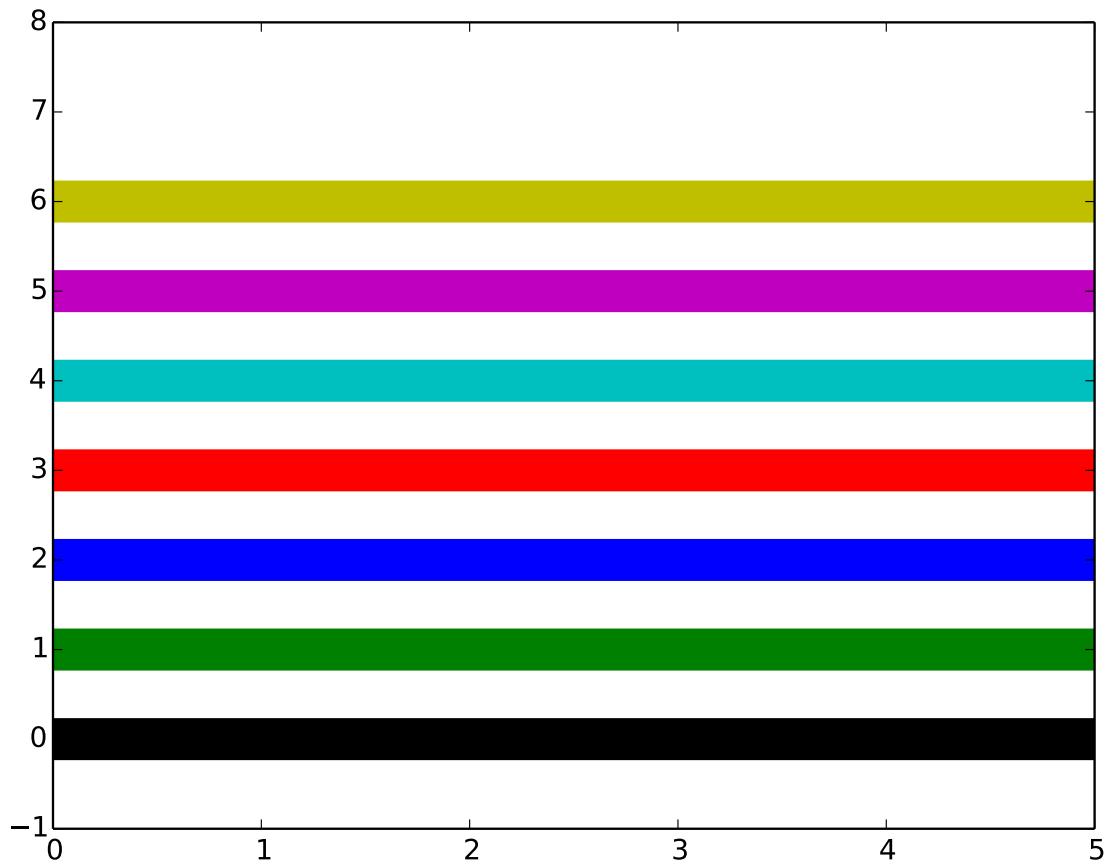


Figure 6.1: A display of all the built-in colors.

There are many other ways to specific colors. A popular method to access colors that are not built-in is to use a RGB tuple. Colors can also be specified using an html hex string or its associated html color name like "DarkOliveGreen", "FireBrick", "LemonChiffon", "MidnightBlue", "PapayaWhip", or "SeaGreen".

Window Limits

You may have noticed the use of `plt.ylim([ymin, ymax])` in the previous code. This explicitly sets the boundary of the y-axis. Similarly, `plt.xlim([xmin, xmax])` can be used to set the boundary of the x-axis. Doing both commands simultaneously is possible with the `plt.axis([xmin, xmax, ymin, ymax])`. Remember that these commands must be executed after the plot.

Lines

Thickness

You may have noticed that the width of the lines above seemed thin considering we wanted to inspect the line color. `linewidth` is a keyword argument that is defaulted to be `None` but can be given any real number to adjust the line width.

The following displays how `linewidth` is implemented. It is displayed in Figure 6.2.

```
1 lw = np.linspace(.5, 15, 8)
2
3 for i in xrange(8):
4     plt.plot(x, i*y, colors[i], linewidth=lw[i])
5
6 plt.ylim([-1, 8])
7 plt.show()
```

`linewidth.py`

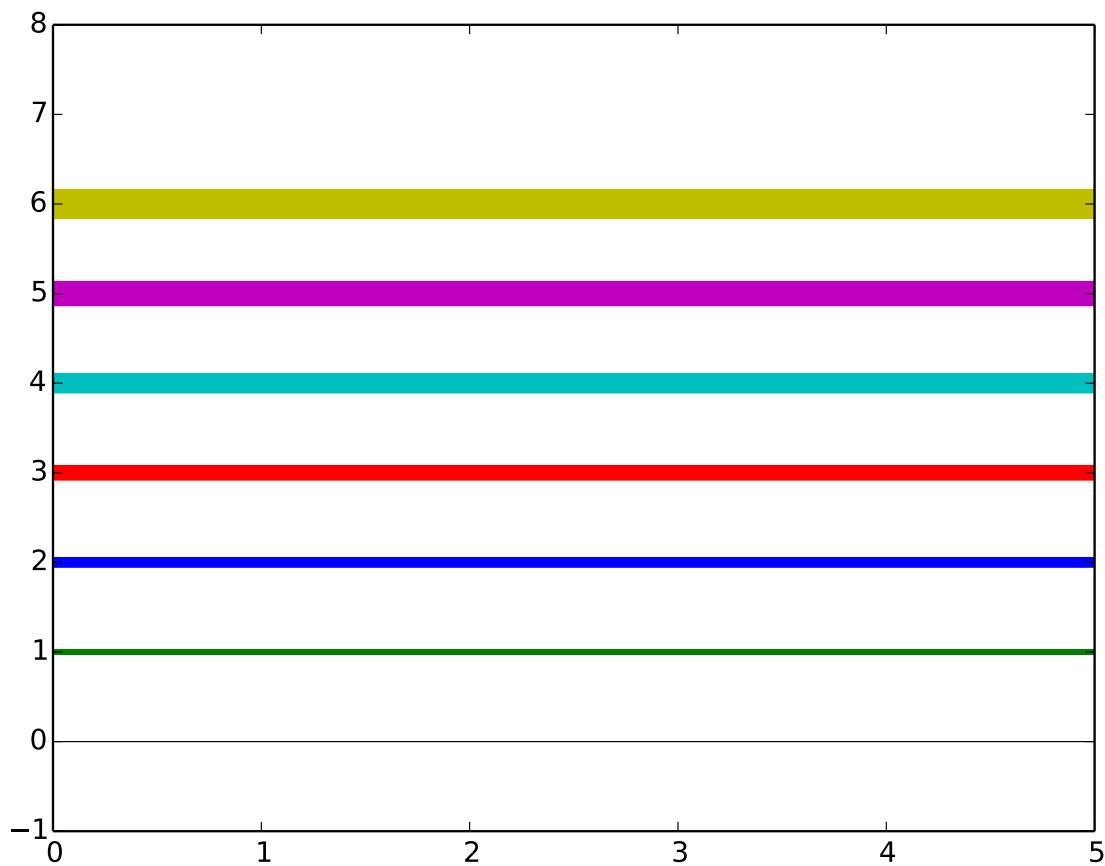


Figure 6.2: plot of varying linewidths.

Style

By default, plots are drawn with a solid line. The following are accepted format string characters to indicate line style.

character	description
-	solid line style
--	dashed line style
-.	dash-dot line style
:	dotted line style
.	point marker
,	pixel marker
o	circle marker
v	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker
3	tri_left marker
4	tri_right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker
+	plus marker
x	x marker
D	diamond marker
d	thin_diamond marker
	vline marker
-	hline marker

The following displays how `linestyle` can be implemented. It is displayed in Figure 6.3.

```

1 x = np.linspace(0, 5, 10)
2 y = np.ones(10)
ls = np.array(['-.', ':', 'o', 's', '*', 'H', 'x', 'D'])
4
5 for i in xrange(8):
6     plt.plot(x, i*y, colors[i]+ls[i])
8
9 plt.axis([-1, 6, -1, 8])
plt.show()

```

linestyle.py

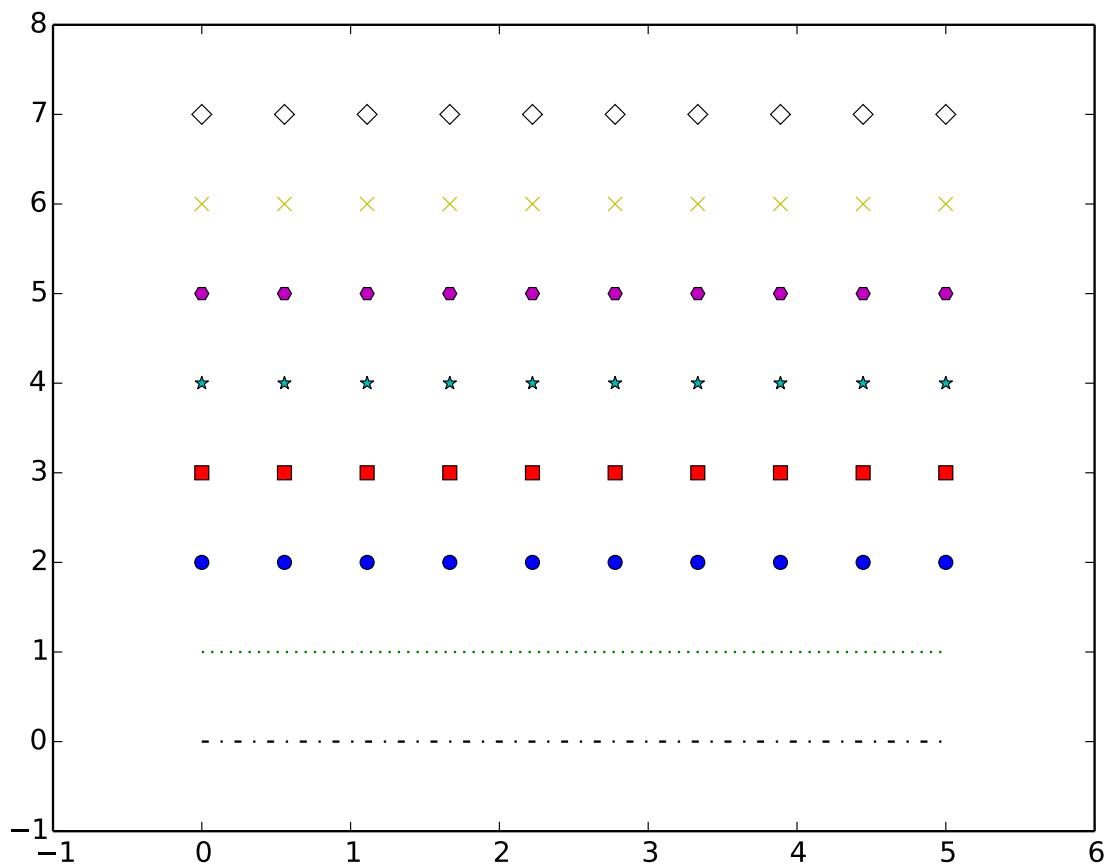


Figure 6.3: plot of varying linestyles.

Text

It is also possible to add text to your plots. To label your axes, the `plt.xlabel()` and the `plt.ylabel()` can both be used. The function `plt.title()` will add a title to a plot. If you are working with subplots, this command will add a title to the subplot you are currently modifying. To add a title above the entire figure, use `plt.suptitle()`.

All of the `text()` commands can be customized with `fontsize` and `color` keyword arguments.

We can add these elements to our previous example. It is displayed in Figure 6.4.

```

1 for i in xrange(8):
2     plt.plot(x, i*y, colors[i]+ls[i])
4 plt.title("My Plot of Varying Linestyles", fontsize = 20, color = "gold")

```

```
plt.xlabel("x-axis", fontsize = 10, color = "darkcyan")
6 plt.ylabel("y-axis", fontsize = 10, color = "darkcyan")
8 plt.axis([-1, 6, -1, 8])
plt.show()
```

text.py

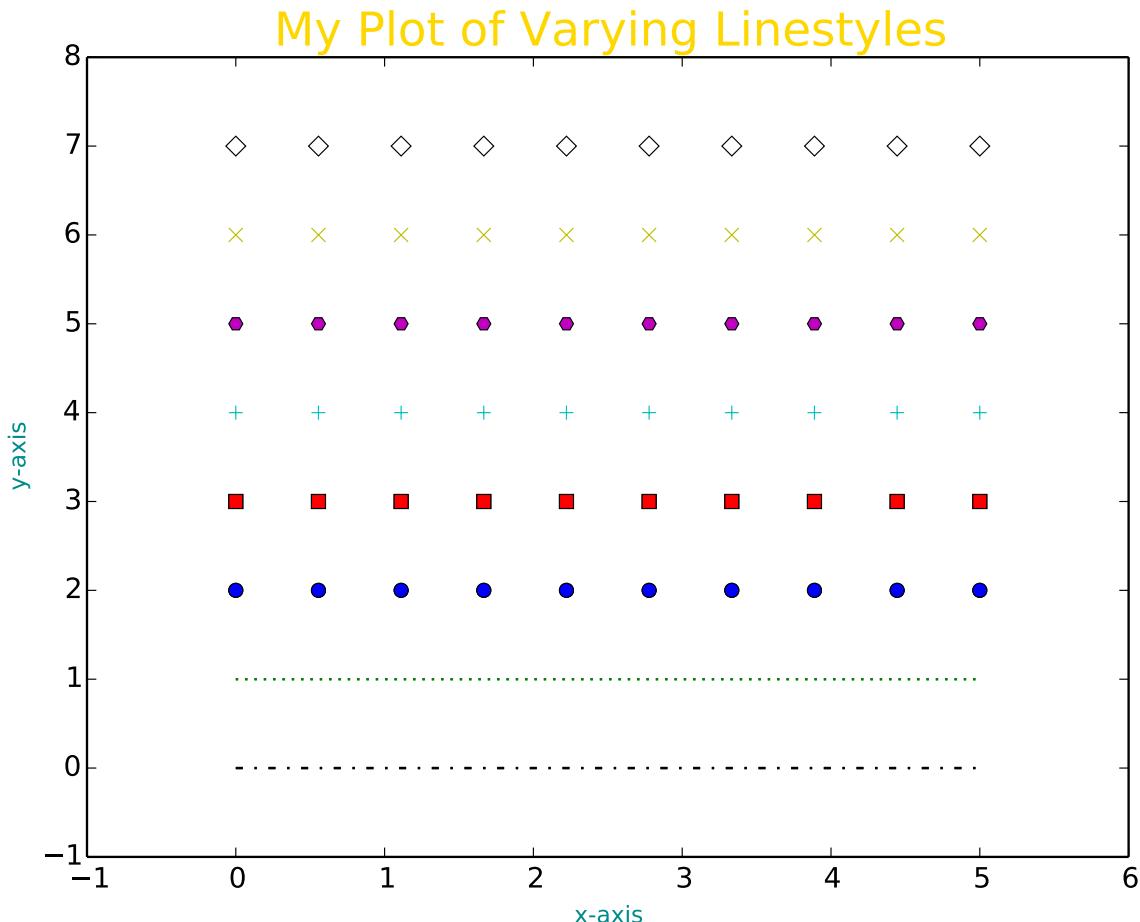


Figure 6.4: plot of varying linestyles using text labels.

See <http://matplotlib.org> for Matplotlib documentation.

6.4 Networkx - A Python Graph Algorithms Package

6.5 PuLP - An Optimization Modeling Tool for Python

Outcomes

- *Install and import PuLP*
- *Run basic first PuLP model*
- *Run "advanced" PuLP model using the algebraic modeling approach and importing data.*
- *Explore PuLP objects and possibilities*
- *Solve a Multi-Objective problem*

Resources

- *Documentation*
- *PyPi installation*
- *Examples*
- *Blog with tutorial*

PuLP is an optimization modeling language that is written for Python. It is free and open source. Yay! See Section ?? for a discussion of other options for implementing your optimization problem. PuLP is convenient for its simple syntax and easy installation.

Key benefits of using an algebraic modeling language like PuLP over Excel

- Easily readable models
- Precompute parameters within Python
- Reuse of common optimization models without recreating the equations

We will follow the introduction to pulp Jupyter Notebook Tutorial and the following application with a cleaner implementation.

6.5.1. Installation

Open a Jupyter notebook. In one of the cells, run the following command, based on which system you are running. It will take a minute to load and download the package.

```
[ ]: ## Install pulp (on windows)
!pip install pulp
```

```
[ ]: # on a mac
pip install pulp
```

```
[ ]: # on the VT ARC servers
import sys
!{sys.executable} -m pip install pulp
```

Installation (Continued) Now restart the kernel of your notebook (find the tab labeled Kernel in your Jupyter notebook, and in the drop down, select restart).

6.5.2. Example Problem

6.5.2.1. Product Mix Problem

$$\text{maximize} \quad Z = 3X_1 + 2X_2 \quad (\text{Objective function}) \quad (1.1)$$

$$\text{subject to} \quad 10X_1 + 5X_2 \leq 300 \quad (\text{Constraint 1}) \quad (1.2)$$

$$4X_1 + 4X_2 \leq 160 \quad (\text{Constraint 2}) \quad (1.3)$$

$$2X_1 + 6X_2 \leq 180 \quad (\text{Constraint 3}) \quad (1.4)$$

$$\text{and} \quad X_1, X_2 \geq 0 \quad (\text{Non-negative}) \quad (1.5)$$

OPTIMIZATION WITH PULP

```
[1]: from pulp import *

# Define problem
prob = LpProblem(name='Product_Mix_Problem', sense=LpMaximize)

# Create decision variables and non-negative constraint
x1 = LpVariable(name='X1', lowBound=0, upBound=None, cat='Continuous')
x2 = LpVariable(name='X2', lowBound=0, upBound=None, cat='Continuous')
```

```
# Set objective function
prob += 3*x1 + 2*x2

# Set constraints
prob += 10*x1 + 5*x2 <= 300
prob += 4*x1 + 4*x2 <= 160
prob += 2*x1 + 6*x2 <= 180

# Solving problem
prob.solve()
print('Status', LpStatus[prob.status])
```

Status Optimal

```
[2]: print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())

for v in prob.variables():
    print(v.name, ': ', v.value())
```

Status: Optimal
 Objective value: 100.0
 X1 : 20.0
 X2 : 20.0

6.5.3. Things we can do

```
[3]: # print the problem
prob
```

```
[3]: Product_Mix_Problem:
MAXIMIZE
3*X1 + 2*X2 + 0
SUBJECT TO
_C1: 10 X1 + 5 X2 <= 300

_C2: 4 X1 + 4 X2 <= 160

_C3: 2 X1 + 6 X2 <= 180

VARIABLES
```

```
X1 Continuous  
X2 Continuous
```

```
[4]: # get the objective function  
prob.objective.value()
```

```
[4]: 100.0
```

```
[5]: # get list of the variables  
prob.variables()
```

```
[5]: [X1, X2]
```

```
[6]: for v in prob.variables():  
    print(f'{v}: {v.varValue}')
```

```
X1: 20.0
```

```
X2: 20.0
```

6.5.3.1. Exploring the variables

```
[7]: v = prob.variables()[0]
```

```
[9]: v.name
```

```
[9]: 'X1'
```

```
[10]: v.value()
```

```
[10]: 20.0
```

```
[11]: v.varValue
```

```
[11]: 20.0
```

6.5.3.2. Other things you can do

```
[12]: # get list of the constraints
prob.constraints
```

```
[12]: OrderedDict([('C1', 10*X1 + 5*X2 + -300 <= 0),
                  ('C2', 4*X1 + 4*X2 + -160 <= 0),
                  ('C3', 2*X1 + 6*X2 + -180 <= 0)])
```

```
[13]: prob.to_dict()
```

```
[13]: {'objective': {'name': 'OBJ',
                   'coefficients': [{'name': 'X1', 'value': 3}, {'name': 'X2', 'value': 2}],},
       'constraints': [{'sense': -1,
                        'pi': 0.2,
                        'constant': -300,
                        'name': None,
                        'coefficients': [{'name': 'X1', 'value': 10}, {'name': 'X2', 'value': 5}],
                        'sense': -1,
                        'pi': 0.25,
                        'constant': -160,
                        'name': None,
                        'coefficients': [{'name': 'X1', 'value': 4}, {'name': 'X2', 'value': 4}],
                        'sense': -1,
                        'pi': -0.0,
                        'constant': -180,
                        'name': None,
                        'coefficients': [{'name': 'X1', 'value': 2}, {'name': 'X2', 'value': 6}]},
                     'variables': [{"lowBound": 0,
                                    'upBound': None,
                                    'cat': 'Continuous',
                                    'varValue': 20.0,
                                    'dj': -0.0,
                                    'name': 'X1'},
                                   {"lowBound": 0,
                                    'upBound': None,
                                    'cat': 'Continuous',
                                    'varValue': 20.0,
                                    'dj': -0.0,
                                    'name': 'X2'}],
                     'parameters': {'name': 'Product_Mix_Problem',
                                    'sense': -1}}
```

```
'status': 1,
'sol_status': 1},
'sos1': [],
'sos2': []}
```

```
[15]: # Store problem information in a json
prob.to_json('Product_Mix_Problem.json')
```

6.5.4. Common issue

If you forget the `<=`, `==`, or `>=` when writing a constraint, you will silently overwrite the objective function instead of adding a constraint!

6.5.4.1. Transportation Problem

Transport programming is a special form of linear programming, and in general, the objective function is cost minimization. The formula form and applicable variables of the Transport Planning Act are as follows. When supply and demand match, the constraint becomes an equation, but when supply and demand do not match, the constraint becomes an inequality.

Sets: - J = set of demand nodes - I = set of supply nodes

Parameters:

- D_j : Demand at node j
- S_i : Supply from node i
- c_{ij} : cost per unit to send supply i to demand j

Variables:

- X_{ij} : Transport volume from supply i to demand j (units)
- Objective function:

$$\min \sum_{i=1}^n \sum_{j=1}^m c_{ij} x_{ij}$$

- Constraints:

$$\sum_{i=1}^n x_{ij} = S_i$$

$$\sum_{i=1}^m x_{ij} = D_j$$

$$x_{ij} \geq 0 \text{ for } i \in I, j \in J$$

6.5.4.2. Optimization with PuLP

Here we do a very basic implementation of the problem

```
[1]: from pulp import *

prob = LpProblem('Transportation_Problem', LpMinimize)

x11 = LpVariable('X11', lowBound=0)
x12 = LpVariable('X12', lowBound=0)
x13 = LpVariable('X13', lowBound=0)
x14 = LpVariable('X14', lowBound=0)
x21 = LpVariable('X21', lowBound=0)
x22 = LpVariable('X22', lowBound=0)
x23 = LpVariable('X23', lowBound=0)
x24 = LpVariable('X24', lowBound=0)
x31 = LpVariable('X31', lowBound=0)
x32 = LpVariable('X32', lowBound=0)
x33 = LpVariable('X33', lowBound=0)
x34 = LpVariable('X34', lowBound=0)

prob += 4*x11 + 5*x12 + 6*x13 + 8*x14 + 4*x21 + 7*x22 + 9*x23 + 2*x24 + 5*x31 + ↴
       8*x32 + 7*x33 + 6*x34

prob += x11 + x12 + x13 + x14 == 120
prob += x21 + x22 + x23 + x24 == 150
prob += x31 + x32 + x33 + x34 == 200

prob += x11 + x21 + x31 == 100
prob += x12 + x22 + x32 == 60
prob += x13 + x23 + x33 == 130
prob += x14 + x24 + x34 == 180

# Solving problem
prob.solve();
```

```
[2]: print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())

for v in prob.variables():
    print(v.name, ': ', v.value())
```

```
Status: Optimal
Objective value: 2130.0
X11 : 60.0
X12 : 60.0
X13 : 0.0
X14 : 0.0
X21 : 0.0
X22 : 0.0
X23 : 0.0
X24 : 150.0
X31 : 40.0
X32 : 0.0
X33 : 130.0
X34 : 30.0
```

6.5.4.3. Optimization with PuLP: Round 2!

We now use set notation for this implementation

```
[3]: from pulp import *

prob = LpProblem('Transportation_Problem', LpMinimize)

# Sets
n_suppliers = 3
n_buyers = 4

I = range(n_suppliers)
J = range(n_buyers)

routes = [(i, j) for i in I for j in J]

# Parameters
costs = [
    [4, 5, 6, 8],
    [4, 7, 9, 2],
    [5, 8, 7, 6]
]

supply = [120, 150, 200]
demand = [100, 60, 130, 180]
```

```

# Variables
x = LpVariable.dicts('X', routes, lowBound=0)

# Objective
prob += lpSum([x[i, j] * costs[i][j] for i in I for j in J])

# Constraints

## Supply Constraints
for i in range(n_suppliers):
    prob += lpSum([x[i, j] for j in J]) == supply[i], f"Supply{i}"

## Demand Constraints
for j in range(n_buyers):
    prob += lpSum([x[i, j] for i in I]) == demand[j], f"Demand{j}"

# Solving problem
prob.solve();

```

```
[4]: print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())

for v in prob.variables():
    print(v.name, ': ', v.value())
```

```

Status: Optimal
Objective value: 2130.0
X_(0,_0) : 60.0
X_(0,_1) : 60.0
X_(0,_2) : 0.0
X_(0,_3) : 0.0
X_(1,_0) : 0.0
X_(1,_1) : 0.0
X_(1,_2) : 0.0
X_(1,_3) : 150.0
X_(2,_0) : 40.0
X_(2,_1) : 0.0
X_(2,_2) : 130.0
X_(2,_3) : 30.0

```

6.5.5. Changing details of the problem

```
[5]: original_obj = prob.objective
val = prob.objective.value()
r = 1.2
```

```
[6]: prob += original_obj <= r*val, "Objective bound"
```

```
[7]: prob
```

[7]: Transportation_Problem:

MINIMIZE

$$4*X_{(0,0)} + 5*X_{(0,1)} + 6*X_{(0,2)} + 8*X_{(0,3)} + 4*X_{(1,0)} + 7*X_{(1,1)} + 9*X_{(1,2)} + 2*X_{(1,3)} + 5*X_{(2,0)} + 8*X_{(2,1)} + 7*X_{(2,2)} + 6*X_{(2,3)} + 0$$

SUBJECT TO

$$\text{Supply0: } X_{(0,0)} + X_{(0,1)} + X_{(0,2)} + X_{(0,3)} = 120$$

$$\text{Supply1: } X_{(1,0)} + X_{(1,1)} + X_{(1,2)} + X_{(1,3)} = 150$$

$$\text{Supply2: } X_{(2,0)} + X_{(2,1)} + X_{(2,2)} + X_{(2,3)} = 200$$

$$\text{Demand0: } X_{(0,0)} + X_{(1,0)} + X_{(2,0)} = 100$$

$$\text{Demand1: } X_{(0,1)} + X_{(1,1)} + X_{(2,1)} = 60$$

$$\text{Demand2: } X_{(0,2)} + X_{(1,2)} + X_{(2,2)} = 130$$

$$\text{Demand3: } X_{(0,3)} + X_{(1,3)} + X_{(2,3)} = 180$$

$$\begin{aligned} \text{Objective_bound: } & 4 X_{(0,0)} + 5 X_{(0,1)} + 6 X_{(0,2)} + 8 X_{(0,3)} \\ & + 4 X_{(1,0)} + 7 X_{(1,1)} + 9 X_{(1,2)} + 2 X_{(1,3)} + 5 X_{(2,0)} + 8 X_{(2,1)} \\ & + 7 X_{(2,2)} + 6 X_{(2,3)} \leq 2556 \end{aligned}$$

VARIABLES

$X_{(0,0)}$ Continuous
 $X_{(0,1)}$ Continuous
 $X_{(0,2)}$ Continuous
 $X_{(0,3)}$ Continuous
 $X_{(1,0)}$ Continuous
 $X_{(1,1)}$ Continuous
 $X_{(1,2)}$ Continuous
 $X_{(1,3)}$ Continuous

```
X_(2,_0) Continuous
X_(2,_1) Continuous
X_(2,_2) Continuous
X_(2,_3) Continuous
```

[8]: # Change the objective
prob += x[0,0] # minimize x[0,0]

```
/opt/anaconda3/envs/python377/lib/python3.7/site-packages/pulp/pulp.py:1544:
UserWarning: Overwriting previously set objective.
    warnings.warn("Overwriting previously set objective.")
```

[9]: prob.solve()

[9]: 1

[10]: LpStatus[prob.status]

[10]: 'Optimal'

[11]: print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())

for v in prob.variables():
 print(v.name, ': ', v.value())

```
Status: Optimal
Objective value:  0.0
X_(0,_0) :  0.0
X_(0,_1) :  60.0
X_(0,_2) :  60.0
X_(0,_3) :  0.0
X_(1,_0) :  100.0
X_(1,_1) :  0.0
X_(1,_2) :  0.0
X_(1,_3) :  50.0
X_(2,_0) :  0.0
X_(2,_1) :  0.0
X_(2,_2) :  70.0
X_(2,_3) :  130.0
```

[12]: original_obj

```
[12]: 4*X_(0,_0) + 5*X_(0,_1) + 6*X_(0,_2) + 8*X_(0,_3) + 4*X_(1,_0) + 7*X_(1,_1) +
9*X_(1,_2) + 2*X_(1,_3) + 5*X_(2,_0) + 8*X_(2,_1) + 7*X_(2,_2) + 6*X_(2,_3) + 0
```

```
[13]: original_obj.value()
```

```
[13]: 2430.0
```

6.5.6. Changing Constraint Coefficients

```
[14]: a = prob.constraints['Supply0']
```

```
[15]: a.changeRHS(500)
```

```
[16]: a
```

```
[16]: 1*X_(0,_0) + 1*X_(0,_1) + 1*X_(0,_2) + 1*X_(0,_3) + -500 = 0
```

```
[17]: prob.constraints['Supply0'].keys()
```

```
[17]: odict_keys([X_(0,_0), X_(0,_1), X_(0,_2), X_(0,_3)])
```

6.6 Multi Objective Optimization with PuLP

We consider two objectives and compute the pareto efficient frontier. We will implement the ε -constraint method. That is, we will add bounds based on an objective function and the optimize the alternate objective function.

6.6.0.1. Transportation Problem

Sets: - J = set of demand nodes - I = set of supply nodes

Parameters: - D_j : Demand at node j - S_i : Supply from node i - c_{ij} : cost per unit to send supply i to demand j

Variables: - x_{ij} : Transport volume from supply i to demand j (units)

- Objective function:

$$\min \left(obj1 = \sum_{i=1}^n \sum_{j=1}^m c_{ij}x_{ij}, \quad obj2 = x_{00} + x_{13} + x_{22} - x_{21} - x_{03} \right)$$

- Constraints:

$$\sum_{i=1}^n x_{ij} = S_i$$

$$\sum_{i=1}^m x_{ij} = D_j$$

- Decision variables:

$$x_{ij} \geq 0 \quad i \in I, j \in J$$

6.6.0.2. Initial Optimization with PuLP

```
[1]: from pulp import *

prob = LpProblem('Transportation_Problem', LpMinimize)

# Sets
n_suppliers = 3
n_buyers = 4

I = range(n_suppliers)
J = range(n_buyers)

routes = [(i, j) for i in I for j in J]

# Parameters
costs = [
    [4, 5, 6, 8],
    [4, 7, 9, 2],
    [5, 8, 7, 6]
]

supply = [120, 150, 200]
demand = [100, 60, 130, 180]

# Variables
x = LpVariable.dicts('X', routes, lowBound=0)

# Objectives
obj1 = lpSum([x[i, j] * costs[i][j] for i in I for j in J])
```

```

obj2 = x[0,0] + x[1,3] + x[2,2] - x[2,1] - x[0,3]

## start with first objective
prob += obj1

# Constraints

## Supply Constraints
for i in range(n_suppliers):
    prob += lpSum([x[i, j] for j in J]) == supply[i], f"Supply{i}"

## Demand Constraints
for j in range(n_buyers):
    prob += lpSum([x[i, j] for i in I]) == demand[j], f"Demand{j}"

# Solving problem
prob.solve();

```

```

[2]: print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())

for v in prob.variables():
    print(v.name, ': ', v.value())

```

Status: Optimal
 Objective value: 2130.0
 X_(0,_0) : 60.0
 X_(0,_1) : 60.0
 X_(0,_2) : 0.0
 X_(0,_3) : 0.0
 X_(1,_0) : 0.0
 X_(1,_1) : 0.0
 X_(1,_2) : 0.0
 X_(1,_3) : 150.0
 X_(2,_0) : 40.0
 X_(2,_1) : 0.0
 X_(2,_2) : 130.0
 X_(2,_3) : 30.0

```

[3]: # Record objective value
obj1_opt = obj1.value()
obj1_opt

```

[3]: 2130.0

```
[4]: # Add both objective values to a list and also the solution
obj1_vals = [obj1.value()]
obj2_vals = [obj2.value()]
feasible_points = [prob.variables()]
```

```
[5]: # Change objective functions and compute optimal objective value for obj2
prob += obj2
prob.solve()

obj2_opt = obj2.value()
obj2_opt
```

/opt/anaconda3/envs/python377/lib/python3.7/site-packages/pulp/pulp.py:1537:
UserWarning: Overwriting previously set objective.
 warnings.warn("Overwriting previously set objective.")

[5]: -180.0

```
[6]: # Append these values to the lists
obj1_vals.append(obj1.value())
obj2_vals.append(obj2.value())
feasible_points.append(prob.variables())
```

6.6.1. Creating the Pareto Efficient Frontier

```
[7]: import numpy as np

# Create an inequality for objective 1
prob += obj1 <= obj1_opt, "Objective_bound1"
obj_constraint = prob.constraints["Objective_bound1"]
```

```
[8]: # Set to optimize objective 2
prob += obj2
```

/opt/anaconda3/envs/python377/lib/python3.7/site-packages/pulp/pulp.py:1537:
UserWarning: Overwriting previously set objective.
 warnings.warn("Overwriting previously set objective.")

```
[9]: # Adjusting objective bound of objective 1
```

```
r_values = np.arange(1,2000,10)
for r in r_values:
    obj_constraint.changeRHS(r + obj1_opt)
    if 1 == prob.solve():
        obj1_vals.append(obj1.value())
        obj2_vals.append(obj2.value())
        feasible_points.append(prob.variables())

# Remove objective 1 constraint
obj_constraint.changeRHS(0)
obj_constraint.clear()
```

[10]: # Create constraint for objective 2

```
prob += obj2 <= obj2_opt, "Objective_bound2"
obj2_constraint = prob.constraints["Objective_bound2"]

# set objective to objective 1
prob += obj1
```

[11]: # Adjusting objective bound of objective 2

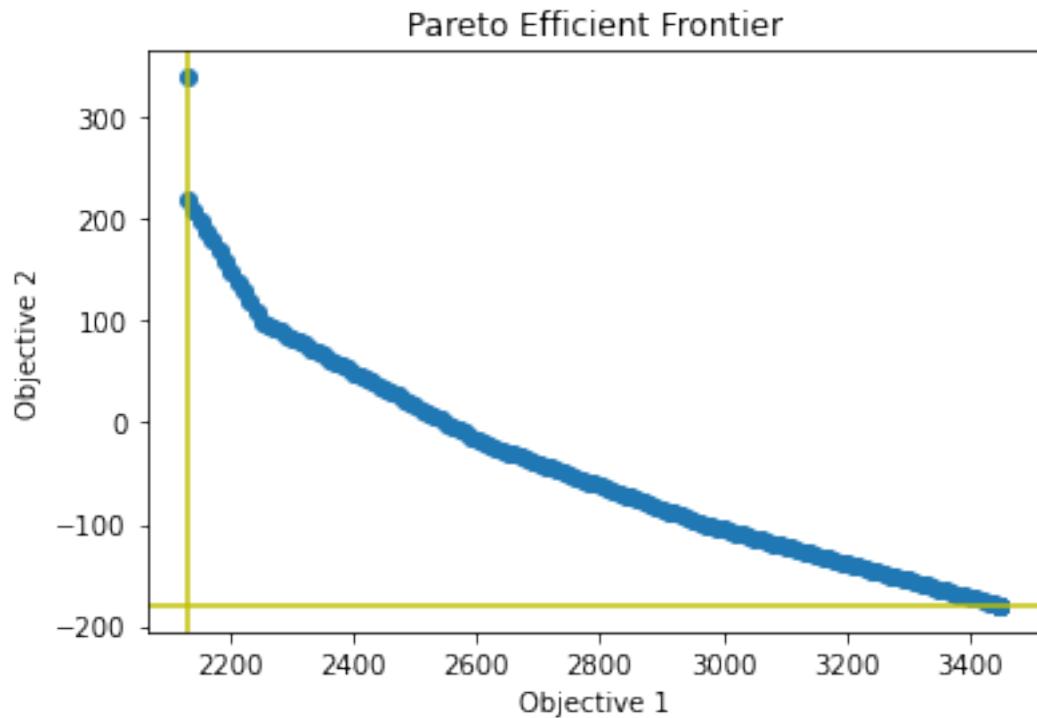
```
r_values = np.arange(1,400,5) # may need to adjust this
for r in r_values:
    obj2_constraint.changeRHS(r*obj2_opt)
    if 1 == prob.solve():
        obj1_vals.append(obj1.value())
        obj2_vals.append(obj2.value())
        feasible_points.append(prob.variables())

# Remove objective 2 constraint
obj2_constraint.changeRHS(0)
obj2_constraint.clear()
```

[12]: import matplotlib.pyplot as plt

```
plt.scatter(obj1_vals, obj2_vals)
plt.axvline(x=obj1_opt, color = 'y')
plt.axhline(y=obj2_opt, color = 'y')
plt.title("Pareto Efficient Frontier")
plt.xlabel("Objective 1")
plt.ylabel("Objective 2")
```

[12]: Text(0, 0.5, 'Objective 2')



6.7 Comments

This code is a bit inefficient. It probably computes more pareto points than needed.

6.8 Jupyter Notebooks

Resources

- <https://github.com/mathinmse/mathinmse.github.io/blob/master/Lecture-00B-Notebook-Basics.ipynb>
- <https://github.com/mathinmse/mathinmse.github.io/blob/master/Lecture-00C-Writing-In-Jupyter.ipynb>

6.9 Reading and Writing

[https://github.com/mathinmse/mathinmse.github.io/blob/master/
Lecture-10B-Reading-and-Writing-Data.ipynb](https://github.com/mathinmse/mathinmse.github.io/blob/master/Lecture-10B-Reading-and-Writing-Data.ipynb)

6.10 Python Crash Course

<https://github.com/rpmuller/PythonCrashCourse>

6.11 Gurobi

You can find lots great tutorial videos through Gurobi's youtube channel.

6.11.1. Introductory Gurobi Examples

1. Gurobi-first-model.ipynb
2. Gurobi-knapsack1.ipynb
3. Gurobi-knapsack2.ipynb
4. Gurobi-knapsack3.ipynb
5. Gurobi-knapsack-SOS1.ipynb
6. Gurobi-SOS2-piecewise-linear.ipynb

Gurobi on GitHub

Gurobi Modeling Examples

Gurobi Log Tools

6.12 Plots, Pandas, and Geopandas

6.12.1. Matplotlib.pyplot

This is perhaps the most commonly used package for plotting. The functionality is a lot like MatLab's plotting functionality.

6.12.2. Pandas

Pandas is the most used package for manipulating data in python. You can load data from excel, csv, json, or other types of files, and manipulate data in a variety of ways.

6.12.3. Geopandas

Geopandas is a fantastic python package that allows you to interact with geospatial data, make plots, and analyze data. This typically involves loading shapefiles that describe a region (or many regions) into a geopandas dataframe. These function much like a regular pandas dataframe, but has an extra column for the geometry of the data and has other functionality for these types of dataframes.

Here are some great tutorials.

- [tutorial](#)
- [tutorial github](#)

6.13 Google OR Tools

Google has been building their own set of optimization tools and marketing them to solve operations research problems. They have a variety of type of solvers and have a number of example problems where they use the appropriate solver to solve a problem. This is a constantly evolving set of tool. See the link below examples on vehicle routing, scheduling, and more.

[Google OR Tools](#)

Youtube! Video!

7. CASE STUDY - Designing a campground - Simplex method

7.1 DESIGNING A CAMPGROUND - SIMPLEX METHOD

The Simplex method is probably the classic method of solving constraint optimization problems. We will use this solution approach, sometimes in modified form, over and over in this class, not just in this chapter.

Outcomes

You will learn

- How to recognize linear programming (LP) problems
- Vocabulary of LP problems
- Graphical solution
- Algebraic solution
- Excel solution with the Excel Solver

7.1.1. Case Study Description - Campground

You are opening a campground in the Florida Keys, and you are trying to make as much money as possible. You are planning a mix of RV sites, tent sites, and yurts. Let's assume you already own 10 acres, and that you can make \$80/ day profit on each RV, \$20/ day profit on each tent, and \$200 profit on each Yurt. However, there are restrictions:

1. Infrastructure takes up 20% of your site
2. You can have 20 RVs per acre, or
3. You can have 40 tents per acre, or
4. You can have 10 yurts per acre
5. You have a budget of \$100,000. It costs you \$1000 to develop an RV site, \$200 for a tent site, and \$8000 for a yurt.

6. Maintenance for the bath houses etc. is 15 min/ week/camper unit. You can afford 70hrs/week in maintenance help
7. Zoning ordinance requires you to have at least 20 tent sites What is the best layout for your camp-ground, and how much profit can you make per day?

7.1.2. References

This case study was inspired by the Knights Key RV park in Florida. Read more about it here: <http://www.knightskeyrvresortandmarina.com/news/> <http://www.miaminewtimes.com/news/developers-plan-to-replace-rv-park-with-fivestar-resort-stirs-fears-hopes-in-keys-8038648>

7.1.3. Solution Approach - Two Variables

We will again first look at a simpler problem by ignoring the yurts and only considering RV spaces and tents.

Let r be the number of RVs, t the number of tents, and $P(r, t)$ the profit. Your goal is to maximize $P(r, t) = 80r + 20t$. This function is called the objective function. The variables r and t are called decision variables. You can see that in the current case $P(r, t)$ gets bigger if you increase r and/or t . If you picture a graph with r on the horizontal and t on the vertical axis, then the direction of increase for the objective function is to the top right.

Translating the relevant restrictions into equations gives

1. $r/20 + t/40 \leq 8$
2. $r \leq 160$
3. $t \leq 320$
4. $(r+t)/4 \leq 70$
5. $t \geq 20$
6. $r, t \geq 0$
7. $1,000r + 200t \leq 100,000$

Simplified

1. $2r + t \leq 320$
2. $r \leq 160$
3. $t \leq 320$
4. $5r + t \leq 500$
5. $r + t \leq 280$

$$6. t \geq 20$$

$$7. r, t \geq 0$$

These are called the functional constraints.

In addition, you can't have negative sites, so we have the non-negativity constraints

$$6. r \geq 0$$

$$7. t \geq 0.$$

A problem like the above with linear constraints and a linear objective function is called a linear programming problem.

7.1.4. Assumptions made about linear programming problem

PROPORTIONALITY For both the objective function and the constraints, a change in a decision variable will result in a proportional change in the objective function or constraint. (Note that this rules out any exponents on the decision variables other than 1.)

ADDITIVITY Both the objective function and the constraints are the sums of the respective changes in the decision variables (This means no multiplying different decision variables).

Basically, the proportionality and additivity assumptions are just fancy ways of saying that all functions in a linear programming problem are linear in the decision variables.

DIVISIBILITY We are assuming that our decision variables can be non-integer, i.e. may take on fractional values. Problems with an integer constraint are called integer programming problems, we will only touch on them briefly later. **Certainty** We act as if the value assigned to each parameter is known, precise, and constant over time. This is rarely the case, so we need to compensate for that by performing sensitivity analysis. Basically, we need to investigate how much it affects our solution if the parameters change.

7.1.5. Graphical Simplex solution procedure

We will start with some vocabulary:

- Feasible solution: A solution for which all constraints are satisfied, not necessarily an optimal Solution.
- Infeasible solution: A solution that violates at least one constraint.
- Optimal solution: a solution that optimizes (could be a minimum or maximum, depending on your problem) the objective function. There may or may not be an optimal solution.

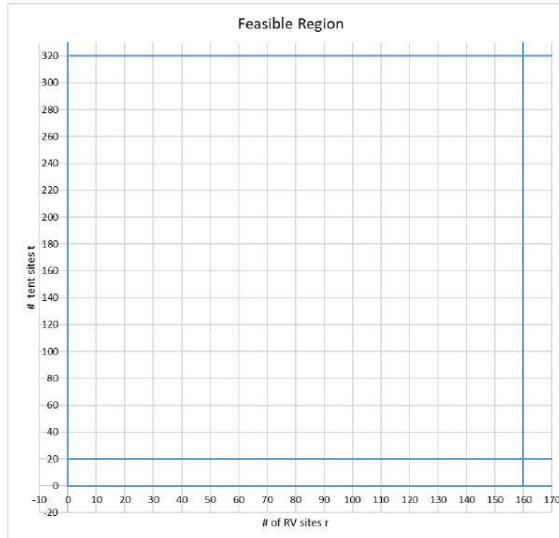
Feasible region: The set of all feasible solutions

- Corner point: the intersection of two or more constraints

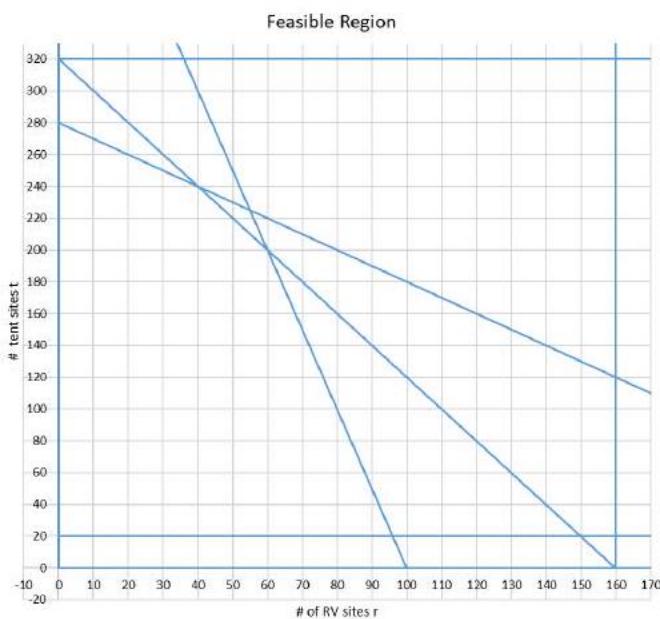
- Corner point feasible solution CPF solution: A solution that occurs at a corner of the feasible region

The first step in the graphical solution procedure is to draw the feasible region (note that this gets really ugly if you have three or more variables).

The non-negativity constraints mean that we are looking for a solution in the first quadrant only. The constraints 2) $r \leq 160$, 3) $t \leq 320$, and 7) $t \geq 20$ mean you have to stay left of the line $r = 160$, below the line $t = 320$ and above the line $t = 20$. You see that the constraint $t \geq 20$ dominates the constraint $t \geq 0$, so the latter is redundant.

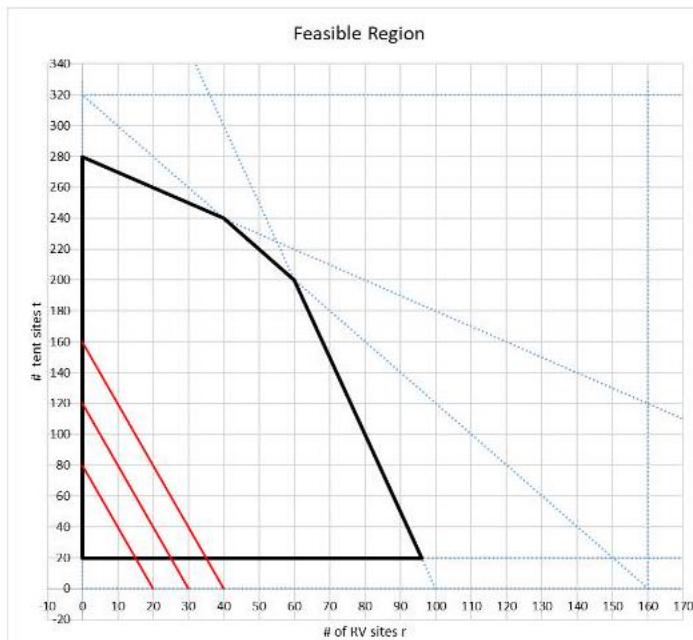


Adding the remaining constraints yields this graph: Go ahead, shade the feasible region and identify all redundant constraints.

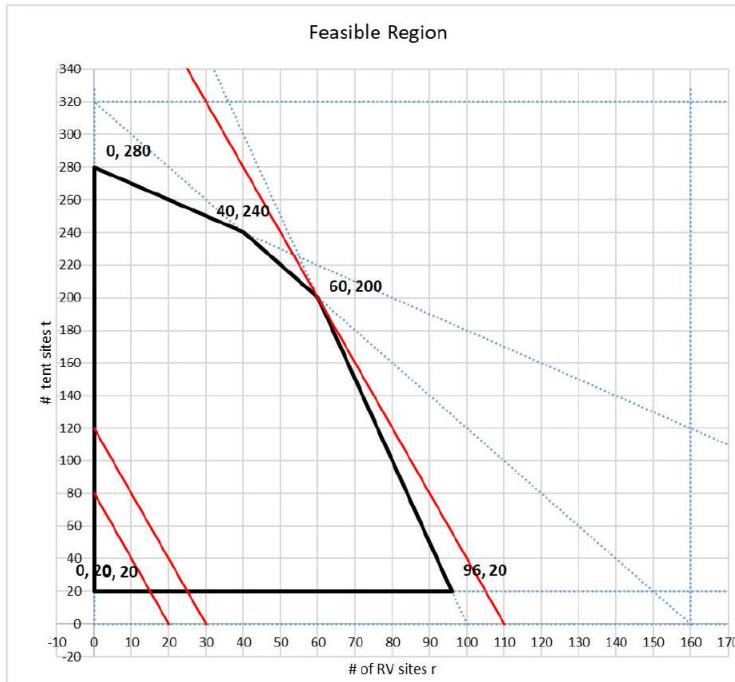


All the points in the feasible region are possible solutions. Your job is to pick (one of) the best solutions. Note that there may not be a single best solution but rather several optimal solutions.

We have not yet used the objective function $P(r,t) = 80r + 20t$. Because we do not have a value for $P(r,t)$, so we will draw $P(r,t)$ for a few random values of $P(r,t)$ to get an idea of what it looks like. For $P(r,t) = 1600, 2400, 3200$ we get the lines shown in the next picture. Note that they are all parallel, and that the lines corresponding to the larger value of $P(r,t)$ move to the top left. The direction of increase is just as we expected, to the top right.



As you move the line for the $P(r,t)$ to the right you increase your profit. But you also have to stay in the feasible region. Convince yourself that one of two cases will occur: either a unique optimal solution will be found at a corner point, or infinitely many optimal solutions returning the same maximum value for $P(r,t)$ will be found along a section of the boundary of the feasible region that includes two corner points. Therefore, we need to compute the corner points, i.e. the intersections of the constraints, and move $P(r,t)$ as far to the right as possible without leaving the feasible region.



From the picture above, you can see that the optimal solution will be at the intersection of the lines corresponding to constraints 1) and 5).

$$r/20 + t/40 \leq 8 \text{ and } 1000r + 200t \leq 100,000$$

which is at the point $r = 60, t = 200$. (Now would be a good time to review how to solve systems of equations....). This gives us a profit $P(60, 200) = 60 \cdot \$80 + 200 \cdot \$20 = \$8800$.

7.1.6. Stating the Solution

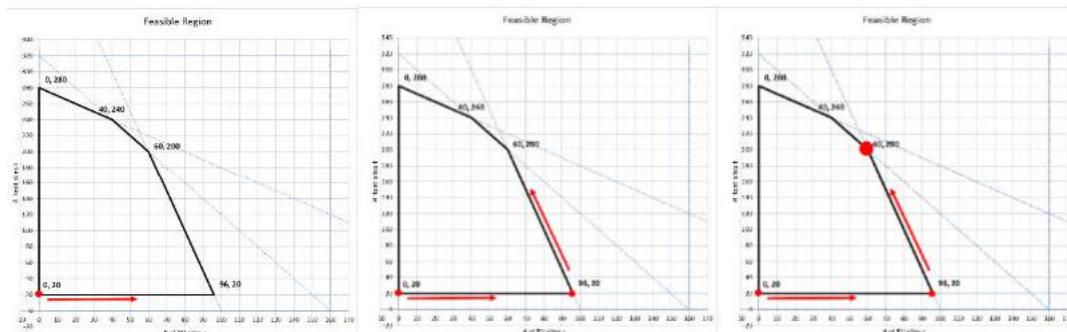
In OR, typically someone hires you to work your problem, and then expects you to give the answer in the context of the problem. You can't just say "the solution is at $r = 60, t = 200, P(r, t) = 8800$ ". Give the answer like this:

"To maximize potential profit, the campground should have 60 RV sites and 200 tent sites. In that case, the potential profit per day, assuming full occupancy, is \$8800. You are limited by the available land and budget available, not by available labor or any zoning rules. You will have to hire 65 hours of help per week (260 camping units at 15 minutes/week/unit)."

7.1.7. Refinements to the graphical solution

One "brute force" approach to the graphical solution method would be to compute all intersections of all constraints, check if that corner is in the feasible region, and then compute the objective function at those points. However, the number of intersections increases quadratically with the number of lines ($\frac{n(n+1)}{2}$ for n non-parallel lines), so this approach quickly gets out of hand. Instead, the idea is to start at an easy to find corner of the feasible region. Often, the origin works. From that point, check the adjacent feasible corner points and move to the "best". Continue until there is no more improvement. Here is what that would look like in our case:

We start at a simple corner. Usually, people use the origin, which is not on the feasible region here, so we start at $(0, 20)$. Adjacent corners are $(0, 280)$ and $(96, 20)$ with $P(0, 20) = 400$, $P(0, 280) = 5600$, and $P(96, 20) = 8080$. $(96, 20)$ is best, so we move there. Next, check the adjacent corner $(60, 200)$. It gives $P(60, 200) = 8800$ which is an improvement, so move there. Check the next adjacent corner $(40, 240)$. It gives $P(40, 240) = 8000$. This is worse, so $(60, 200)$ is the optimal solution.



There is of course a big problem with this approach. It relies on us having a graph of the feasible region and being able to see the adjacent feasible corner points. If you are dealing with a large number of constraints and variables, this is not possible. We therefore take the idea of looking at adjacent feasible corners and moving towards the one that gives the best value for the objective function, and translate it into an algebraic method.

SOLUTION APPROACH - ALGEBRAIC SIMPLEX METHOD As these are only three variables, we could still draw the feasible region, now a solid bound by planes. However, we need an approach that works for any number of variables. The key to this method is the fact that an optimal solution will occur at a corner point of the feasible region. While the n-dimensional proof is beyond the scope of this class, this fact should be intuitively clear in the 2-d and 3-d cases.

We will first demonstrate the algebraic method on the two-variable problem (RV and tent only) and then solve the full problem.

CORNER POINTS, INTERIOR CORNER POINTS, SLACK VARIABLES We introduce slack variables to turn the inequalities into equalities. Basically, a slack variable lets you know how close you are to maxing out the constraint. In the current case, we have:

Constraints	Augmented constraints
1) $4r + t \leq 320$	1) $2r + t + s_1 = 320$
2) $5r + t \leq 500$	2) $5r + t + s_2 = 500$
3) $r + t \leq 280$	3) $r + t + s_3 = 280$
4) $t \geq 20$	4) $t - s_4 = 20$
5) $r, t \geq 0$	5) $r, t, s_1, s_2, s_3, s_4 \geq 0$

A point is a corner point if it sits at the intersection of two or more constraints, i.e. if two or more slack variables are zero. A point is a feasible solution (i.e. inside the feasible region) if all slack variables are non-negative. A point is outside the feasible region if any slack variable is negative. We will from now on express each point as $(r, t, s_1, s_2, s_3, s_4)$. Given r and t , one computes the values of the slack variables from the augmented constraints.

Examples:	$r = 100, t = 20 \rightarrow (100, 20, 100, -20, 140, 0)$	outside the feasible region
	$r = 10, t = 30 \rightarrow (10, 30, 270, 420, 0, 0)$	corner outside fe
$r = 150, t = 20 \rightarrow$	$(150, 20, 0, -270, 110, 0)$	corner outside fe

INITIALIZING Because $(0,0)$ is not on the feasible region, we again start at $(r,t) = (0,20)$, which has the augmented form $(0,20,300,480,240,0)$. We are sitting on the intersection of the lines $r = 0$ and $t = 20$.

THE ADJUSTED OBJECTIVE FUNCTION Note: If the origin is a feasible corner point and you start at the origin, you can skip this step.

We are sitting on the intersection of the lines $r = 0$ and $t = 20$. To reach the next adjacent corner, we have to move along one of those lines, but which one? If we move along the line $r = 0$, we move away from the line $t = 20$, which is the same as saying we are increasing the corresponding slack, s_4 . If we move along the line $t = 20$, we increase r . We want to choose the direction of increase that gives us the fastest increase in P . The original objective function is $P(r,t) = 80r + 20t$, which does not let us see what happens if we increase s_4 . We have to rewrite $P(r,t)$ in terms of r and s_4 .

Using equation 4) we get $t = 20 + s_4$, and thus $P(r,s_4) = 80r + 20(20 + s_4) = 400 + 80r + 20s_4$

Determining which way to move

We want to choose the direction of increase that gives us the fastest increase in P . The objective function is $P(r,s_4) = 400 + 80r + 20s_4$. Because the variable r has the biggest coefficient, 80, an increase in r should give the best return.

Determining how far to move - the next corner

We will leave s_4 at its current value, 0, and increase r as much as possible without leaving the feasible region, i.e. without having t, s_1, s_2, s_3, s_4 become negative.

$$1. \quad 2r + t + s_1 = 320$$

$$s_1 = 320 - 20 - 2r \geq 0, \text{ so } r \leq 150$$

$$s_2 = 500 - 20 - 5r \geq 0, \text{ so } r \leq 96$$

$$2. \quad 5r + t + s_2 = 500$$

$$S_3 = 280 - 20 - r \geq 0, \text{ so } r \leq 240$$

$$3. \quad r + t + s_3 = 280$$

$$4. \quad t - S_4 = 20$$

$$t = 20$$

So, r can be increased to 96.

Augmented form of the next corner

Using $r = 96, s_4 = 0$, and substituting into the equations 1 – 4, we have the augmented point $(96, 20, 108, 0, 164, 0)$. Note that this is the same corner $(96, 20)$ we used above.

The adjusted objective function

We are now sitting on the intersection of the lines $5r + t + s_2 = 500$ and $t = 20$. To reach the next adjacent corner, we have to move along one of those lines, which means either increasing s_2 or s_4 . We have to rewrite $P(r, t)$ in terms of s_2 and s_4 .

Using equations 2 and 4, we find that $5r = 500 - t - s_2$ and $t = 20 + s_4$, which yields $5r = 480 - s_4 - s_2$. Substituting into P :

$$P(s_2, s_4) = 400 + 80r + 20s_4 = 400 + 16(480 - s_4 - s_2) = 20s_4 = 8080 + 4s_4 - 16s_2$$

Now we will repeat the above steps until the solution/objective function can no longer be improved upon.

DETERMINING WHICH WAY TO MOVE Because the S_4 has the only positive coefficient, this is the only direction that will yield an increase in P .

DETERMINING HOW FAR TO MOVE - THE NEXT CORNER We will leave s_2 at its current value, 0, and increase s_4 as much as possible without leaving the feasible region.

$$1. \quad 2r + t + s_1 = 320 \quad s_1 = 320 - t - 2r$$

$$2. \quad 5r + t + s_2 = 500 \\ s_2 = 500 - t - 5r$$

$$3. \quad r + t + s_3 = 280 \\ s_3 = 280 - t - r$$

$$4. \quad t - s_4 = 20 \quad t = 20 + s_4$$

We use that $s_2 = 0$ and $t = 20 + s_4$. This gives the set of equations:

$$1. \quad s_1 = 108 - 0.6s_4 \\ \geq 0, \text{ so } s_4 \leq 96 \\ \geq 0, \text{ so } S_4 \leq 480$$

$$2. \quad r = 96 - 0.2s_4 \\ \geq 0, \text{ so } S_4 \leq 205$$

$$3. \quad s_3 = 164 - 0.8s_4$$

$$4. \quad t = 20 + s_4$$

≥ 0 , so $s_4 \geq -20$ (as 20 is positive, this is true anyway) So S_4 can be increased up to 180.

AUGMENTED FORM OF THE NEXT CORNER Using $s_2 = 0, s_4 = 180$, and substituting into the equations 1–4, we have the augmented point $(60, 200, 0, 0, 20, 180)$. Note that this is the second corner $(60, 200)$ we used above.

THE ADJUSTED OBJECTIVE FUNCTION We again re-write the objective function, this time in terms of s_1 and s_2 : $P(s_1, s_2) = 8080 + 4s_4 - 16s_2 = 8080 + 4(180 - 1.6s_1) - 16s_2 = 8800 - 6.6s_1 - 16s_2$. Note that increasing either s_1 or s_2 will decrease the value of P , so we have reached the maximum. As s_1 and s_2 are non-negative, we can also see that the maximum for P occurs when s_1 and s_2 are zero, at $P = 8800$. Again, this is the same answer we arrived at earlier.

A nice side effect is that we can tell which constraints are holding us back, namely those associated with the zero slack variables s_1 and s_2 . s_1 corresponds to the space limitations, and s_2 to the budget restrictions.

Solving the full problem

We are now ready to look at the original problem. We will assume you went to the bank and got a loan for \$246,000 to supplement your original budget. Here are the constraints again:

1. Infrastructure takes up 20% of your site
2. You can have 20RV s per acre, or
3. You can have 40 tents per acre, or
4. You can have 10 yurts per acre
5. You have a budget of \$346,000. It costs you \$1000 to develop an RV site, \$200 for a tent site, and \$8000 for a yurt.
6. Maintenance for the bath houses etc. is 15 min/ week/camper unit, you can afford 70hrs/ week in maintenance help
7. Zoning ordinance requires you to have at least 20 tent sites

Functional Constraints	Simplified Constraints	Augmented Constraints
$r/20 + t/40 + y/10 \leq 8$	$2r + t + 4y \leq 320$	$2r + t + 4y + s_1 = 320$
$1000r + 200t + 8000y \leq 346,000$	$5r + t + 40y \leq 1730$	$5r + t + 40y + s_2 = 1750$
$(r + t + y)/4 \leq 70$	$r + t + y \leq 280$	$r + t + y + s_3 = 280$
$t \geq 20$	$t \geq 20$	$t - s_4 = 20$

NON-NEGATIVITY CONSTRAINTS: $r, t, y, s_1, s_2, s_3, s_4 \geq 0$

OBJECTIVE FUNCTION: Maximize $P(r, t, y) = 80r + 20t + 200y$

INITIALIZING Because $(0, 0, 0)$ is not on the feasible region, we start at $(r, t, y) = (0, 20, 0)$, which has the augmented form $(0, 20, 0, 300, 480, 240, 0)$. We are sitting on the intersection of the planes $r = 0, t = 20, y = 0$. The augmented form of this corner is $(0, 20, 0, 300, 1710, 260, 0)$

THE ADJUSTED OBJECTIVE FUNCTION Rewriting P as $P(r, y, s_4)$ gives:
 $P(P(r, y, s_4)) = 400 + 80r + 200y + 20s_4$

DETERMINING WHICH WAY TO MOVE Looking at the coefficients of r, y, s_4 in the objective function, we find that we should increase y and leave r and $s_4 = 0$

DETERMINING HOW FAR TO MOVE - THE NEXT CORNER Using $r = 0$ and $s_4 = 0$, the constraints become

$$\begin{aligned} t + 4y + s_1 &= 320 & 4y + s_1 &= 300 & s_1 &= 300 - 4y \geq 0 \rightarrow y \leq 75 \\ t + 40y + s_2 &= 1750 & 40y + s_2 &= 1730 & s_2 &= 1730 - 40y \geq 0 \rightarrow y \leq 42.75 \\ t + y + s_3 &= 280 & y + s_3 &= 260 & s_3 &= 260 - y \geq 0 \rightarrow y \leq 260 \end{aligned}$$

$$t = 20$$

So, y can be increased up to 42.75

AUGMENTED FORM OF THE NEXT CORNER With $r = 0$, $s_4 = 0$, and $y = 42.75$, we find the new augmented corner to be $(0, 20, 42.75, 129, 0, 217.25, 0)$.

THE ADJUSTED OBJECTIVE FUNCTION We need to rewrite the objective function in terms of r, s_2 , and s_4 :

$$P(r, s_2, s_4) = 8950 + 55r + 15s_4 - 5s_2$$

The solution is not optimal yet, (there are still positive coefficients in the objective function), so we keep going.

DETERMINING WHICH WAY TO MOVE We see that we should increase r and leave s_2 and $s_4 = 0$.

Determining how far to move - the next corner

With s_2 and $s_4 = 0$, we have

$$\begin{aligned} 2r + t + 4y + s_1 &= 320 & 2r + 4y + s_1 &= 300 & s_1 &= 129 - 1.5r \geq 0 \rightarrow r \leq 86 \\ 5r + t + 40y &= 1750 & 5r + 40y &= 1730 & 40y &= 1710 - 5r \\ r + t + y + s_3 &= 280 & r + y + s_3 &= 260 & s_3 &= 236.25 - 0.875r \geq 0 \rightarrow r \leq 342 \end{aligned}$$

$$t = 20$$

So, y can be increased up to 86

AUGMENTED FORM OF THE NEXT CORNER With $y = 86$, $s_2 = 2$, and $s_4 = 0$ we find the new augmented corner to be $(86, 20, 32, 0, 0, 142, 0)$

THE ADJUSTED OBJECTIVE FUNCTION We need to rewrite the objective function in terms of s_1 , s_2 , and s_4 :

$$P(s_1, s_2, s_4) = 13680 - 36\frac{2}{3}s_1 - 1\frac{1}{3}s_2 - 18s_4$$

Note that now all variables have negative coefficients, so we cannot increase the value of P past 13680. The first, second, and fourth constraints are maxed out; we are limited in our ability to increase the profit by space, money, and zoning restrictions.

Stating the Solution

To maximize potential profit, the campground should have 86 RV sites, 20 tent sites, and 32 yurts. This will take an initial investment of \$346,000. The potential profit per day, assuming full occupancy, is \$13,680. You are limited by the available land and budget available and the zoning law requiring 20 tent sites. You will have to hire 34.5 hours of help per week (138 camping units at 15 minutes/week/unit)."

7.1.8. Solution Approach - Using Excel

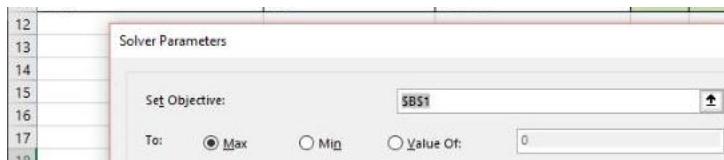
Now that we know how the solution method works, we can use Excel to do the work for us. First, we must set up the work sheet. One way that works well is shown on the next page. The fields highlighted in green are necessary, the others serve to explain and label what we are doing.

A	B	C	D	E	F	G
1 objective function	=B2*80+B3*20+B4*200					
2 # of RV sites, r	0					
3 # of tent sites, t	0					
4 # of yurts, y	0					
5						
6						
7 Original functional constraints	Simplified constraints	Constraints w/ slack vars				
8 $r/20+t/40+y/10 \leq 8$	$2r+t+4y=320$	$2r+t+4y+s_1=320$	$=2*B2+B3+4*B4$	\leq	320	
9 $1,000r+200t+8000y \leq 346,000$	$5r+t+40y=1730$	$5r+t+40y+s_2=1730$	$=5*B2+B3+40*B4$	\leq	1730	
10 $(r+t+y)/4 \leq 70$	$r+t+y=280$	$r+t+y+s_3=280$	$=B2+B3+B4$	\leq	280	
11 $t \geq 20$	$t=20$	$t-s_4=20$	$=B3$	\geq	20	
12						

Excel has a built-in Solver under the Data tab (if you don't see it, you have to add it in. Go to file/options/Add-ins/Manage Excel Add-ins/Solver Add-in). If you choose "show iteration results in the options tab, the solver will stop at each iteration and show you the corner/solution it has arrived found at that step. You will see that the solver goes through the same steps and corner points as we did when we worked the problem "by hand".

	A	B	C	D	E	F	G
1	objective function	0					
2	# of RV sites, r	0					
3	# of tent sites, t	0			s1	s2	s3
4	# of yurts, y	0			320	1730	280
5							20
6							
7	Original functional constraints	Simplified constraints	Constraints w/ slack vars				
8	$r/20+t/40+y/10 \leq 8$	$2r+t+4y=320$	$2r+t+4y+s1=320$	0	\leq	320	
9	$1,000r+200t+8000y \leq 346,000$	$5r+t+40y=1730$	$5r+t+40y+s2=1730$	0	\leq	1730	
10	$(r+t+y)/4 \leq 70$	$r+t+y=280$	$r+t+y+s3=280$	0	\leq	280	
11	$t \geq 20$	$t=20$	$t-s4=20$	0	\geq	20	
12							

AllMethods|GRGNonlinear|Evolutionery|



Constraint tolerance: 0.000001.

- Automatic scaling

What Automatic scaling

How Iteration results

Solving with integer Constraints

Solving with integer constraints

18	By Changing Variable Cells:	\$
19	SBS2:SBS4	
20	Subject to the Constraints:	SDS11 = SFs11
21		

Ignore integer constraints

By Changing Variable Cells:

- T 2

1

(2. Solving Limits

4



Subject to the Constraints:

Max Time (Seconds):

Iterations:

Evolutionary and Integer Constraints:

Evolutionary and integer

Local subproblems:

Local feasible Solutions:

Options



•

Make Unconstrained Variables Non-Negative

Select a Solving Method:

Options

Solving Method

Solving Method Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the

Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

Help

Solve

Close

8. Simplex Method

Definition 8.1: Standard Form

A linear program is in standard form if it is written as

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

Definition 8.2: Extreme Point

A point x in a convex set C is called an extreme point if it cannot be written as a strict convex combination of other points in C .

Theorem 8.3: Optimal Extreme Point - Bounded Case

Consider a linear optimization problem in standard form. Suppose that the feasible region is bounded and non-empty.

Then there exists an optimal solution at an extreme point of the feasible region.

Proof. [Proof Sketch]



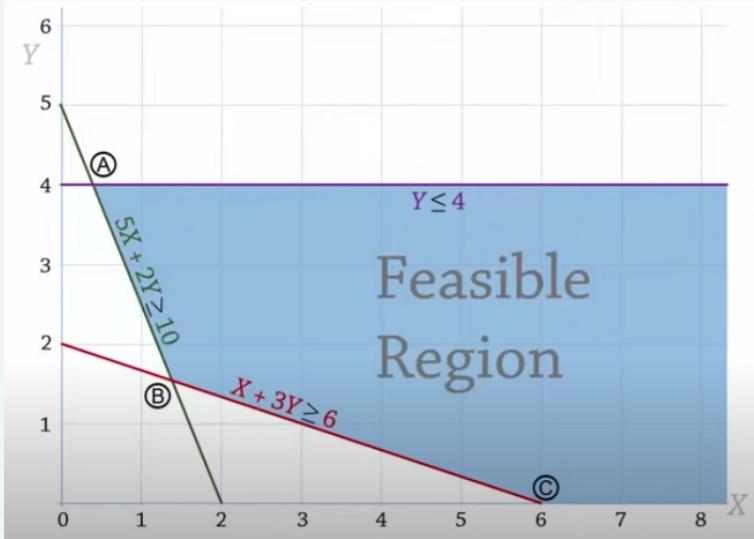
Definition 8.4: Basic solution

A basic solution to $Ax = b$ is obtained by setting $n - m$ variables equal to 0 and solving for the values of the remaining m variables. This assumes that the setting $n - m$ variables equal to 0 yields unique values for the remaining m variables or, equivalently, the columns of the remaining m variables are linearly independent.

Example 8.5

Consider the problem

$$\begin{aligned} \max \quad & Z = -5X - 7Y \\ \text{s.t.} \quad & X + 3Y \geq 6 \\ & 5X + 2Y \geq 10 \\ & Y \leq 4 \\ & X, Y \geq 0 \end{aligned}$$



We begin by converting this problem to standard form.

$$\begin{aligned} \max \quad & Z = -5X - 7Y + 0s_1 + 0s_2 + 0s_3 \\ \text{s.t.} \quad & X + 3Y - s_1 = 6 \\ & 5X + 2Y - s_2 = 10 \\ & Y + s_3 = 4 \\ & X, Y, s_1, s_2, s_3 \geq 0 \end{aligned}$$

Thus, we can write this problem in matrix form with

$$\max \begin{bmatrix} -5 \\ -7 \\ 0 \\ 0 \\ 0 \end{bmatrix}^\top \begin{bmatrix} X \\ Y \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \quad (8.1)$$

$$\begin{bmatrix} 1 & 3 & -1 & 0 & 0 \\ 5 & 2 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 10 \\ 4 \end{bmatrix} \quad (8.2)$$

$$(X, Y, s_1, s_2, s_3) \geq 0 \quad (8.3)$$

Definition 8.6: Basic feasible solution

Any basic solution in which all the variables are non-negative is a basic feasible solution.

Theorem 8.7: BFS iff extreme

A point in the feasible region of an LP is an extreme point if and only if it is a basic feasible solution to the LP.

To prove this theorem, we

Theorem 8.8: Representation

Consider an LP in standard form, having bfs b_1, \dots, b_k . Any point x in the LP's feasible region may be written in the form

$$x = d + \sum_{i=1}^k \sigma_i b_i$$

where d is 0 or a direction of unboundedness and $\sum_{i=1}^k \sigma_i = 1$ and $\sigma_i \geq 0$.

Theorem 8.9: Optimal bfs

If an LP has an optimal solution, then it has an optimal bfs.

Proof. Let x be an optimal solution. Then

$$x = d + \sum_{i=1}^k \sigma_i b_i$$

where d is 0 or a direction of unboundeness.

- If $c^\top d > 0$, the $x' = \lambda d + \sum_{i=1}^k \sigma_i b_i$ has bigger objective value for $|\lambda| > 1$, which is a contradiction since x was optimal.
- If $c^\top d < 0$, the $x'' = \sum_{i=1}^k \sigma_i b_i$ has a bigger objective value, which is a contradiction since x was optimal.

Thus, we conclude that $c^\top d = 0$.

Since

$$c^\top x \geq c^\top b_i$$

for all $i = 1, \dots, k$, we can conclude that

$$c^\top x = c^\top b_i$$

for all i such that $\sigma_i > 0$. Hence, there exists an optimal basic feasible solution. ♠

8.1 The Simplex Method

The Simplex Method Lab Objective: *The Simplex Method is a straightforward algorithm for finding optimal solutions to optimization problems with linear constraints and cost functions. Because of its simplicity and applicability, this algorithm has been named one of the most important algorithms invented within the last 100 years. In this lab we implement a standard Simplex solver for the primal problem.*

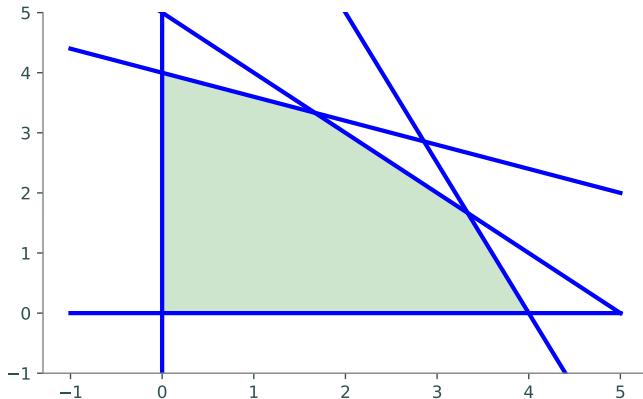
Standard Form

The Simplex Algorithm accepts a linear constrained optimization problem, also called a *linear program*, in the form given below:

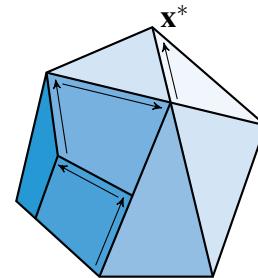
$$\begin{array}{ll} \text{minimize} & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{array}$$

Note that any linear program can be converted to standard form, so there is no loss of generality in restricting our attention to this particular formulation.

Such an optimization problem defines a region in space called the *feasible region*, the set of points satisfying the constraints. Because the constraints are all linear, the feasible region forms a geometric object called a *polytope*, having flat faces and edges (see Figure 8.1). The Simplex Algorithm jumps among the vertices of the feasible region searching for an optimal point. It does this by moving along the edges of the feasible region in such a way that the objective function is always increased after each move.



(a) The feasible region for a linear program with 2-dimensional constraints.



(b) The feasible region for a linear program with 3-dimensional constraints.

Figure 8.1: If an optimal point exists, it is one of the vertices of the polyhedron. The simplex algorithm searches for optimal points by moving between adjacent vertices in a direction that increases the value of the objective function until it finds an optimal vertex.

Implementing the Simplex Algorithm is straightforward, provided one carefully follows the procedure. We will break the algorithm into several small steps, and write a function to perform each one. To become familiar with the execution of the Simplex algorithm, it is helpful to work several examples by hand.

The Simplex Solver

Our program will be more lengthy than many other lab exercises and will consist of a collection of functions working together to produce a final result. It is important to clearly define the task of each function and how all the functions will work together. If this program is written haphazardly, it will be much longer and more difficult to read than it needs to be. We will walk you through the steps of implementing the Simplex Algorithm as a Python class.

For demonstration purposes, we will use the following linear program.

$$\begin{aligned}
 & \text{minimize} && -3x_0 - 2x_1 \\
 & \text{subject to} && x_0 - x_1 \leq 2 \\
 & && 3x_0 + x_1 \leq 5 \\
 & && 4x_0 + 3x_1 \leq 7 \\
 & && x_0, x_1 \geq 0.
 \end{aligned}$$

Accepting a Linear Program

Our first task is to determine if we can even use the Simplex algorithm. Assuming that the problem is presented to us in standard form, we need to check that the feasible region includes the origin. For now, we only check for feasibility at the origin. A more robust solver sets up the auxiliary problem and solves it to find a starting point if the origin is infeasible.

Problem 8.10: Check feasibility at the origin.

Write a class that accepts the arrays \mathbf{c} , A , and \mathbf{b} of a linear optimization problem in standard form. In the constructor, check that the system is feasible at the origin. That is, check that $A\mathbf{x} \leq \mathbf{b}$ when $\mathbf{x} = 0$. Raise a `ValueError` if the problem is not feasible at the origin.

Adding Slack Variables

The next step is to convert the inequality constraints $A\mathbf{x} \leq \mathbf{b}$ into equality constraints by introducing a slack variable for each constraint equation. If the constraint matrix A is an $m \times n$ matrix, then there are m slack variables, one for each row of A . Grouping all of the slack variables into a vector \mathbf{w} of length m , the constraints now take the form $A\mathbf{x} + \mathbf{w} = \mathbf{b}$. In our example, we have

$$\mathbf{w} = \begin{bmatrix} x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

When adding slack variables, it is useful to represent all of your variables, both the original primal variables and the additional slack variables, in a convenient manner. One effective way is to refer to a variable by its subscript. For example, we can use the integers 0 through $n - 1$ to refer to the original (non-slack) variables x_0 through x_{n-1} , and we can use the integers n through $n + m - 1$ to track the slack variables (where the slack variable corresponding to the i th row of the constraint matrix is represented by the index $n + i - 1$).

We also need some way to track which variables are *independent* (non-zero) and which variables are *dependent* (those that have value 0). This can be done using the objective function. At anytime during the optimization process, the non-zero variables in the objective function are *independent* and all other variables are *dependent*.

Creating a Dictionary

After we have determined that our program is feasible, we need to create the *dictionary* (sometimes called the *tableau*), a matrix to track the state of the algorithm.

There are many different ways to build your dictionary. One way is to mimic the dictionary that is often used when performing the Simplex Algorithm by hand. To do this we will set the corresponding dependent variable equations to 0. For example, if x_5 were a dependent variable we would expect to see a -1 in the column that represents x_5 . Define

$$\bar{A} = [A \quad I_m],$$

where I_m is the $m \times m$ identity matrix we will use to represent our slack variables, and define

$$\bar{\mathbf{c}} = \begin{bmatrix} \mathbf{c} \\ 0 \end{bmatrix}.$$

That is, $\bar{\mathbf{c}} \in \mathbb{R}^{n+m}$ such that the first n entries are \mathbf{c} and the final m entries are zeros. Then the initial dictionary has the form

$$D = \begin{bmatrix} 0 & \bar{\mathbf{c}}^T \\ \mathbf{b} & -\bar{A} \end{bmatrix} \tag{8.1}$$

The columns of the dictionary correspond to each of the variables (both primal and slack), and the rows of the dictionary correspond to the dependent variables.

For our example the initial dictionary is

$$D = \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix}.$$

The advantage of using this kind of dictionary is that it is easy to check the progress of your algorithm by hand.

Problem 8.11: Initialize the dictionary.

dd a method to your Simplex solver that takes in arrays c , A , and b to create the initial dictionary (D) as a NumPy array.

8.1.1. Pivoting

Pivoting is the mechanism that really makes Simplex useful. Pivoting refers to the act of swapping dependent and independent variables, and transforming the dictionary appropriately. This has the effect of moving from one vertex of the feasible polytope to another vertex in a way that increases the value of the objective function. Depending on how you store your variables, you may need to modify a few different parts of your solver to reflect this swapping.

When initiating a pivot, you need to determine which variables will be swapped. In the dictionary representation, you first find a specific element on which to pivot, and the row and column that contain the pivot element correspond to the variables that need to be swapped. Row operations are then performed on the dictionary so that the pivot column becomes a negative elementary vector.

Let's break it down, starting with the pivot selection. We need to use some care when choosing the pivot element. To find the pivot column, search from left to right along the top row of the dictionary (ignoring the first column), and stop once you encounter the first negative value. The index corresponding to this column will be designated the *entering index*, since after the full pivot operation, it will enter the basis and become a dependent variable.

Using our initial dictionary D in the example, we stop at the second column:

$$D = \left[\begin{array}{c|ccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right]$$

We now know that our pivot element will be found in the second column. The entering index is thus 1.

Next, we select the pivot element from among the negative entries in the pivot column (ignoring the entry in the first row). *If all entries in the pivot column are non-negative, the problem is unbounded and has no solution.* In this case, the algorithm should terminate. Otherwise, assuming our pivot column is the j th column of the dictionary and that the negative entries of this column are $D_{i_1,j}, D_{i_2,j}, \dots, D_{i_k,j}$, we calculate the ratios

$$\frac{-D_{i_1,0}}{D_{i_1,j}}, \frac{-D_{i_2,0}}{D_{i_2,j}}, \dots, \frac{-D_{i_k,0}}{D_{i_k,j}},$$

and we choose our pivot element to be one that minimizes this ratio. If multiple entries minimize the ratio, then we utilize *Bland's Rule*, which instructs us to choose the entry in the row corresponding to the smallest index (obeying this rule is important, as it prevents the possibility of the algorithm cycling back on itself infinitely). The index corresponding to the pivot row is designated as the *leaving index*, since after the full pivot operation, it will leave the basis and become an independent variable.

In our example, we see that all entries in the pivot column (ignoring the entry in the first row, of course) are negative, and hence they are all potential choices for the pivot element. We then calculate the ratios, and obtain

$$\frac{-2}{-1} = 2, \quad \frac{-5}{-3} = 1.66\dots, \quad \frac{-7}{-4} = 1.75.$$

We see that the entry in the third row minimizes these ratios. Hence, the element in the second column (index 1), third row (index 2) is our designated pivot element.

$$D = \left[\begin{array}{cccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & \boxed{-3} & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right]$$

Definition 8.12: Bland's Rule

Choose the independent variable with the smallest index that has a negative coefficient in the objective function as the leaving variable. Choose the dependent variable with the smallest index among all the binding dependent variables.

Bland's Rule is important in avoiding cycles when performing pivots. This rule guarantees that a feasible Simplex problem will terminate in a finite number of pivots.

Finally, we perform row operations on our dictionary in the following way: divide the pivot row by the negative value of the pivot entry. Then use the pivot row to zero out all entries in the pivot column above and below the pivot entry. In our example, we first divide the pivot row by -3, and then zero out the two entries above the pivot element and the single entry below it:

$$\begin{aligned} \left[\begin{array}{cccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] &\rightarrow \left[\begin{array}{cccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] \rightarrow \\ \left[\begin{array}{cccccc} -5 & 0 & -1 & 0 & 1 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] &\rightarrow \left[\begin{array}{cccccc} -5 & 0 & -1 & 0 & 1 & 0 \\ 1/3 & 0 & -4/3 & 1 & -1/3 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] \rightarrow \\ \left[\begin{array}{cccccc} -5 & 0 & -1 & 0 & 1 & 0 \\ 1/3 & 0 & 4/3 & -1 & 1/3 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 1/3 & 0 & -5/3 & 0 & 4/3 & -1 \end{array} \right]. \end{aligned}$$

The result of these row operations is our updated dictionary, and the pivot operation is complete.

Problem 8.13: Pivoting

Add a method to your solver that checks for unboundedness and performs a single pivot operation from start to completion. If the problem is unbounded, raise a `ValueError`.

8.1.2. Termination and Reading the Dictionary

Up to this point, our algorithm accepts a linear program, adds slack variables, and creates the initial dictionary. After carrying out these initial steps, it then performs the pivoting operation iteratively until the optimal point is found. But how do we determine when the optimal point is found? The answer is to look at the top row of the dictionary, which represents the objective function. More specifically, before each pivoting operation, check whether all of the entries in the top row of the dictionary (ignoring the entry in the first column) are nonnegative. If this is the case, then we have found an optimal solution, and so we terminate the algorithm.

The final step is to report the solution. The ending state of the dictionary and index list tells us everything we need to know. The minimal value attained by the objective function is found in the upper leftmost entry of the dictionary. The dependent variables all have the value 0 in the objective function or first row of our dictionary array. The independent variables have values given by the first column of the dictionary. Specifically, the independent variable whose index is located at the i th entry of the index list has the value $T_{i+1,0}$.

In our example, suppose that our algorithm terminates with the dictionary and index list in the following state:

$$D = \begin{bmatrix} -5.2 & 0 & 0 & 0 & 0.2 & 0.6 \\ 0.6 & 0 & 0 & -1 & 1.4 & -0.8 \\ 1.6 & -1 & 0 & 0 & -0.6 & 0.2 \\ 0.2 & 0 & -1 & 0 & 0.8 & -0.6 \end{bmatrix}$$

Then the minimal value of the objective function is -5.2 . The independent variables have indices 4, 5 and have the value 0. The dependent variables have indices 3, 1, and 2, and have values .6, 1.6, and .2, respectively. In the notation of the original problem statement, the solution is given by

$$x_0 = 1.6$$

$$x_1 = .2.$$

Problem 8.14: SimplexSolver.solve()

Write an additional method in your solver called `solve()` that obtains the optimal solution, then returns the minimal value, the dependent variables, and the independent variables. The dependent and independent variables should be represented as two dictionaries that map the index of the variable to its corresponding value.

For our example, we would return the tuple

(-5.2 , {0: 1.6, 1: .2, 2: .6}, {3: 0, 4: 0}).

At this point, you should have a Simplex solver that is ready to use. The following code demonstrates how your solver is expected to behave:

```
>>> import SimplexSolver

# Initialize objective function and constraints.
>>> c = np.array([-3., -2.])
>>> b = np.array([2., 5, 7])
>>> A = np.array([[1., -1], [3, 1], [4, 3]])

# Instantiate the simplex solver, then solve the problem.
>>> solver = SimplexSolver(c, A, b)
>>> sol = solver.solve()
>>> print(sol)
(-5.2,
 {0: 1.6, 1: 0.2, 2: 0.6},
 {3: 0, 4: 0})
```

If the linear program were infeasible at the origin or unbounded, we would expect the solver to alert the user by raising an error.

Note that this simplex solver is *not* fully operational. It can't handle the case of infeasibility at the origin. This can be fixed by adding methods to your class that solve the *auxiliary problem*, that of finding an initial feasible dictionary when the problem is not feasible at the origin. Solving the auxiliary problem involves pivoting operations identical to those you have already implemented, so adding this functionality is not overly difficult.

8.1.3. Exercises

EXERCISE 1.0 (LEARN L_TE_X) Learn to use L_TE_X for writing all of your homework solutions. Personally, I use MiKTEX, which is an implementation of ETEX for Windows. Specifically, within MiKTEX I am using pdfeTEX (it only matters for certain things like including graphics and also pdf into a document). I find it convenient to use the editor WinEdt, which is very LATEX friendly. A good book on ETTX is

In A.1 there is a template to get started. Also, there are plenty of tutorials and beginner's guides on the web.

EXERCISE 1.1 (CONVERT TO STANDARD FORM) Give an original example (i.e., with actual numbers) to demonstrate that you know how to transform a general linear-optimization problem to one in standard form.

EXERCISE 1.2 (WEAK DUALITY EXAMPLE) Give an original example to demonstrate the Weak Duality Theorem.

EXERCISE 1.3 (CONVERT TO \leq FORM) Describe a general recipe for transforming an arbitrary linear-optimization problem into one in which all of the linear constraints are of \leq type.

EXERCISE 1.4 ($m + 1$ INEQUALITIES) Prove that the system of m equations in n variables $Ax = b$ is equivalent to the system $Ax \leq b$ augmented by only one additional linear inequality - that is, a total of only $m + 1$ inequalities.

EXERCISE 1.5 (WEAK DUALITY FOR ANOTHER FORM) Give and prove a Weak Duality Theorem for

$$\begin{aligned} \max \quad & c'x \\ \text{subject to } & Ax \leq b; \\ & x \geq 0. \end{aligned}$$

HINT: Convert (P') to a standard-form problem, and then apply the ordinary Weak Duality Theorem for standard-form problems.

EXERCISE 1.6 (WEAK DUALITY FOR A COMPLICATED FORM) Give and prove a Weak Duality Theorem for

$$\begin{aligned} \min \quad & c'x + f'w \\ \text{subject to } & Ax + Bw \leq b; \\ & Dx = g; \\ & x \geq 0 \quad w \leq 0 \end{aligned}$$

HINT: Convert (P') to a standard-form problem, and then apply the ordinary Weak Duality Theorem for standard-form problems.

EXERCISE 1.7 (WEAK DUALITY FOR A COMPLICATED FORM - WITH MATLAB) The MATLAB code below makes and solves an instance of (P') from Exercise 1.6. Study the code to see how it works. Now, extend the code to solve the dual of (P') . Also, after converting (P') to standard form (as indicated in the HINT for Exercise 1.6), use MATLAB to solve that problem and its dual. Make sure that you get the same optimal value for all of these problems.

8.1.4. 2.5 Exercises

EXERCISE 2.1 (DUAL IN AMPL) Without changing the file `production.dat`, use AMPL to solve the dual of the Production Problem example, as described in Section 2.1. You will need to modify `production.mod` and `production.run`.

EXERCISE 2.2 (SPARSE SOLUTION FOR LINEAR EQUATIONS WITH AMPL) In some application areas, it is interesting to find a "sparse solution" - that is, one with few non-zeros - to a system of equations $Ax = b$. It is well known that a 1-norm minimizing solution is a good heuristic for finding a sparse solution. Using AMPL, try this idea out on several large examples, and report on your results.

HINT: To get an interesting example, try generating a random $m \times n$ matrix A of zeros and ones, perhaps $m = 50$ equations and $n = 500$ variables, maybe with probability $1/2$ of an entry being equal to one. Then choose maybe $m/2$ columns from A and add them up to get b . In this way, you will know that there is a solution with only $m/2$ non-zeros (which is already pretty sparse). Your 1-norm minimizing solution might in fact recover this solution (\odot) , or it may be sparser $(\odot\odot)$, or perhaps less sparse (\odot) .

EXERCISE 2.3 (BLOODY AMPL) A transportation problem is a special kind of (single-commodity min-cost) networkflow problem. There are certain nodes v called supply nodes which have net supply $b_v > 0$. The other nodes v are called demand nodes, and they have net supply $b_v < 0$. There are no nodes with $b_v = 0$, and all arcs point from supply nodes to demand nodes.

A simplified example is for matching available supply and demand of blood, in types A, B, AB and O . Suppose that we have s_v units of blood available, in types $v \in \{A, B, AB, O\}$. Also, we have requirements d_v by patients of different types $v \in \{A, B, AB, O\}$. It is very important to understand that a patient of a certain type can accept blood not just from their own type. Do some research to find out the compatible blood types for a patient; don't make a mistake - lives depend on this! In this spirit, if your model misallocates any blood in an incompatible fashion, you will receive a grade of F on this problem.

Describe a linear-optimization problem that satisfies all of the patient demand with compatible blood. You will find that type O is the most versatile blood, then both A and B , followed by AB . Factor in this point when you formulate your objective function, with the idea of having the left-over supply of blood being as versatile as possible.

Using AMPL, set up and solve an example of a blood-distribution problem.

EXERCISE 2.4 (MIX IT UP) "I might sing a gospel song in Arabic or do something in Hebrew. I want to mix it up and do it differently than one might imagine." - Stevie Wonder

We are given a set of ingredients $1, 2, \dots, m$ with availabilities b_i and per unit costs c_i . We are given a set of products $j, 2, \dots, m$ with minimum production requirements d_j and per unit revenues e_j . It is required that product j have at least a fraction of l_{ij} of ingredient i and at most a fraction of u_{ij} of ingredient i . The goal is to devise a plan to maximize net profit.

Formulate, mathematically, as a linear-optimization problem. Then, model with AMPL, make up some data, try some computations, and report on your results. Exercise 2.5 (Task scheduling)

We are given a set of tasks, numbered $1, 2, \dots, n$ that should be completed in the minimum amount of time. For convenience, task 0 is a "start task" and task $n + 1$ is an "end task". Each task, except for the start and end task, has a known duration d_i . For convenience, let $d_0 := 0$. There are precedences between tasks. Specifically, Ψ_i is the set of tasks that must be completed before task i can be started. Let $t_0 := 0$, and for all other tasks i , let t_i be a decision variable representing its start time.

Formulate the problem, mathematically, as a linear-optimization problem. The objective should be to minimize the start time t_{n+1} of the end task. Then, model the problem with AMPL, make up some data, try some computations, and report on your results.

EXERCISE 2.6 (INVESTING WISELY) Almost certainly, Albert Einstein did not say that "compound interest is the most powerful force in the universe."

A company wants to maximize their cash holdings after T time periods. They have an external inflow of p_t dollars at the start of time period t , for $t = 1, 2, \dots, T$. At the start of each time period, available cash can be allocated to any of K different investment vehicles (in any available non-negative amounts). Money allocated to investment vehicle k at the start of period t must be held in that investment k for all remaining time periods, and it generates income $v_{t,t}^k, v_{t,t+1}^k, \dots, v_{t,T}^k$, per dollar invested. It should be assumed that money obtained from cashing out the investment at the end of the planning horizon (that is, at the end of period T) is part of $v_{t,T}^k$. Note that at the start of time period t , the cash available is the external inflow of p_t , plus cash accumulated from all investment vehicles in prior periods that was not reinvested. Finally, assume that cash held over for one time period earns interest of q percent.

Formulate the problem, mathematically, as a linear-optimization problem. Then, model the problem with AMPL, make up some data, try some computations, and report on your results.

8.2 Finding Feasible Basis

Finding an Initial BFS When a basic feasible solution is not apparent, we can produce one using *artificial variables*. This *artificial* basis is undesirable from the perspective of the original problem, we do not want the artificial variables in our solution, so we penalize them in the objective function, and allow the simplex algorithm to drive them to zero (if possible) and out of the basis. There are two such methods, the **Big M method** and the **Two-phase method**, which we illustrate below:

Solve the following LP using the Big M Method and the simplex algorithm:

$$\begin{aligned} \max z &= 9x_1 + 6x_2 \\ \text{s.t. } &3x_1 + 3x_2 \leq 9 \\ &2x_1 - 2x_2 \geq 3 \\ &2x_1 + 2x_2 \geq 4 \\ &x_1, x_2 \geq 0. \end{aligned}$$

Here is the LP is transformed into standard form by using slack variables x_3 , x_4 , and x_5 , with the required artificial variables x_6 and x_7 , which allow us to easily find an initial basic feasible solution (to the artificial problem).

$$\begin{aligned} \max z_a &= 9x_1 + 6x_2 - Mx_6 - Mx_7 \\ \text{s.t. } &3x_1 + 3x_2 + x_3 = 9 \\ &2x_1 - 2x_2 - x_4 + x_6 = 3 \\ &2x_1 + 2x_2 - x_5 + x_7 = 4 \\ &x_i \geq 0, \quad i = 1, \dots, 7. \end{aligned}$$

z	x_1	x_2	x_3	x_4	x_5	x_6	x_7	RHS	ratio
1	-9	-6	0	0	0	M	M	0	
0	3	3	1	0	0	0	0	9	
0	2	-2	0	-1	0	1	0	3	
0	2	2	0	0	-1	0	1	4	

This tableau is not in the correct form, it does not represent a basis, the columns for the artificial variables need to be adjusted.

z	x_1	x_2	x_3	x_4	x_5	x_6	x_7	RHS	ratio
1	-9 - 4M	-6	0	M	M	0	0	-7M	
0	3	3	1	0	0	0	0	9	3
0	2	-2	0	-1	0	1	0	3	3/2
0	2	2	0	0	-1	0	1	4	2

The current solution is not optimal, so x_1 enters the basis, and by the ratio test, x_6 (an artificial variable) leaves the basis.

z	x_1	x_2	x_3	x_4	x_5	x_6	x_7	RHS	ratio
1	0	-15 - 4M	0	-9/2 - M	M	9/2 + 2M	0	27/2 - M	
0	0	6	1	3/2	0	-3/2	0	3/2	3/4
0	1	-1	0	-1/2	0	1/2	0	3/2	-
0	0	4	0	1	-1	-1	1	1	1/4

The current solution is not optimal, so x_2 enters the basis, and by the ratio test, x_7 (an artificial variable) leaves the basis.

z	x_1	x_2	x_3	x_4	x_5	x_6	x_7	RHS	ratio
1	0	0	0	-3/4	-15/4	-	-	17 1/4	-
0	0	0	1	0	3/2	0	-3/2	3	-
0	1	0	0	-1/4	-1/4	1/2	1/4	7/4	-
0	0	1	0	1/4	-1/4	-1/4	1/4	1/4	1

The current solution is not optimal, so x_4 enters the basis, and by the ratio test, x_2 leaves the basis.

z	x_1	x_2	x_3	x_4	x_5	x_6	x_7	RHS	ratio
1	0	3	0	0	-9/2	-	-	18	-
0	0	0	1	0	3/2	0	-3/2	3	-
0	1	1	0	0	-1/2	0	1/2	2	-
0	0	4	0	1	-1	-1	1	1	1

The current solution is not optimal, so x_5 enters the basis, and by the ratio test, x_3 leaves the basis.

z	x_1	x_2	x_3	x_4	x_5	x_6	x_7	RHS	ratio
1	0	3	3	0	0	-	-	27	-
0	0	0	2/3	0	1	0	-1	2	-
0	1	1	1/3	0	0	0	0	3	-
0	0	4	2/3	1	0	-1	0	3	-

The current solution is optimal!

Solve the following LP using the Two-phase Method and Simplex Algorithm.

$$\begin{aligned} \max z &= 2x_1 + 3x_2 \\ \text{s.t. } 3x_1 + 3x_2 &\geq 6 \\ 2x_1 - 2x_2 &\leq 2 \\ -3x_1 + 3x_2 &\leq 6 \\ x_1, x_2 &\geq 0. \end{aligned}$$

Here is first phase LP (in standard form), where x_3 , x_4 , and x_5 are slack variables, and x_6 is an artificial variable.

$$\begin{aligned} \min z_a &= x_6 \\ \text{s.t. } 3x_1 + 3x_2 - x_3 + x_6 &= 6 \\ 2x_1 - 2x_2 + x_4 &= 2 \\ -3x_1 + 3x_2 + x_5 &= 6 \\ x_i &\geq 0, \quad i = 1, \dots, 6. \end{aligned}$$

Next, we put the LP into a tableau, which, still is not in the right form for our basic variables (x_6 , x_4 , and x_5).

z	x_1	x_2	x_3	x_4	x_5	x_6	RHS	ratio
1	0	0	0	0	0	-1	0	
0	3	3	-1	0	0	1	6	
0	2	-2	0	1	0	0	2	
0	-3	3	0	0	1	0	6	

To remedy this, we use row operation to modify the row 0 coefficients, yielding the following:

z	x_1	x_2	x_3	x_4	x_5	x_6	RHS	ratio
1	3	3	-1	0	0	0	6	
0	3	3	-1	0	0	1	6	2
0	2	-2	0	1	0	0	2	-
0	-3	3	0	0	1	0	6	2

The current solution is not optimal, either x_1 or x_2 can enter the basis, let's choose x_2 . Then by the ratio test, either x_6 (an artificial variable) or x_5 (a slack variable) can leave the basis. Let's choose x_6 .

z	x_1	x_2	x_3	x_4	x_5	x_6	RHS	ratio
1	0	0	0	0	0	-1	0	
0	1	1	-1/3	0	0	1/3	2	
0	4	0	-2/3	1	0	2/3	6	
0	-6	0	1	0	1	-1	0	

The current solution is optimal, so we end the first phase with a basic feasible solution to the original problem, with x_2 , x_4 , and x_5 as the basic variables. Now we provide a new row zero that corresponds to the original problem.

z	x_1	x_2	x_3	x_4	x_5	x_6	RHS	ratio
1	1	0	-1	0	0	0	6	
0	1	1	-1/3	0	0	1/3	2	
0	4	0	-2/3	1	0	2/3	6	
0	-6	0	1	0	1	-1	0	

z	x_1	x_2	x_3	x_4	x_5	x_6	RHS	ratio
1	-5	0	0	0	1	-1	6	
0	-1	1	0	0	1/3	0	2	
0	0	0	0	1	2/3	0	6	
0	-6	0	1	0	1	-1	0	

From this tableau we can see that the LP is unbounded and an extreme point is $[0, 2, 0, 6, 0]$ and an extreme direction is $[1, 1, 6, 0, 0]$.

Degeneracy and the Simplex Algorithm

Degeneracy must be considered in the simplex algorithm, as it causes some trouble. For instance, it might mislead us into thinking there are multiple optimal solutions, or provide faulty insight. Further, the algorithm as described can *cycle*, that is, remain on a degenerate extreme point repeatedly cycling through a subset of bases that represent that point, never leaving.

min	z	x_1	x_2	x_3	x_4	x_5	x_6	x_7	rhs
	1	0	0	0	3/4	-20	1/2	-6	0
	0	1	0	0	1/4	-8	-1	9	0
	0	0	1	0	1/2	-12	-1/2	3	0
	0	0	0	1	0	0	1	0	1

Solve the following LP using the Simplex Algorithm:

$$\begin{aligned} \max \quad & z = 40x_1 + 30x_2 \\ \text{s.t.} \quad & 6x_1 + 4x_2 \leq 40 \\ & 4x_1 + 2x_2 \leq 20 \\ & x_1, x_2 \geq 0. \end{aligned}$$

By adding slack variables, we have the following tableau. Luckily, this tableau represents a basis, where

z	x_1	x_2	s_1	s_2	RHS
1	-40	-30	0	0	0
0	6	4	1	0	40
0	4	2	0	1	20

$BV=\{s_1, s_2\}$, but by inspecting the row 0 (objective function row) coefficients, we can see that this is not optimal. By Dantzig's Rule, we enter x_1 into the basis, and by the ratio test we see that s_2 leaves the basis. By performing elementary row operations, we obtain the following tableau for the new basis $BV=\{s_1, x_1\}$.

z	x_1	x_2	s_1	s_2	RHS
1	0	-10	0	10	200
0	0	1	1	-3/2	10
0	1	1/2	0	1/4	5

This tableau is not optimal, entering x_2 into the basis can improve the objective function value. The basic variables s_1 and x_1 tie in the ration test. If we have x_1 leave the basis, we get the following tableau ($BV=\{s_1, x_2\}$).

z	x_1	x_2	s_1	s_2	RHS
1	20	0	0	15	300
0	-2	0	1	-2	0
0	2	1	0	1/2	10

This is an optimal tableau, with an objective function value of 300, If instead of x_1 leaving the basis, suppose s_1 left, this would lead to the following tableau ($BV=\{x_2, x_1\}$).

z	x_1	x_2	s_1	s_2	RHS
1	0	0	10	-5	300
0	0	1	1	-3/2	10
0	1	0	-1/2	1	0

This tableau does not look optimal, yet the objective function value is the same as the optimal solution's. This occurs because the optimal extreme point is a degenerate.

9. Duality

Before I prove the stronger duality theorem, let me first provide some intuition about where this duality thing comes from in the first place.⁶ Consider the following linear programming problem:

$$\begin{aligned} & \text{maximize } 4x_1 + x_2 + 3x_3 \\ \text{subject to } & x_1 + 4x_2 \leq 2 \\ & 3x_1 - x_2 + x_3 \leq 4 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Let σ^* denote the optimum objective value for this LP. The feasible solution $x = (1, 0, 0)$ gives us a lower bound $\sigma^* \geq 4$. A different feasible solution $x = (0, 0, 3)$ gives us a better lower bound $\sigma^* \geq 9$. We could play this game all day, finding different feasible solutions and getting ever larger lower bounds. How do we know when we're done? Is there a way to prove an upper bound on σ^* ?

In fact, there is. Let's multiply each of the constraints in our LP by a new non-negative scalar value y_i :

$$\begin{aligned} & \text{maximize } 4x_1 + x_2 + 3x_3 \\ \text{subject to } & y_1(x_1 + 4x_2) \leq 2y_1 \\ & y_2(3x_1 - x_2 + x_3) \leq 4y_2 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Because each y_i is non-negative, we do not reverse any of the inequalities. Any feasible solution (x_1, x_2, x_3) must satisfy both of these inequalities, so it must also satisfy their sum:

$$(y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \leq 2y_1 + 4y_2.$$

Now suppose that each y_i is larger than the i th coefficient of the objective function:

$$y_1 + 3y_2 \geq 4, \quad 4y_1 - y_2 \geq 1, \quad y_2 \geq 3.$$

This assumption lets us derive an upper bound on the objective value of any feasible solution:

$$4x_1 + x_2 + 3x_3 \leq (y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \leq 2y_1 + 4y_2.$$

In particular, by plugging in the optimal solution (x_1^*, x_2^*, x_3^*) for the original LP, we obtain the following upper bound on σ^* :

$$\sigma^* = 4x_1^* + x_2^* + 3x_3^* \leq 2y_1 + 4y_2.$$

Now it's natural to ask how tight we can make this upper bound. How small can we make the expression $2y_1 + 4y_2$ without violating any of the inequalities we used to prove the upper bound? This is just another

linear programming problem.

$$\begin{array}{ll} \text{minimize} & 2y_1 + 4y_2 \\ \text{subject to} & y_1 + 3y_2 \geq 4 \\ & 4y_1 - y_2 \geq 1 \\ & y_2 \geq 3 \\ & y_1, y_2 \geq 0 \end{array}$$

"This example is taken from Robert Vanderbei's excellent textbook Linear Programming: Foundations and Extensions [Springer, 2001], but the idea appears earlier in Jens Clausen's 1997 paper 'Teaching Duality in Linear Programming: The Multiplier Approach'.

<https://www.cs.purdue.edu/homes/egrigore/580FT15/26-lp-jefferickson.pdf>

In which we introduce the theory of duality in linear programming.

9.1 The Dual of Linear Program

Suppose that we have the following linear program in maximization standard form:

$$\begin{array}{ll} \text{maximize} & x_1 + 2x_2 + x_3 + x_4 \\ \text{subject to} & x_1 + 2x_2 + x_3 \leq 2 \\ & x_2 + x_4 \leq 1 \\ & x_1 + 2x_3 \leq 1 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_3 \geq 0 \end{array}$$

and that an LP-solver has found for us the solution $x_1 := 1, x_2 := \frac{1}{2}, x_3 := 0, x_4 := \frac{1}{2}$ of cost 2.5. How can we convince ourselves, or another user, that the solution is indeed optimal, without having to trace the steps of the computation of the algorithm?

Observe that if we have two valid inequalities

$$a \leq b \text{ and } c \leq d$$

then we can deduce that the inequality

$$a + c \leq b + d$$

(derived by "summing the left hand sides and the right hand sides" of our original inequalities) is also true. In fact, we can also scale the inequalities by a positive multiplicative factor before adding them up, so for every non-negative values $y_1, y_2 \geq 0$ we also have

$$y_1a + y_2c \leq y_1b + y_2d$$

Going back to our linear program (1), we see that if we scale the first inequality by $\frac{1}{2}$, add the second inequality, and then add the third inequality scaled by $\frac{1}{2}$, we get that, for every (x_1, x_2, x_3, x_4) that is feasible for (1),

$$x_1 + 2x_2 + 1.5x_3 + x_4 \leq 2.5$$

And so, for every feasible (x_1, x_2, x_3, x_4) , its cost is

$$x_1 + 2x_2 + x_3 + x_4 \leq x_1 + 2x_2 + 1.5x_3 + x_4 \leq 2.5$$

meaning that a solution of cost 2.5 is indeed optimal.

In general, how do we find a good choice of scaling factors for the inequalities, and what kind of upper bounds can we prove to the optimum?

Suppose that we have a maximization linear program in standard form.

$$\begin{aligned} & \text{maximize} && c_1x_1 + \dots + c_nx_n \\ & \text{subject to} && \\ & && a_{1,1}x_1 + \dots + a_{1,n}x_n \leq b_1 \\ & && \vdots \\ & && a_{m,1}x_1 + \dots + a_{m,n}x_n \leq b_m \\ & && x_1 \geq 0 \\ & && \vdots \\ & && x_n \geq 0 \end{aligned}$$

For every choice of non-negative scaling factors y_1, \dots, y_m , we can derive the inequality

$$\begin{aligned} & y_1 \cdot (a_{1,1}x_1 + \dots + a_{1,n}x_n) \\ & + \dots \\ & + y_n \cdot (a_{m,1}x_1 + \dots + a_{m,n}x_n) \\ & \leq y_1b_1 + \dots + y_mb_m \end{aligned}$$

which is true for every feasible solution (x_1, \dots, x_n) to the linear program (2). We can rewrite the inequality as

$$\begin{aligned} & (a_{1,1}y_1 + \dots + a_{m,1}y_m) \cdot x_1 \\ & + \dots \end{aligned}$$

$$\begin{aligned}
& + (a_{1,n}y_1 \cdots a_{m,n}y_m) \cdot x_n \\
& \leq y_1b_1 + \cdots y_mb_m
\end{aligned}$$

So we get that a certain linear function of the x_i is always at most a certain value, for every feasible (x_1, \dots, x_n) . The trick is now to choose the y_i so that the linear function of the x_i for which we get an upper bound is, in turn, an upper bound to the cost function of (x_1, \dots, x_n) . We can achieve this if we choose the y_i such that

$$\begin{aligned}
c_1 & \leq a_{1,1}y_1 + \cdots a_{m,1}y_m \\
& \vdots \\
c_n & \leq a_{1,n}y_1 + \cdots a_{m,n}y_m
\end{aligned}$$

Now we see that for every non-negative (y_1, \dots, y_m) that satisfies (3), and for every (x_1, \dots, x_n) that is feasible for (2),

$$\begin{aligned}
& c_1x_1 + \cdots c_nx_n \\
& \leq (a_{1,1}y_1 + \cdots a_{m,1}y_m) \cdot x_1 \\
& \quad + \cdots \\
& \quad + (a_{1,n}y_1 + \cdots a_{m,n}y_m) \cdot x_n \\
& \leq y_1b_1 + \cdots y_mb_m
\end{aligned}$$

Clearly, we want to find the non-negative values y_1, \dots, y_m such that the above upper bound is as strong as possible, that is we want to

$$\begin{aligned}
\text{minimize} \quad & b_1y_1 + \cdots b_my_m \\
\text{subject to} \quad & a_{1,1}y_1 + \cdots + a_{m,1}y_m \geq c_1 \\
& \vdots \\
& a_{n,1}y_1 + \cdots + a_{m,n}y_m \geq c_n \\
& y_1 \geq 0 \\
& \vdots \\
& y_m \geq 0
\end{aligned}$$

So we find out that if we want to find the scaling factors that give us the best possible upper bound to the optimum of a linear program in standard maximization form, we end up with a new linear program, in standard minimization form. Definition 1 If

$$\begin{aligned}
& \mathbf{c}^T \mathbf{x} \\
\text{maximize} \quad & \\
\text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\
& \mathbf{x} \geq 0
\end{aligned}$$

is a linear program in maximization standard form, then its dual is the minimization linear program

$$\begin{aligned} & \text{minimize}_{\mathbf{y}} \quad \mathbf{b}^T \mathbf{y} \\ & \text{subject to} \\ & \quad A^T \mathbf{y} \geq \mathbf{c} \\ & \quad \mathbf{y} \geq 0 \end{aligned}$$

So if we have a linear program in maximization linear form, which we are going to call the primal linear program, its dual is formed by having one variable for each constraint of the primal (not counting the non-negativity constraints of the primal variables), and having one constraint for each variable of the primal (plus the nonnegative constraints of the dual variables); we change maximization to minimization, we switch the roles of the coefficients of the objective function and of the right-hand sides of the inequalities, and we take the transpose of the matrix of coefficients of the left-hand side of the inequalities.

The optimum of the dual is now an upper bound to the optimum of the primal. How do we do the same thing but starting from a minimization linear program? We can rewrite

$$\begin{aligned} & \text{minimize}_{\mathbf{y}} \quad \mathbf{c}^T \mathbf{y} \\ & \text{subject to} \\ & \quad A \mathbf{y} \leq \mathbf{b} \\ & \quad \mathbf{y} \geq 0 \end{aligned}$$

in an equivalent way as

$$\begin{aligned} & \text{mubject to}_{\mathbf{y}} \quad -\mathbf{c}^T \mathbf{y} \\ & \text{maximize} \\ & \quad -A \mathbf{y} \leq -\mathbf{b} \\ & \quad \mathbf{y} \geq 0 \end{aligned}$$

If we compute the dual of the above program we get

$$\begin{aligned} & \text{mubject to}_{\mathbf{z}} \quad -\mathbf{b}^T \mathbf{z} \\ & \text{minimize} \\ & \quad -A^T \mathbf{z} \geq -\mathbf{c} \\ & \quad \mathbf{z} \geq 0 \end{aligned}$$

that is,

$$\begin{aligned} & \text{maximize}_{\mathbf{z}} \quad \mathbf{b}^T \mathbf{z} \\ & \text{subject to} \\ & \quad A^T \mathbf{z} \leq \mathbf{c} \\ & \quad \mathbf{z} \geq 0 \end{aligned}$$

So we can form the dual of a linear program in minimization normal form in the same way in which we formed the dual in the maximization case:

- switch the type of optimization,
- introduce as many dual variables as the number of primal constraints (not counting the non-negativity constraints),
- define as many dual constraints (not counting the non-negativity constraints) as the number of primal variables.
- take the transpose of the matrix of coefficients of the left-hand side of the inequality,
- switch the roles of the vector of coefficients in the objective function and the vector of right-hand sides in the inequalities.

Note that:

Fact 2 The dual of the dual of a linear program is the linear program itself.

We have already proved the following:

Fact 3 If the primal (in maximization standard form) and the dual (in minimization standard form) are both feasible, then

$$\text{opt(primal)} \leq \text{opt(dual)}$$

Which we can generalize a little

Theorem 4 (Weak Duality Theorem) If LP_1 is a linear program in maximization standard form, LP_2 is a linear program in minimization standard form, and LP_1 and LP_2 are duals of each other then:

- If LP_1 is unbounded, then LP_2 is infeasible; - If LP_2 is unbounded, then LP_1 is infeasible;
- If LP_1 and LP_2 are both feasible and bounded, then

$$\text{opt}(LP_1) \leq \text{opt}(LP_2)$$

ProOF: We have proved the third statement already. Now observe that the third statement is also saying that if LP_1 and LP_2 are both feasible, then they have to both be bounded, because every feasible solution to LP_2 gives a finite upper bound to the optimum of LP_1 (which then cannot be $+\infty$) and every feasible solution to LP_1 gives a finite lower bound to the optimum of LP_2 (which then cannot be $-\infty$).

What is surprising is that, for bounded and feasible linear programs, there is always a dual solution that certifies the exact value of the optimum.

Theorem 5 (Strong Duality) If either LP_1 or LP_2 is feasible and bounded, then so is the other, and

$$\text{opt}(LP_1) = \text{opt}(LP_2)$$

To summarize, the following cases can arise:

- If one of LP_1 or LP_2 is feasible and bounded, then so is the other;

- If one of LP_1 or LP_2 is unbounded, then the other is infeasible;
- If one of LP_1 or LP_2 is infeasible, then the other cannot be feasible and bounded, that is, the other is going to be either infeasible or unbounded. Either case can happen.

9.2 Linear programming duality

Consider the following problem:

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A} \mathbf{x} \geq \mathbf{b}. \end{aligned} \tag{9.1}$$

In the remark at the end of Chapter ??, we saw that if (9.1) has an optimal solution, then there exists $\mathbf{y}^* \in \mathbb{R}^m$ such that $\mathbf{y}^* \geq 0$, $\mathbf{y}^{*\top} \mathbf{A} = \mathbf{c}^T$, and $\mathbf{y}^{*\top} \mathbf{b} = \gamma$ where γ denotes the optimal value of (9.1).

Take any $\mathbf{y} \in \mathbb{R}^m$ satisfying $\mathbf{y} \geq 0$ and $\mathbf{y}^T \mathbf{A} = \mathbf{c}^T$. Then we can infer from $\mathbf{A} \mathbf{x} \geq \mathbf{b}$ the inequality $\mathbf{y}^T \mathbf{A} \mathbf{x} \geq \mathbf{y}^T \mathbf{b}$, or more simply, $\mathbf{c}^T \mathbf{x} \geq \mathbf{y}^T \mathbf{b}$. Thus, for any such \mathbf{y} , $\mathbf{y}^T \mathbf{b}$ gives a lower bound for the objective function value of any feasible solution to (9.1). Since γ is the optimal value of (P), we must have $\gamma \geq \mathbf{y}^T \mathbf{b}$.

As $\mathbf{y}^{*\top} \mathbf{b} = \gamma$, we see that γ is the optimal value of

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{A} = \mathbf{c}^T \\ & \mathbf{y} \geq 0. \end{aligned} \tag{9.2}$$

Note that (9.2) is a linear programming problem! We call it the **dual problem** of the **primal problem** (9.1). We say that the dual variable y_i is **associated** with the constraint $\mathbf{a}^{(i)\top} \mathbf{x} \geq b_i$ where $\mathbf{a}^{(i)\top}$ denotes the i th row of \mathbf{A} .

In other words, we define the dual problem of (9.1) to be the linear programming problem (9.2). In the discussion above, we saw that if the primal problem has an optimal solution, then so does the dual problem and the optimal values of the two problems are equal. Thus, we have the following result:

Theorem 9.1: strong-duality-special

Suppose that (9.1) has an optimal solution. Then (9.2) also has an optimal solution and the optimal values of the two problems are equal.

At first glance, requiring all the constraints to be \geq -inequalities as in (9.1) before forming the dual problem seems a bit restrictive. We now see how the dual problem of a primal problem in general form can be defined. We first make two observations that motivate the definition.

Observation 1

Suppose that our primal problem contains a mixture of all types of linear constraints:

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{Ax} \geq \mathbf{b} \\ & \mathbf{A}' \mathbf{x} \leq \mathbf{b}' \\ & \mathbf{A}'' \mathbf{x} = \mathbf{b}'' \end{aligned} \tag{9.3}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{A}' \in \mathbb{R}^{m' \times n}$, $\mathbf{b}' \in \mathbb{R}^{m'}$, $\mathbf{A}'' \in \mathbb{R}^{m'' \times n}$, and $\mathbf{b}'' \in \mathbb{R}^{m''}$.

We can of course convert this into an equivalent problem in the form of (9.1) and form its dual.

However, if we take the point of view that the function of the dual is to infer from the constraints of (9.3) an inequality of the form $\mathbf{c}^T \mathbf{x} \geq \gamma$ with γ as large as possible by taking an appropriate linear combination of the constraints, we are effectively looking for $\mathbf{y} \in \mathbb{R}^m$, $\mathbf{y} \geq 0$, $\mathbf{y}' \in \mathbb{R}^{m'}$, $\mathbf{y}' \leq 0$, and $\mathbf{y}'' \in \mathbb{R}^{m''}$, such that

$$\mathbf{y}^T \mathbf{A} + \mathbf{y}'^T \mathbf{A}' + \mathbf{y}''^T \mathbf{A}'' = \mathbf{c}^T$$

with $\mathbf{y}^T \mathbf{b} + \mathbf{y}'^T \mathbf{b}' + \mathbf{y}''^T \mathbf{b}''$ to be maximized.

(The reason why we need $\mathbf{y}' \leq 0$ is because inferring a \geq -inequality from $\mathbf{A}' \mathbf{x} \leq \mathbf{b}'$ requires nonpositive multipliers. There is no restriction on \mathbf{y}'' because the constraints $\mathbf{A}'' \mathbf{x} = \mathbf{b}''$ are equalities.)

This leads to the dual problem:

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} + \mathbf{y}'^T \mathbf{b}' + \mathbf{y}''^T \mathbf{b}'' \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{A} + \mathbf{y}'^T \mathbf{A}' + \mathbf{y}''^T \mathbf{A}'' = \mathbf{c}^T \\ & \mathbf{y} \geq 0 \\ & \mathbf{y}' \leq 0. \end{aligned} \tag{9.4}$$

In fact, we could have derived this dual by applying the definition of the dual problem to

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \begin{bmatrix} \mathbf{A} \\ -\mathbf{A}' \\ \mathbf{A}'' \\ -\mathbf{A}'' \end{bmatrix} \mathbf{x} \geq \begin{bmatrix} \mathbf{b} \\ -\mathbf{b}' \\ \mathbf{b}'' \\ -\mathbf{b}'' \end{bmatrix}, \end{aligned}$$

which is equivalent to (9.3). The details are left as an exercise.

Observation 2

Consider the primal problem of the following form:

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{Ax} \geq \mathbf{b} \\ & x_i \geq 0 \quad i \in P \\ & x_i \leq 0 \quad i \in N \end{aligned} \tag{9.5}$$

where P and N are disjoint subsets of $\{1, \dots, n\}$. In other words, constraints of the form $x_i \geq 0$ or $x_i \leq 0$ are separated out from the rest of the inequalities.

Forming the dual of (9.5) as defined under Observation 1, we obtain the dual problem

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{a}^{(i)} = c_i \quad i \in \{1, \dots, n\} \setminus (P \cup N) \\ & \mathbf{y}^T \mathbf{a}^{(i)} + p_i = c_i \quad i \in P \\ & \mathbf{y}^T \mathbf{a}^{(i)} + q_i = c_i \quad i \in N \\ & p_i \geq 0 \quad i \in P \\ & q_i \leq 0 \quad i \in N \end{aligned} \tag{9.6}$$

where $\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$. Note that this problem is equivalent to the following without the variables p_i , $i \in P$ and q_i , $i \in N$:

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{a}^{(i)} = c_i \quad i \in \{1, \dots, n\} \setminus (P \cup N) \\ & \mathbf{y}^T \mathbf{a}^{(i)} \leq c_i \quad i \in P \\ & \mathbf{y}^T \mathbf{a}^{(i)} \geq c_i \quad i \in N, \end{aligned} \tag{9.7}$$

which can be taken as the dual problem of (9.5) instead of (9.6). The advantage here is that it has fewer variables than (9.6).

Hence, the dual problem of

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A} \mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

is simply

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T \\ & \mathbf{y} \geq 0. \end{aligned}$$

As we can see from above, there is no need to associate dual variables to constraints of the form $x_i \geq 0$ or $x_i \leq 0$ provided we have the appropriate types of constraints in the dual problem. Combining all the observations lead to the definition of the dual problem for a primal problem in general form as discussed next.

9.2.1. The dual problem

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{c} \in \mathbb{R}^n$. Let $\mathbf{a}^{(i)^\top}$ denote the i th row of \mathbf{A} . Let \mathbf{A}_j denote the j th column of \mathbf{A} .

Let (P) denote the minimization problem with variables in the tuple $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ given as follows:

- The objective function to be minimized is $\mathbf{c}^\top \mathbf{x}$
- The constraints are

$$\mathbf{a}^{(i)^\top} \mathbf{x} \sqcup_i b_i$$

where \sqcup_i is \leq , \geq , or $=$ for $i = 1, \dots, m$.

- For each $j \in \{1, \dots, n\}$, x_j is constrained to be nonnegative, nonpositive, or free (i.e. not constrained to be nonnegative or nonpositive.)

Then the **dual problem** is defined to be the maximization problem with variables in the tuple $\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$ given as follows:

- The objective function to be maximized is $\mathbf{y}^\top \mathbf{b}$
- For $j = 1, \dots, n$, the j th constraint is

$$\begin{cases} \mathbf{y}^\top \mathbf{A}_j \leq c_j & \text{if } x_j \text{ is constrained to be nonnegative} \\ \mathbf{y}^\top \mathbf{A}_j \geq c_j & \text{if } x_j \text{ is constrained to be nonpositive} \\ \mathbf{y}^\top \mathbf{A}_j = c_j & \text{if } x_j \text{ is free.} \end{cases}$$

- For each $i \in \{1, \dots, m\}$, y_i is constrained to be nonnegative if \sqcup_i is \geq ; y_i is constrained to be nonpositive if \sqcup_i is \leq ; y_i is free if \sqcup_i is $=$.

The following table can help remember the above.

Primal (min)	Dual (max)
\geq constraint	≥ 0 variable
\leq constraint	≤ 0 variable
$=$ constraint	free variable
≥ 0 variable	\leq constraint
≤ 0 variable	\geq constraint
free variable	$=$ constraint

Below is an example of a primal-dual pair of problems based on the above definition:

Consider the primal problem:

$$\begin{array}{lllll} \min & x_1 & - & 2x_2 & + & 3x_3 \\ \text{s.t.} & -x_1 & & & + & 4x_3 = 5 \\ & 2x_1 & + & 3x_2 & - & 5x_3 \geq 6 \\ & & & & 7x_2 & \leq 8 \\ & x_1 & & & & \geq 0 \\ & & x_2 & & & \text{free} \\ & & & & x_3 & \leq 0. \end{array}$$

Here, $\mathbf{A} = \begin{bmatrix} -1 & 0 & 4 \\ 2 & 3 & -5 \\ 0 & 7 & 0 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix}$, and $\mathbf{c} = \begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix}$.

The primal problem has three constraints. So the dual problem has three variables. As the first constraint in the primal is an equation, the corresponding variable in the dual is free. As the second constraint in the primal is a \geq -inequality, the corresponding variable in the dual is nonnegative. As the third constraint in the primal is a \leq -inequality, the corresponding variable in the dual is nonpositive. Now, the primal problem has three variables. So the dual problem has three constraints. As the first variable in the primal is nonnegative, the corresponding constraint in the dual is a \leq -inequality. As the second variable in the primal is free, the corresponding constraint in the dual is an equation. As the third variable in the primal is nonpositive, the corresponding constraint in the dual is a \geq -inequality. Hence, the dual problem is:

$$\begin{array}{lllll} \max & 5y_1 & + & 6y_2 & + & 8y_3 \\ \text{s.t.} & -y_1 & + & 2y_2 & & \leq 1 \\ & & & 3y_2 & + & 7y_3 = -2 \\ & 4y_1 & - & 5y_2 & & \geq 3 \\ & y_1 & & & & \text{free} \\ & & y_2 & & & \geq 0 \\ & & & & y_3 & \leq 0. \end{array}$$

Remarks. Note that in some books, the primal problem is always a maximization problem. In that case, what is our primal problem is their dual problem and what is our dual problem is their primal problem.

One can now prove a more general version of Theorem 9.2 as stated below. The details are left as an exercise.

Theorem 9.2: Duality Theorem for Linear Programming

Let (P) and (D) denote a primal-dual pair of linear programming problems. If either (P) or (D) has an optimal solution, then so does the other. Furthermore, the optimal values of the two problems are equal.

Theorem 9.2.1 is also known informally as **strong duality**.

Exercises

1. Write down the dual problem of

$$\begin{array}{lll} \min & 4x_1 - 2x_2 \\ \text{s.t.} & x_1 + 2x_2 \geq 3 \\ & 3x_1 - 4x_2 = 0 \\ & x_2 \geq 0. \end{array}$$

2. Write down the dual problem of the following:

$$\begin{array}{lll} \min & 3x_2 + x_3 \\ \text{s.t.} & x_1 + x_2 + 2x_3 = 1 \\ & x_1 - 3x_3 \leq 0 \\ & x_1, x_2, x_3 \geq 0. \end{array}$$

3. Write down the dual problem of the following:

$$\begin{array}{lll} \min & x_1 - 9x_3 \\ \text{s.t.} & x_1 - 3x_2 + 2x_3 = 1 \\ & x_1 \leq 0 \\ & x_2 \text{ free} \\ & x_3 \geq 0. \end{array}$$

4. Determine all values c_1, c_2 such that the linear programming problem

$$\begin{array}{ll} \min & c_1x_1 + c_2x_2 \\ \text{s.t.} & 2x_1 + x_2 \geq 2 \\ & x_1 + 3x_2 \geq 1. \end{array}$$

has an optimal solution. Justify your answer

Solutions

1. The dual is

$$\begin{array}{lll} \max & 3y_1 \\ \text{s.t.} & y_1 + 3y_2 = 4 \\ & 2y_1 - 4y_2 \leq -2 \\ & y_1 \geq 0. \end{array}$$

2. The dual is

$$\begin{array}{lll} \max & y_1 \\ \text{s.t.} & y_1 + y_2 \leq 0 \\ & y_1 \leq 3 \\ & 2y_1 - 3y_2 \leq 1 \\ & y_1 \text{ free} \\ & y_2 \leq 0. \end{array}$$

3. The dual is

$$\begin{array}{ll} \max & y_1 \\ \text{s.t.} & y_1 \geq 1 \\ & -3y_1 = 0 \\ & 2y_1 \leq -9 \\ & y_1 \text{ free.} \end{array}$$

4. Let (P) denote the given linear programming problem.

Note that $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ is a feasible solution to (P). Therefore, by Theorem ??, it suffices to find all values c_1, c_2 such that

(P) is not unbounded. This amounts to finding all values c_1, c_2 such that the dual problem of (P) has a feasible solution.

The dual problem of (P) is

$$\begin{array}{ll} \max & 2y_1 + y_2 \\ & 2y_1 + y_2 = c_1 \\ & y_1 + 3y_2 = c_2 \\ & y_1, y_2 \geq 0. \end{array}$$

The two equality constraints gives $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{3}{5}c_1 - \frac{1}{5}c_2 \\ -\frac{1}{5}c_1 + \frac{2}{5}c_2 \end{bmatrix}$. Thus, the dual problem is feasible if and only if c_1 and c_2 are real numbers satisfying

$$\begin{array}{ll} \frac{3}{5}c_1 - \frac{1}{5}c_2 \geq & 0 \\ -\frac{1}{5}c_1 + \frac{2}{5}c_2 \geq & 0, \end{array}$$

or more simply,

$$\frac{1}{3}c_2 \leq c_1 \leq 2c_2.$$

10. Sensitivity Analysis

Sensitivity analysis is an important tool for understanding the behavior of linear programs. It allows us to analyze how the optimal solution of a linear program changes as the coefficients of the constraints or the objective function are varied within certain bounds. In this section, we will present the basic concepts of sensitivity analysis and provide two examples to illustrate its use.

The first step in sensitivity analysis is to solve the linear program using a method such as the simplex method. Once the optimal solution has been obtained, we can then analyze how the solution changes as the coefficients of the constraints or the objective function are varied. For example, consider the following linear program:

$$\text{Maximize } 3x_1 + 2x_2 \text{ Subject to } \begin{array}{l} x_1 + x_2 \leq 4 \\ 2x_1 + x_2 \leq 5 \\ x_1, x_2 \geq 0 \end{array}$$

The optimal solution to this linear program is $x_1 = 2$, $x_2 = 2$, with an optimal objective value of $3x_1 + 2x_2 = 10$.

To perform sensitivity analysis, we can consider how the optimal solution changes as the right-hand side (RHS) of the constraints is varied. For example, suppose we increase the RHS of the first constraint by 1, to 5. This corresponds to the modified constraint $x_1 + x_2 \leq 5$. The optimal solution to this modified linear program is $x_1 = 2$, $x_2 = 3$, with an optimal objective value of $3x_1 + 2x_2 = 12$. Thus, we can see that increasing the RHS of the first constraint by 1 has led to an increase in the optimal objective value.

Another example of sensitivity analysis is consider the effect of changing the coefficient of the objective function. For example, suppose we multiply the coefficient of x_1 by 2, to obtain the modified objective function $6x_1 + 2x_2$. The optimal solution to this modified linear program is $x_1 = 1$, $x_2 = 2$, with an optimal objective value of $6x_1 + 2x_2 = 8$. Thus, we can see that multiplying the coefficient of x_1 by 2 has led to a decrease in the optimal objective value.

In summary, sensitivity analysis is a useful tool for understanding how the optimal solution of a linear program changes as the coefficients of the constraints or the objective function are varied. It allows us to determine how robust the solution is to changes in the input data, and to identify the most critical factors affecting the solution.

11. Multi-Objective Optimization

Outcomes

- Define multi objective optimization problems
- Discuss the solutions in terms of the Pareto Frontier
- Explore approaches for finding the Pareto Frontier
- Use software to solve for or approximate the Pareto Frontier

Resources

Python Multi Objective Optimization (Pymoo)

11.1 Multi Objective Optimization and The Pareto Frontier

On Dealing with Ties and Multiple Objectives in Linear Programming

Consider a high end furniture manufacturer which builds dining tables and chairs out of expensive bocote and rosewood.

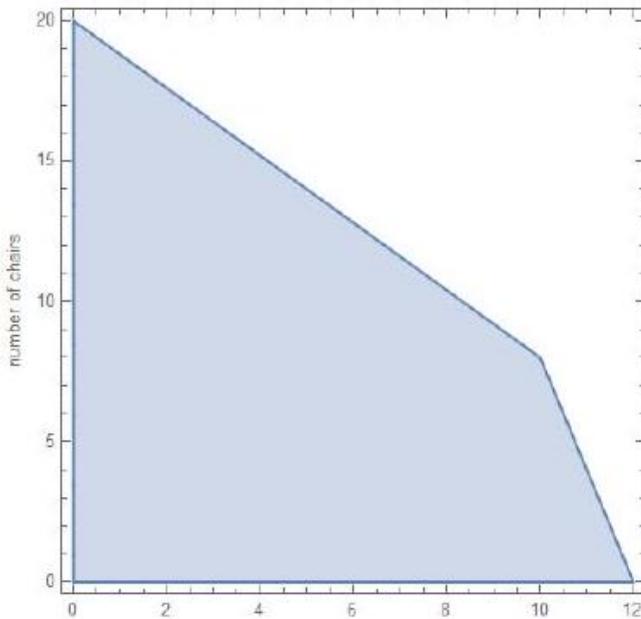


The manufacturer has an ongoing deal with a foreign sawmill which supplies them with 960 and 200 board-feet (bdft) of bocote and rosewood respectively each month.

A single table requires 80 bdft of bocote and 20 bdft of rosewood.

Each chair requires only 20 bdft of bocote but 10 bdft of rosewood.

$$\begin{aligned} P = \{(x,y) \in \mathbb{R}^2 : \\ 80x + 20y \leq 960 \\ 12x + 10y \leq 200 \\ x, y \geq 0\} \end{aligned}$$



Suppose that each table will sell for \$7000 while a chair goes for \$1500. To increase profit we want to maximize

$$F(x, y) = 8000x + 2000y$$

over P . Having taken a linear programming class, the manager knows his way around these problems and begins the simplex method:

Maximize $8000x + 2000y$

$$\text{s.t. } 80x + 20y \leq 960$$

$$12x + 10y \leq 200$$

$$x, y \geq 0$$

-4	-1	0	0	0
4	1	1	0	48
6	5	0	1	100

Maximize $4x + y$

$$\text{s.t. } 4x + y + s_1 = 48$$

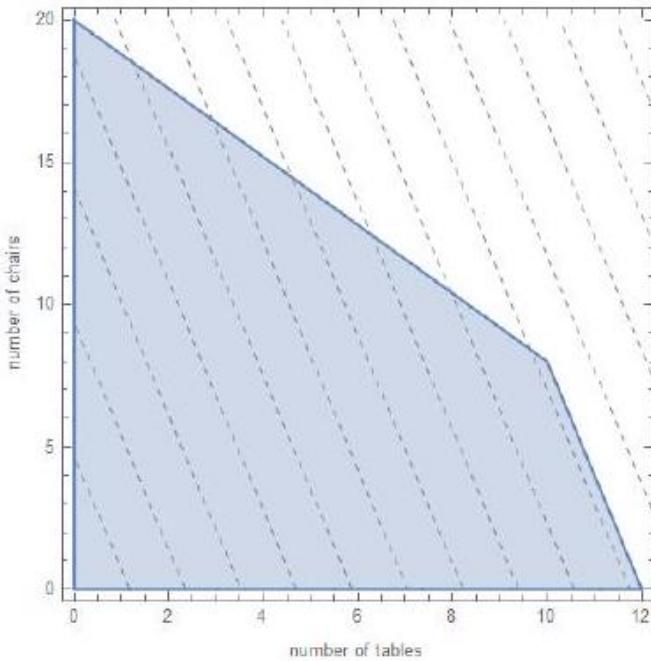
$$\begin{array}{ll} f & 6x + 5y + s_2 = 100 \\ 2000 & x, y \geq 0 \end{array}$$

Having found an optimal solution, the manager is quick to set up production. The best thing to do is produce 12 tables a month and no chairs!

But there are actually multiple optima!

How could we have noticed this from the tableau? From the original formulation?

Is the manager's solution really the best?



Having fired the prior manager for producing no chairs, a new and more competent manager is hired. This one knows that *Dalbergia stevensonii* (the tree which produces their preferred rosewood) is a threatened species and decides that she doesn't want to waste any more rosewood than is necessary.

After some investigation, she finds that table production wastes nearly 10 bdft of rosewood per table while chairs are dramatically more efficient wasting only 2 bdft per chair. She comes up with a new, secondary objective function that she would like to minimize:

$$w(x, y) = 10x + 2y.$$

Having noticed that there are multiple profit-maximizers, she formulates a new problem to break the tie:

$$\begin{aligned} & \text{Minimize } 10x + 2y \\ \text{s.t. } & 80x + 20y = 960 \\ & x \in [10, 12] \\ & y \in [0, 8]. \end{aligned}$$

This is easy in this case because the set of profit-optimal solutions is simple.

Because this is an LP, the optimal solution will be at an extreme point; there are only two here, so the problem reduces to

$$\arg \min \{10x + 2y : (x, y) \in \{(12, 0), (10, 8)\}\}$$

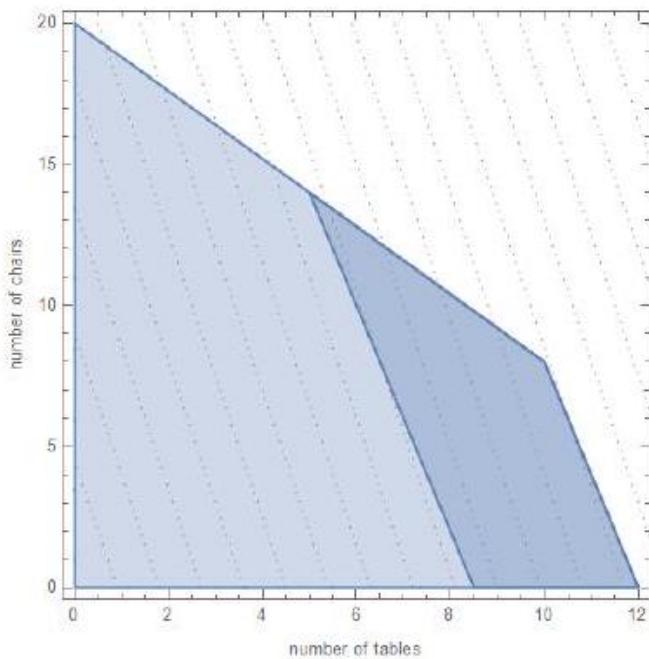
Therefore, swapping out some tables for chairs reduces waste and without affecting revenue!

What the manager just did is called the Ordered Criteria or Lexicographic method for Multi-Objective Optimization. After a few months, the manager convinces the owners that reducing waste is worth a small

loss in profit. The owners concede to a 30% loss in revenue and our manager gets to work on a new model:

$$\begin{aligned}
 & \text{Minimize } 10x + 2y \\
 \text{s.t. } & 8000x + 2000y \geq (\alpha)96000 \\
 & 80x + 20y \leq 960 \\
 & 12x + 10y \leq 200 \\
 & x, y \geq 0
 \end{aligned}$$

where $\alpha = 0.7$. This new constraint limits us to solutions which offered at least 70% of maximum possible revenue.



The strategy is called the Benchmark or Rollover method because we choose a benchmark for one of our objectives (revenue in this case), roll that benchmark into the constraints, and optimize for the second objective (waste).

Notice that if we set α to 1, the rollover problem is equivalent to the lexicographic problem. Either approach requires a known optimal value to the first objective function.

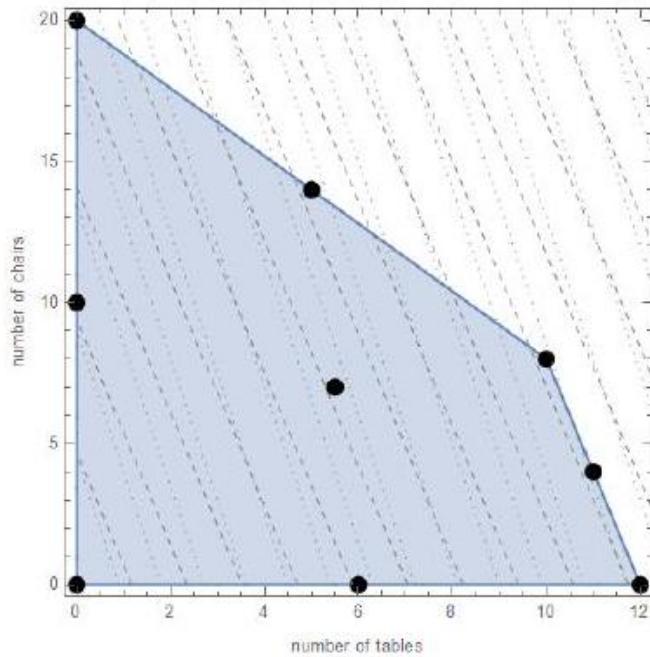
Interestingly, our rollover solution is NOT an extreme point to the ORIGINAL feasible region. Given a set P and some number of functions $f_i : P \rightarrow \mathbb{R}$ that we seek to maximize, we call a point $\mathbf{x} \in P$ Pareto Optimal or Efficient if there does not exist another point $\bar{\mathbf{x}} \in P$ such that

- $f_i(\bar{\mathbf{x}}) > f_i(\mathbf{x})$ for some i and
 $\rightarrow f_j(\bar{\mathbf{x}}) \geq f_j(\mathbf{x})$ for all $j \neq i$.

That is, we cannot make any objective better without making some other objective worse.

The Pareto Frontier is the set of all Pareto optimal points for some problem. Which of these points is Pareto optimal?

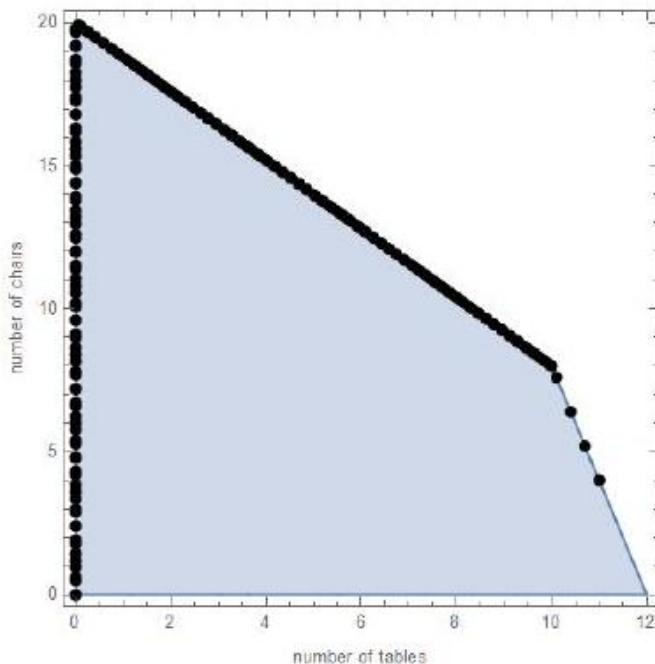
What is the frontier of this problem?



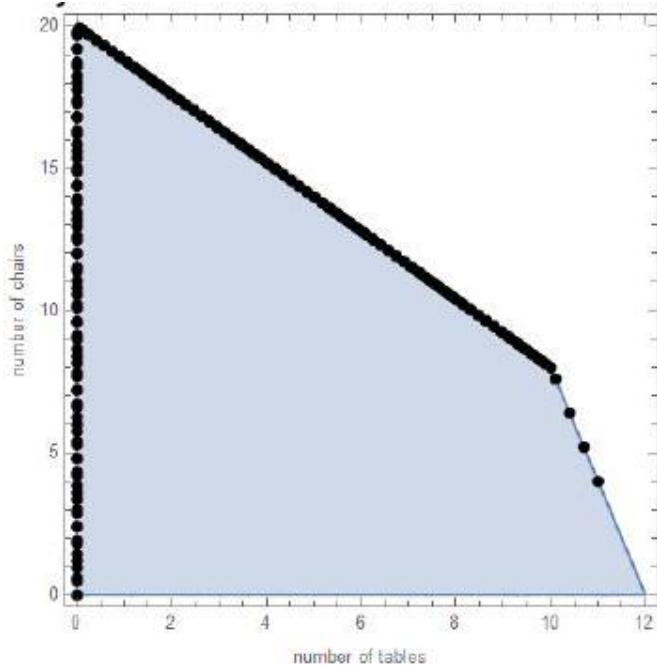
number of tables The rollover method is generalized in Goal Programming

By varying α , it is possible to generate many distinct efficient solutions.

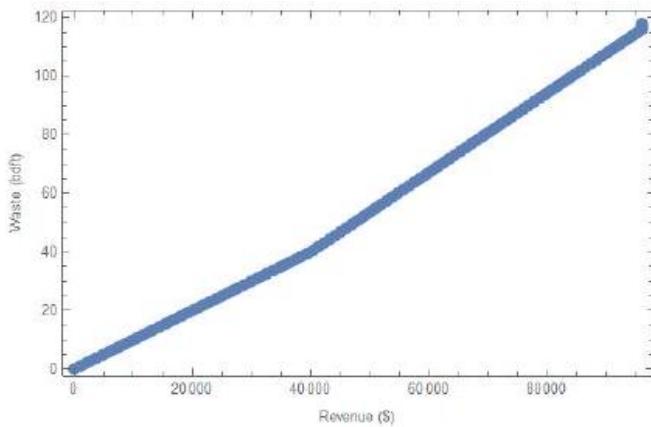
However, this method can generate inefficient solutions if the underlying model is poorly constructed.



number of tables It is more common to see a Pareto frontier plotted with respect to its objectives.



number of table



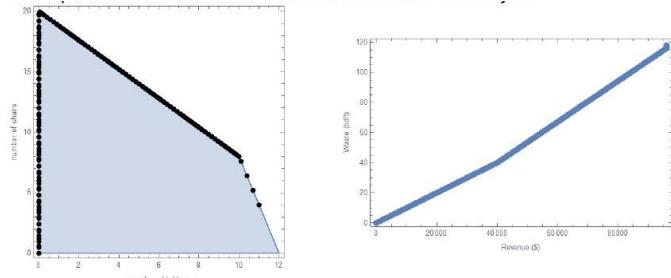
One of the owners of our manufactory decides to explore possible planning himself; he implements the multi-objective method that he remembers, Scalarization by picking some arbitrary constant $\lambda \in [0, 1]$ and combining his two objectives like so:

$$\begin{aligned} \text{Minimize} \quad & \lambda(8000x + 2000y) + (1 - \lambda)(10x + 2y) \\ \text{s.t.} \quad & 80x + 20y \leq 960 \\ & 12x + 10y \leq 200 \\ & x, y \geq 0 \end{aligned}$$

What is the benefit of this method?

Where does it fall short?

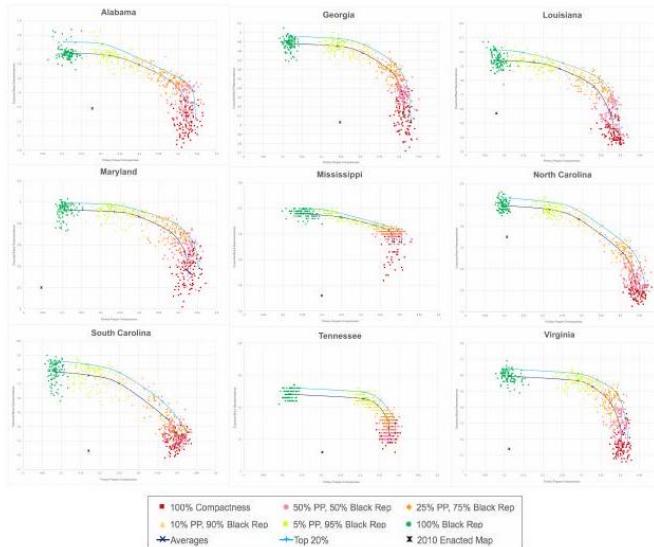
11.2 What points will the Scalarization method find if we vary λ ?



These are all nice ideas, but the problem presented above is neither difficult nor practical.

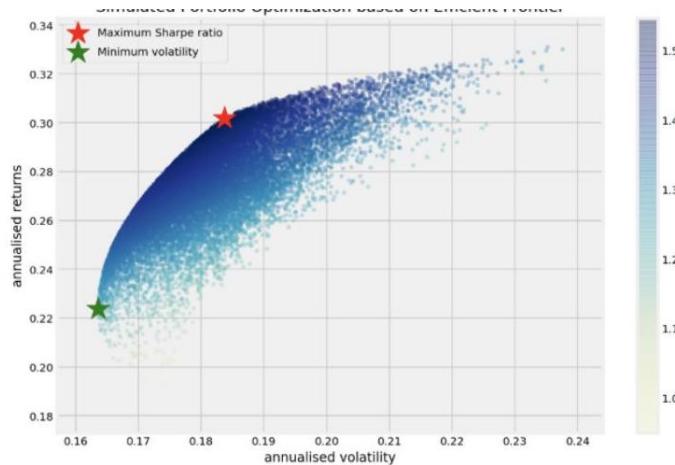
What are some areas that a Pareto frontier would be actually useful?

11.3 Political Redistricting [3]

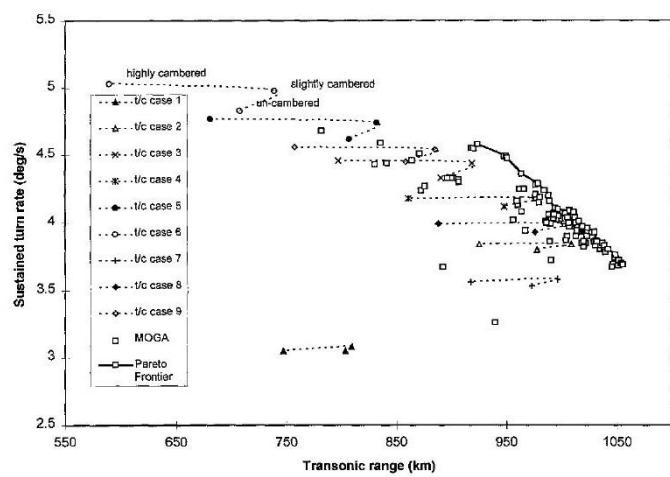


11.4 Portfolio Optimization [5]

11.5 Simulated Portfolio Optimization based on Efficient Frontier



11.6 Aircraft Design [1]



11.7 Vehicle Dynamics [4]

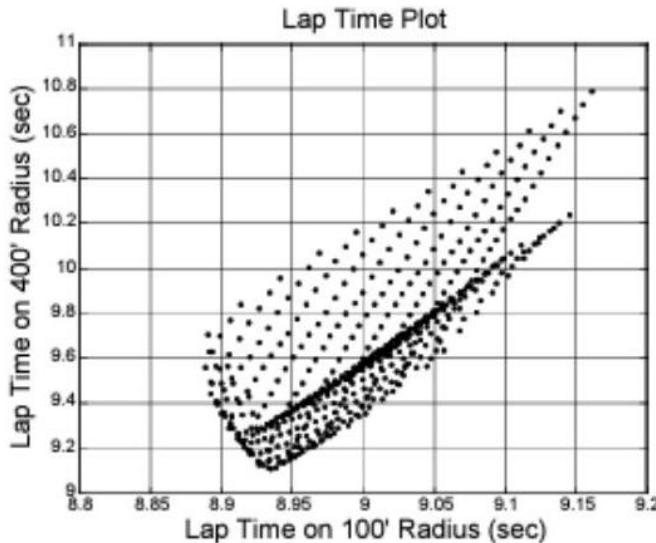
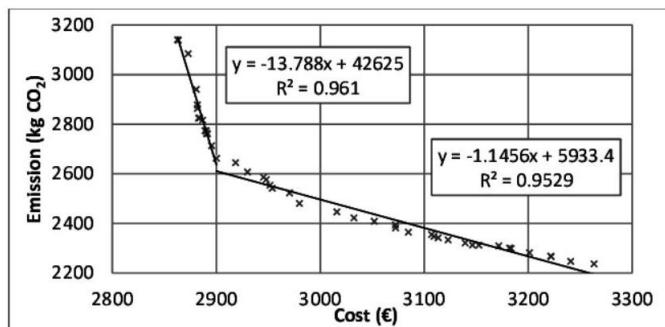


Figure 7: Grid Search Results in the Performance Space

11.8 Sustainable Constriction [2]



11.9 References

S. Fenwick and John C. Harris. the application of pareto frontier methods in the multidisciplinary wing design of a generic modern military delta aircraft: Semantic scholar, Jan 1999.

URL: <https://WWW.semanticscholar.org/paper/>

The-application-of-Pareto-frontier-methods-in-the-a-Fenwick-Harris/ fced 00a59 d200c2c74 ed 655 a 457344 bcleea 6 ff 5.

T García-Segura, V Yepes, and J Alcalá.

Sustainable design using multiobjective optimization of high-strength concrete i-beams. In The 2014 International Conference on High Performance and Optimum Design of Structures and Materials HPSM/OPTI, volume 137, pages 347 – 358, 2014.

URL: https://www.researchgate.net/publication/271439836_Sustainable_design_using_multiobjective_optimization_of_high-strength_concrete_l-beams.

Nicholas Goedert, Robert Hildebrand, Laurel Travis, Matthew Pierson, and Jamie Fravel. Black representation and district compactness in southern congressional districts. not yet published, ask Dr. Hildebrand for it.

Edward M Kasprzak and Kemper E Lewis.

Pareto analysis in multiobjective optimization using the collinearity theorem and scaling method. Structural and Multidisciplinary Optimization.

Ricky Kim.

Efficient frontier portfolio optimisation in python, Jun 2021.

URL: <https://towardsdatascience.com/efficient-frontier-portfolio-optimisation-in-python-e7844051e7f>.

Part II

Discrete Algorithms

12. Graph Algorithms

12.1 Graph Theory and Network Flows

In the modern world, planning efficient routes is essential for business and industry, with applications as varied as product distribution, laying new fiber optic lines for broadband internet, and suggesting new friends within social network websites like Facebook.

This field of mathematics started nearly 300 years ago as a look into a mathematical puzzle (we'll look at it in a bit). The field has exploded in importance in the last century, both because of the growing complexity of business in a global economy and because of the computational power that computers have provided us.

12.2 Graphs

12.2.1. Drawing Graphs

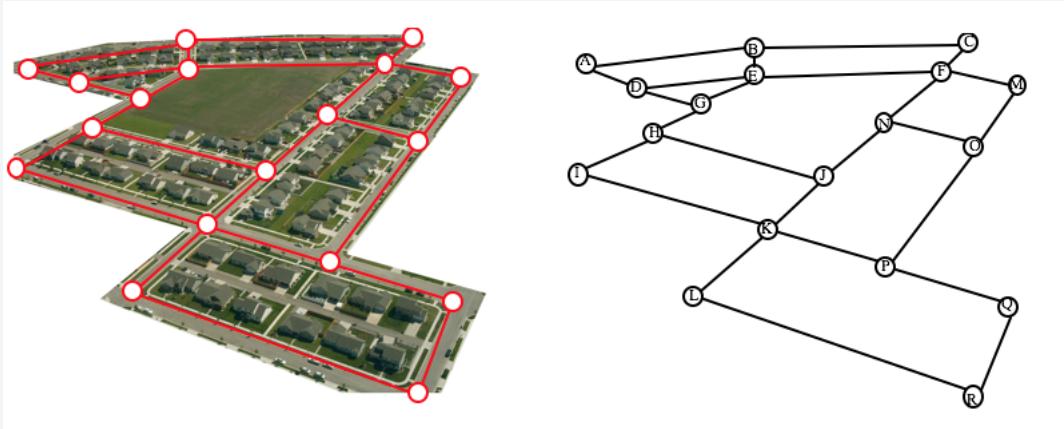
Example 12.1

Here is a portion of a housing development from Missoula, Montana^a. As part of her job, the development's lawn inspector has to walk down every street in the development making sure homeowners' landscaping conforms to the community requirements.



Naturally, she wants to minimize the amount of walking she has to do. Is it possible for her to walk down every street in this development without having to do any backtracking? While you might be able to answer that question just by looking at the picture for a while, it would be ideal to be able to answer the question for any picture regardless of its complexity.

To do that, we first need to simplify the picture into a form that is easier to work with. We can do that by drawing a simple line for each street. Where streets intersect, we will place a dot.



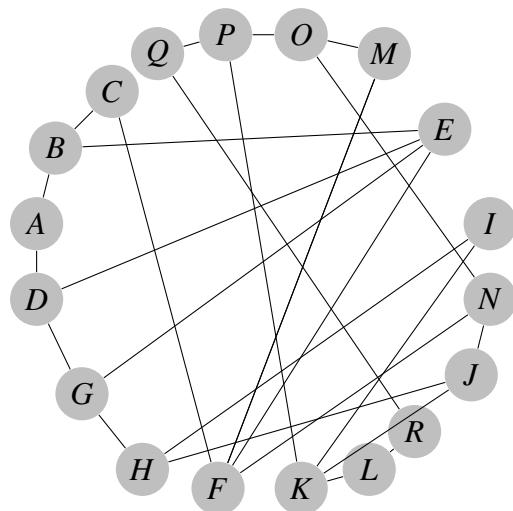
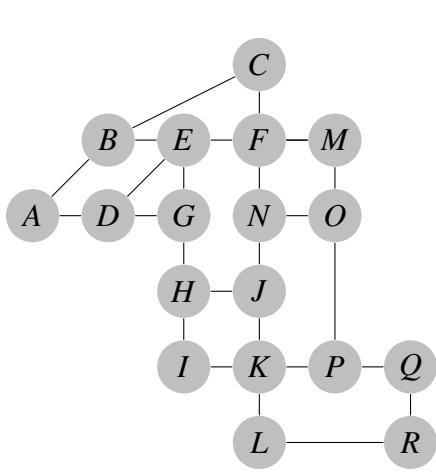
^aSame Beebe. <http://www.flickr.com/photos/sbeebe/2850476641/>

This type of simplified picture is called a **graph**.

Definition 12.2: Graphs, Vertices, and Edges

A graph consists of a set of dots, called vertices, and a set of edges connecting pairs of vertices.

While we drew our original graph to correspond with the picture we had, there is nothing particularly important about the layout when we analyze a graph. Both of the graphs below are equivalent to the one drawn above since they show the same edge connections between the same vertices as the original graph.



You probably already noticed that we are using the term graph differently than you may have used the term in the past to describe the graph of a mathematical function.

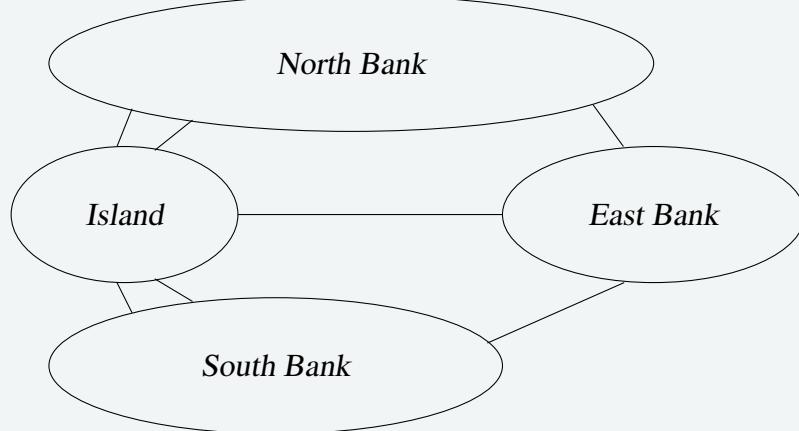
Example 12.3

Back in the 18th century in the Prussian city of Königsberg, a river ran through the city and seven bridges crossed the forks of the river. The river and the bridges are highlighted in the picture to the right

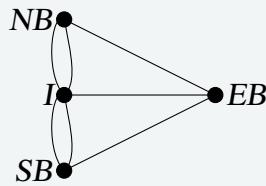
Picture

As a weekend amusement, townsfolk would see if they could find a route that would take them across every bridge once and return them to where they started.

Leonard Euler (pronounced OY-lur), one of the most prolific mathematicians ever, looked at this problem in 1735, laying the foundation for graph theory as a field in mathematics. To analyze this problem, Euler introduced edges representing the bridges:



Since the size of each land mass it is not relevant to the question of bridge crossings, each can be shrunk down to a vertex representing the location:



Notice that in this graph there are two edges connecting the north bank and island, corresponding to the two bridges in the original drawing. Depending upon the interpretation of edges and vertices appropriate to a scenario, it is entirely possible and reasonable to have more than one edge connecting two vertices.

While we haven't answered the actual question yet of whether or not there is a route which crosses every bridge once and returns to the starting location, the graph provides the foundation for exploring this question.

12.3 Definitions

While we loosely defined some terminology earlier, we now will try to be more specific.

Definition 12.4: Vertex

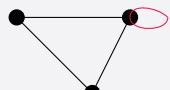
A vertex is a dot in the graph that could represent an intersection of streets, a land mass, or a general location, like "work?" or "school". Vertices are often connected by edges. Note that vertices only occur when a dot is explicitly placed, not whenever two edges cross. Imagine a freeway overpass – the freeway and side street cross, but it is not possible to change from the side street to the freeway at that point, so there is no intersection and no vertex would be placed.

Definition 12.5: Edges

Edges connect pairs of vertices. An edge can represent a physical connection between locations, like a street, or simply that a route connecting the two locations exists, like an airline flight.

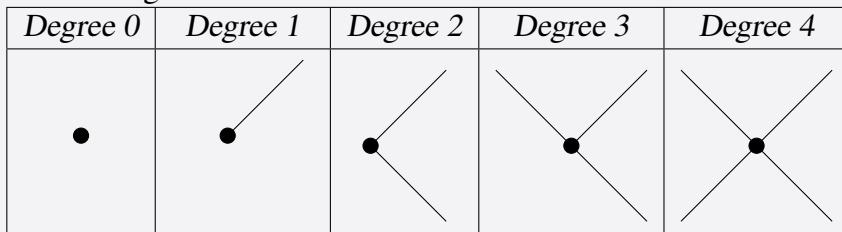
Definition 12.6: Loop

A loop is a special type of edge that connects a vertex to itself. Loops are not used much in street network graphs.

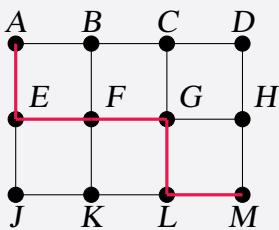


Definition 12.7: Degree of a vertex

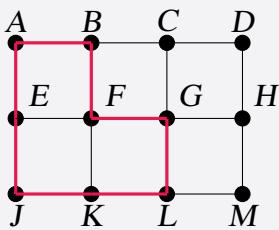
The degree of a vertex is the number of edges meeting at that vertex. It is possible for a vertex to have a degree of zero or larger.

**Definition 12.8: Path**

A path is a sequence of vertices using the edges. Usually we are interested in a path between two vertices. For example, a path from vertex A to vertex M is shown below. It is one of many possible paths in this graph.

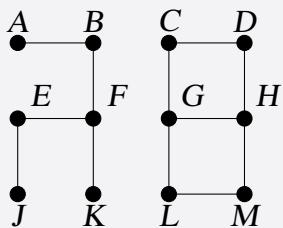
**Definition 12.9: Circuit (a.k.a. cycle)**

A circuit (a.k.a. cycle) is a path that begins and ends at the same vertex. A circuit (a.k.a. cycle) starting and ending at vertex A is shown below.



Definition 12.10: Connected

A graph is connected if there is a path from any vertex to any other vertex. Every graph drawn so far has been connected. The graph below is **disconnected**; there is no way to get from the vertices on the left to the vertices on the right.

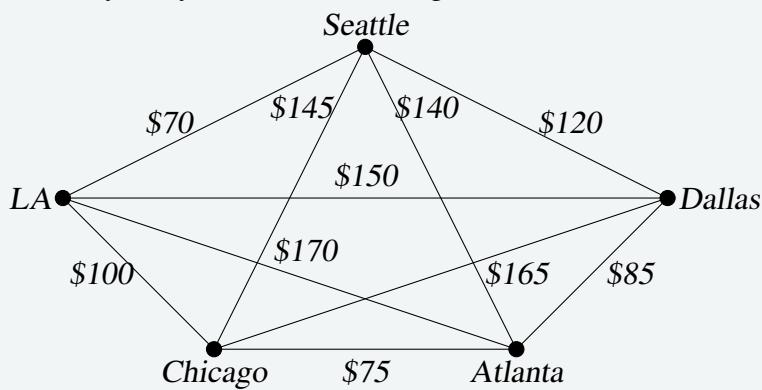
**Definition 12.11: Weights**

Depending upon the problem being solved, sometimes weights are assigned to the edges. The weights could represent the distance between two locations, the travel time, or the travel cost. It is important to note that the distance between vertices in a graph does not necessarily correspond to the weight of an edge.

Exercise 12.12

The graph below shows 5 cities. The weights on the edges represent the airfare for a one-way flight between the cities.

- How many vertices and edges does the graph have?
- Is the graph connected?
- What is the degree of the vertex representing LA?
- If you fly from Seattle to Dallas to Atlanta, is that a path or a circuit?
- If you fly from LA to Chicago to Dallas to LA, is that a path or a circuit?



12.4 Shortest Path

Outcomes

- *What is the problem statement?*
- *How to use Dijkstra's algorithm*
- *Software solutions*

Resources

- *How Dijkstra's algorithm works*
- *YouTube Video of Dijkstra's Algorithm*
- *Python Example using Networkx and also showing Dijkstra's algorithm*

When you visit a website like Google Maps or use your Smartphone to ask for directions from home to your Aunt's house in Pasadena, you are usually looking for a shortest path between the two locations. These computer applications use representations of the street maps as graphs, with estimated driving times as edge weights.

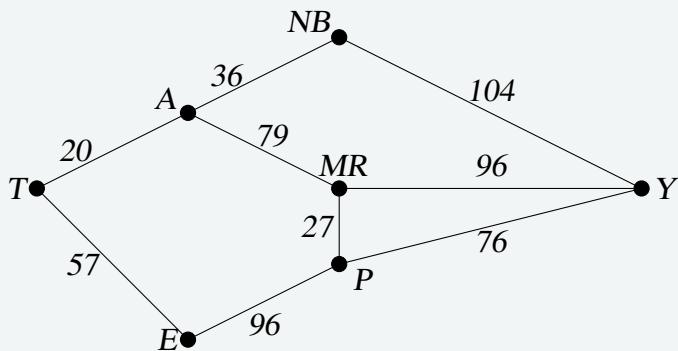
While often it is possible to find a shortest path on a small graph by guess-and-check, our goal in this chapter is to develop methods to solve complex problems in a systematic way by following **algorithms**. An algorithm is a step-by-step procedure for solving a problem. Dijkstra's (pronounced dike-strah) algorithm will find the shortest path between two vertices.

Dijkstra's Algorithm

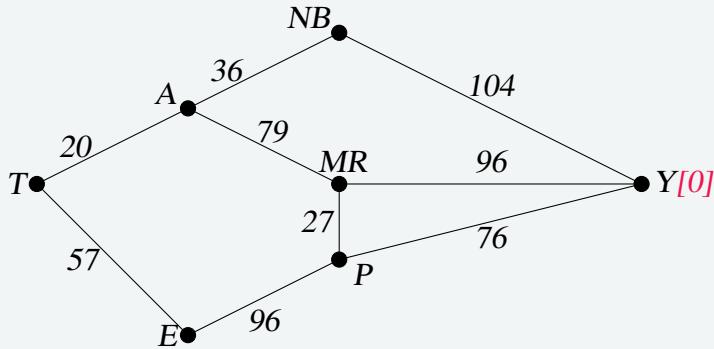
1. Mark the ending vertex with a distance of zero. Designate this vertex as current.
2. Find all vertices leading to the current vertex. Calculate their distances to the end. Since we already know the distance the current vertex is from the end, this will just require adding the most recent edge. Don't record this distance if it is longer than a previously recorded distance.
3. Mark the current vertex as visited. We will never look at this vertex again.
4. Mark the vertex with the smallest distance as current, and repeat from step 2.

Example 12.13

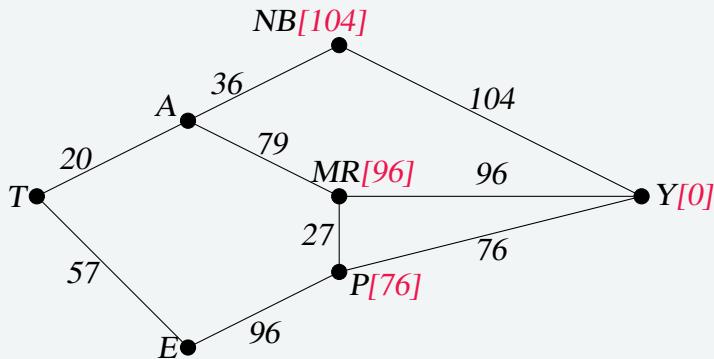
Suppose you need to travel from Yakima, WA (vertex Y) to Tacoma, WA (vertex T). Looking at a map, it looks like driving through Auburn (A) then Mount Rainier (MR) might be shortest, but it's not totally clear since that road is probably slower than taking the major highway through North Bend (NB). A graph with travel times in minutes is shown below. An alternate route through Eatonville (E) and Packwood (P) is also shown.



Step 1: Mark the ending vertex with a distance of zero. The distances will be recorded in [brackets] after the vertex name.



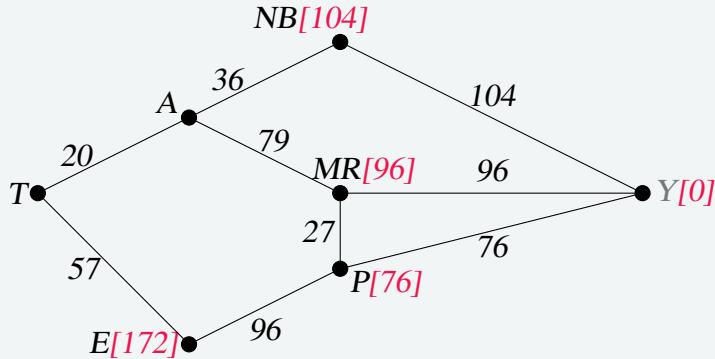
Step 2: For each vertex leading to Y, we calculate the distance to the end. For example, NB is a distance of 104 from the end, and MR is 96 from the end. Remember that distances in this case refer to the travel time in minutes.



Step 3 & 4: We mark Y as visited, and mark the vertex with the smallest recorded distance as current. At this point, P will be designated current. Back to step 2.

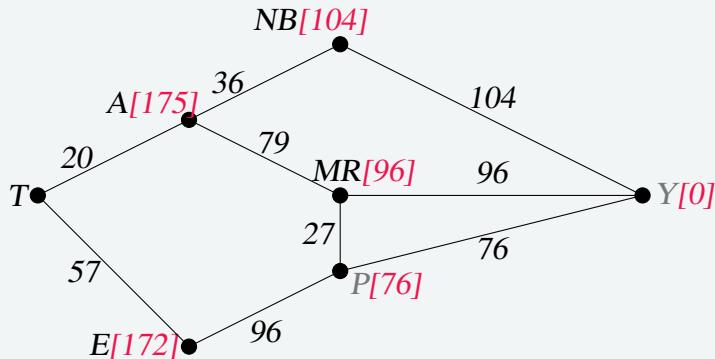
Step 2 (#2): For each vertex leading to P (and not leading to a visited vertex) we find the distance from the end. Since E is 96 minutes from P, and we've already calculated P is 76 minutes from Y, we can compute that E is $96 + 76 = 172$ minutes from Y.

If we make the same computation for MR, we'd calculate $76 + 27 = 103$. Since this is larger than the previously recorded distance from Y to MR, we will not replace it.



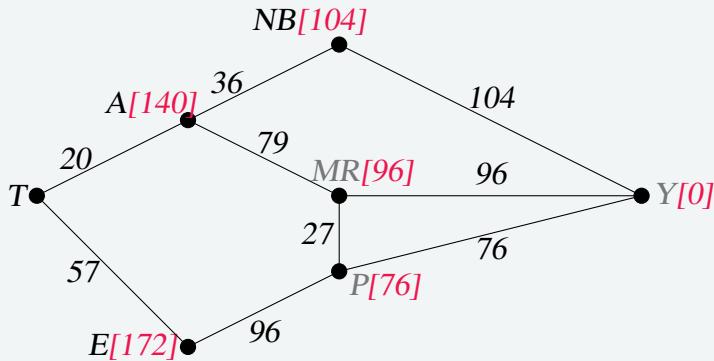
Step 3 & 4 (#2): We mark P as visited, and designate the vertex with the smallest recorded distance as current: MR. Back to step 2.

Step 2 (#3): For each vertex leading to MR (and not leading to a visited vertex) we find the distance to the end. The only vertex to be considered is A, since we've already visited Y and P. Adding MR's distance 96 to the length from A to MR gives the distance $96 + 79 = 175$ minutes from A to Y.



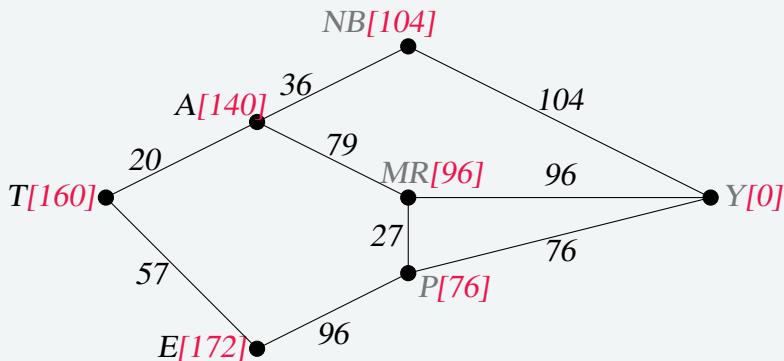
Step 3 & 4 (#3): We mark MR as visited, and designate the vertex with smallest recorded distance as current: NB. Back to step 2.

Step 2 (#4): For each vertex leading to NB, we find the distance to the end. We know the shortest distance from NB to Y is 104 and the distance from A to NB is 36, so the distance from A to Y through NB is $104 + 36 = 140$. Since this distance is shorter than the previously calculated distance from Y to A through MR, we replace it.



Step 3 & 4 (#4): We mark NB as visited, and designate A as current, since it now has the shortest distance.

Step 2 (#5): T is the only non-visited vertex leading to A, so we calculate the distance from T to Y through A: $20 + 140 = 160$ minutes.



Step 3 & 4 (#5): We mark A as visited, and designate E as current.

Step 2 (#6): The only non-visited vertex leading to E is T. Calculating the distance from T to Y through E, we compute $172 + 57 = 229$ minutes. Since this is longer than the existing marked time, we do not replace it.

Step 3 (#6): We mark E as visited. Since all vertices have been visited, we are done.

From this, we know that the shortest path from Yakima to Tacoma will take 160 minutes. Tracking which sequence of edges yielded 160 minutes, we see the shortest path is Y-NB-A-T.

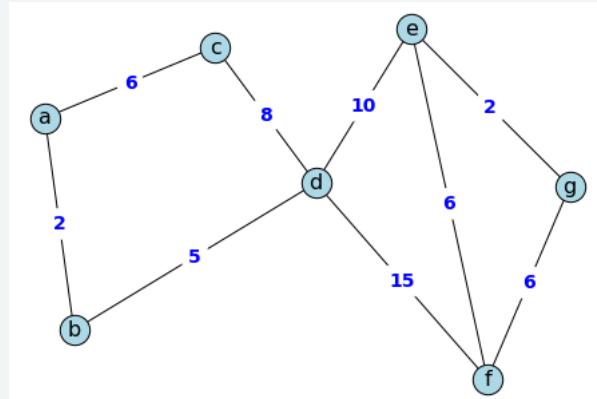
Dijkstra's algorithm is an **optimal algorithm**, meaning that it always produces the actual shortest path, not just a path that is pretty short, provided one exists. This algorithm is also **efficient**, meaning that it can be implemented in a reasonable amount of time. Dijkstra's algorithm takes around V^2 calculations, where V is the number of vertices in a graph¹. A graph with 100 vertices would take around 10,000 calculations. While that would be a lot to do by hand, it is not a lot for computer to handle. It is because of this efficiency that your car's GPS unit can compute driving directions in only a few seconds.

¹It can be made to run faster through various optimizations to the implementation.

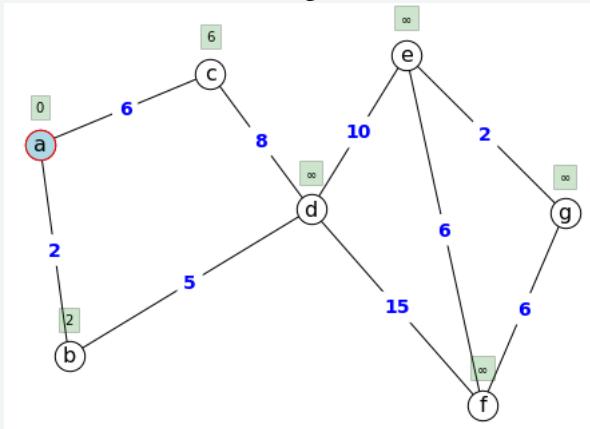
In contrast, an **inefficient** algorithm might try to list all possible paths then compute the length of each path. Trying to list all possible paths could easily take 10^{25} calculations to compute the shortest path with only 25 vertices; that's a 1 with 25 zeros after it! To put that in perspective, the fastest computer in the world would still spend over 1000 years analyzing all those paths.

Example 12.14: Dijkstra's algorithm example

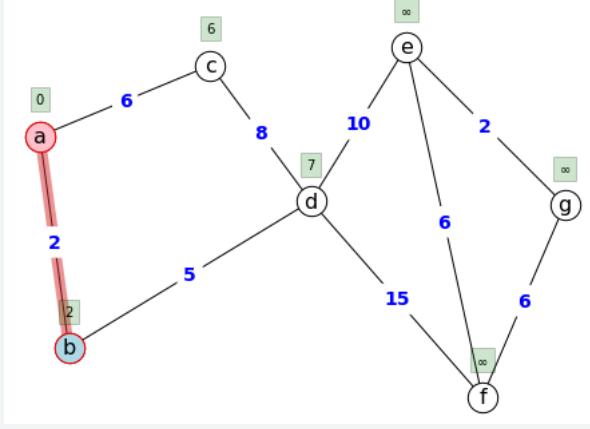
We would like to find a shortest path in the graph from node a to node g. See *Code for python code to solve this problem and create these graphics.*



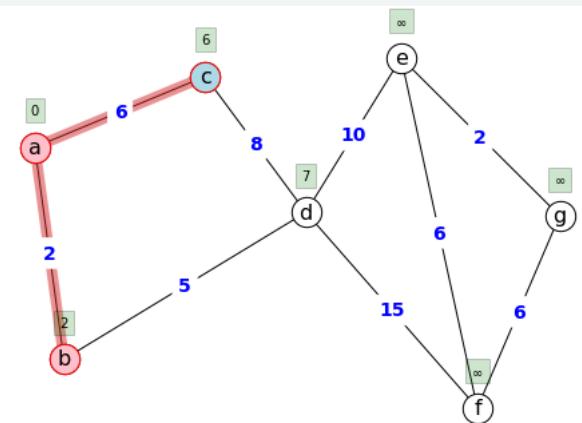
We will initialize our algorithm at node 'a'.



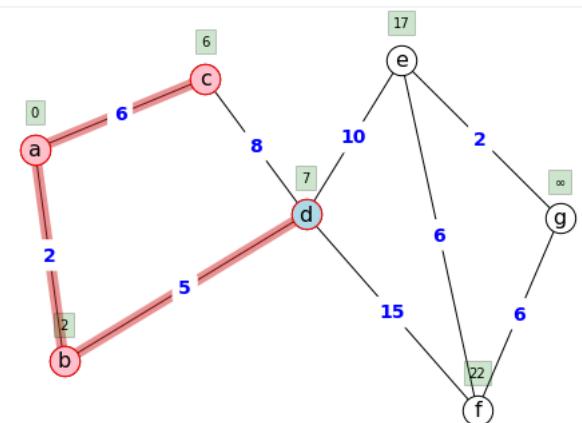
current	a	b	c	d	e	f	g
a	0	2	6	∞	∞	∞	∞



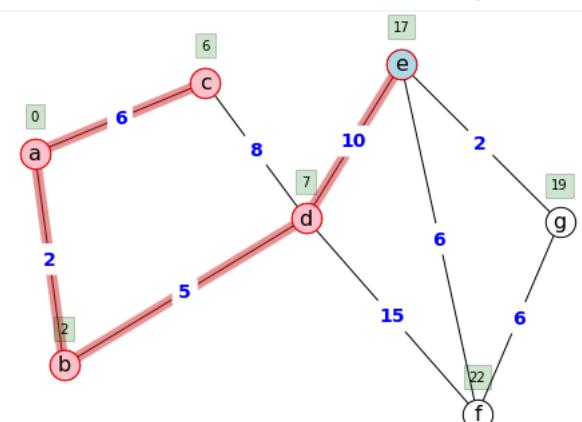
current	a	b	c	d	e	f	g
b	0	2	6	7	∞	∞	∞



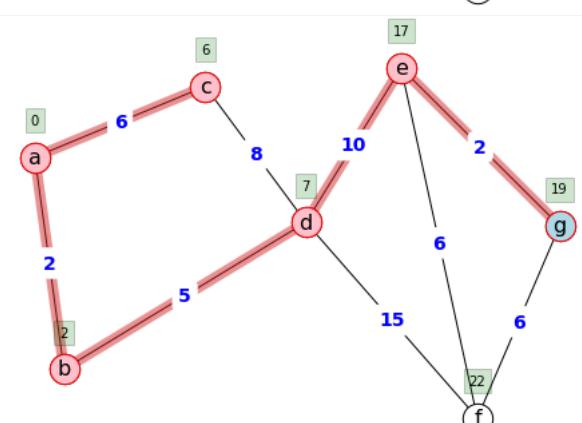
current	a	b	c	d	e	f	g
c	0	2	6	7	∞	∞	∞



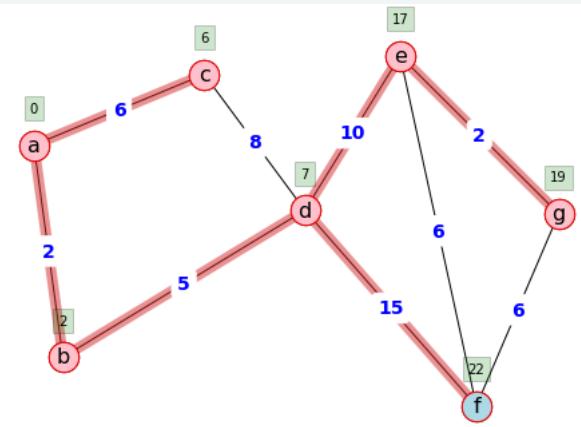
current	a	b	c	d	e	f	g
d	0	2	6	7	17	22	∞



current	a	b	c	d	e	f	g
e	0	2	6	7	17	22	19

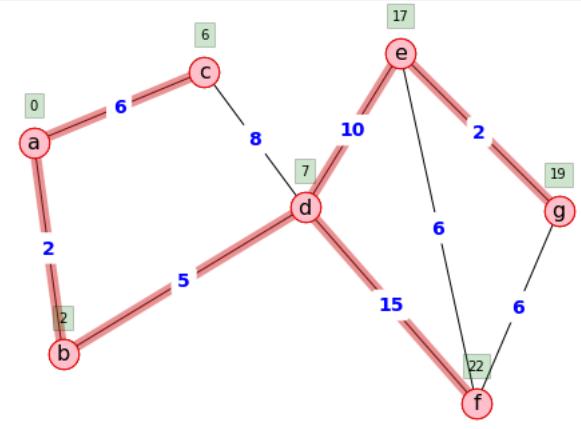


current	a	b	c	d	e	f	g
g	0	2	6	7	17	22	19



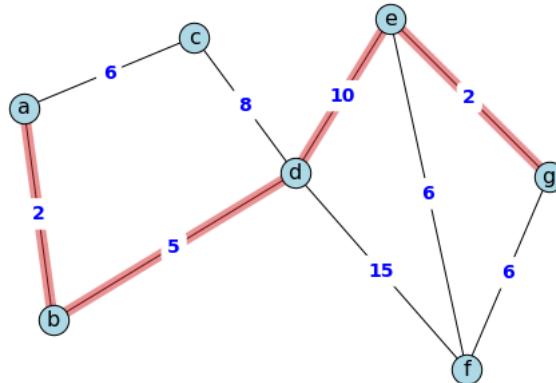
current	a	b	c	d	e	f	g
f	0	2	6	7	17	22	19

We can now summarize our calculations that followed Dijkstra's algorithm.



current	a	b	c	d	e	f	g
a	0	2	6	∞	∞	∞	∞
b	0	2	6	7	∞	∞	∞
c	0	2	6	7	∞	∞	∞
d	0	2	6	7	17	22	∞
e	0	2	6	7	17	22	19
g	0	2	6	7	17	22	19
f	0	2	6	7	17	22	19

FINAL SOLUTION The shortest path from a to g is the path a - b - d - e - g,



and has length

$$2 + 5 + 10 + 2 = 19.$$

Example 12.15

A shipping company needs to route a package from Washington, D.C. to San Diego, CA. To minimize costs, the package will first be sent to their processing center in Baltimore, MD then sent as part of mass shipments between their various processing centers, ending up in their processing center in Bakersfield, CA. From there it will be delivered in a small truck to San Diego.

The travel times, in hours, between their processing centers are shown in the table below. Three hours has been added to each travel time for processing. Find the shortest path from Baltimore to Bakersfield.

	Baltimore	Denver	Dallas	Chicago	Atlanta	Bakersfield
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	8	
Bakersfield		19	25			*

While we could draw a graph, we can also work directly from the table.

Step 1: The ending vertex, Bakersfield, is marked as current.

Step 2: All cities connected to Bakersfield, in this case Denver and Dallas, have their distances calculated; we'll mark those distances in the column headers.

Step 3 & 4: Mark Bakersfield as visited. Here, we are doing it by shading the corresponding row and column of the table. We mark Denver as current, shown in bold, since it is the vertex with the shortest distance.

	Baltimore	Denver [19]	Dallas [25]	Chicago	Atlanta	Bakersfield [0]
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	8	
Bakersfield		19	25			*

Step 2 (#2): For cities connected to Denver, calculate distance to the end. For example, Chicago is 18 hours from Denver, and Denver is 19 hours from the end, the distance for Chicago to the end is $18 + 19 = 37$ (Chicago to Denver to Bakersfield). Atlanta is 24 hours from Denver, so the distance to the end is $24 + 19 = 43$ (Atlanta to Denver to Bakersfield).

Step 3 & 4 (#2): We mark Denver as visited and mark Dallas as current.

	Baltimore	Denver [19]	Dallas [25]	Chicago [37]	Atlanta [43]	Bakersfield [0]
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	8	
Bakersfield		19	25			*

Step 2 (#3): For cities connected to Dallas, calculate the distance to the end. For Chicago, the distance from Chicago to Dallas is 18 and from Dallas to the end is 25, so the distance from Chicago to the end through Dallas would be $18 + 25 = 43$. Since this is longer than the currently marked distance for Chicago, we do not replace it. For Atlanta, we calculate $15 + 25 = 40$. Since this is shorter than the currently marked distance for Atlanta, we replace the existing distance.

Step 3 & 4 (#3): We mark Dallas as visited, and mark Chicago as current.

	Baltimore	Denver [19]	Dallas [25]	Chicago [37]	Atlanta [40]	Bakersfield [0]
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	8	
Bakersfield		19	25			*

Step 2 (#4): Baltimore and Atlanta are the only non-visited cities connected to Chicago. For Baltimore, we calculate $15 + 37 = 52$ and mark that distance. For Atlanta, we calculate $14 + 37 = 51$. Since this is longer than the existing distance of 40 for Atlanta, we do not replace that distance.

Step 3 & 4 (#4): Mark Chicago as visited and Atlanta as current.

	Baltimore [52]	Denver [19]	Dallas [25]	Chicago [37]	Atlanta [40]	Bakersfield [0]
Baltimore	*			15	14	
Denver		*		18	24	19
Dallas			*	18	15	25
Chicago	15	18	18	*	14	
Atlanta	14	24	15	14	8	
Bakersfield		19	25			*

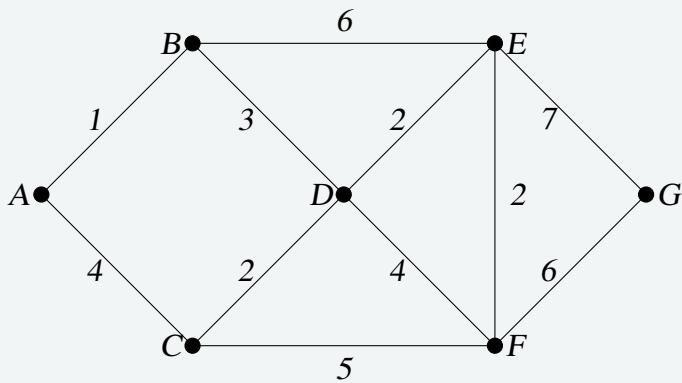
Step 2 (#5): The distance from Atlanta to Baltimore is 14. Adding that to the distance already calculated for Atlanta gives a total distance of $14 + 40 = 54$ hours from Baltimore to Bakersfield through Atlanta. Since this is larger than the currently calculated distance, we do not replace the distance for Baltimore.

Step 3 & 4 (#5): We mark Atlanta as visited. All cities have been visited and we are done.

The shortest route from Baltimore to Bakersfield will take 52 hours, and will route through Chicago and Denver.

Exercise 12.16

Find the shortest path between vertices A and G in the graph below.



12.5 Spanning Trees

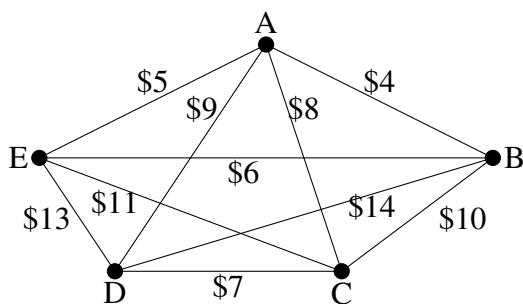
Outcomes

- Find the smallest set of edges that connects a graph

Resources

- YouTube Video: Kruskal's algorithm to find a minimum weight spanning tree

A company requires reliable internet and phone connectivity between their five offices (named A, B, C, D, and E for simplicity) in New York, so they decide to lease dedicated lines from the phone company. The phone company will charge for each link made. The costs, in thousands of dollars per year, are shown in the graph.

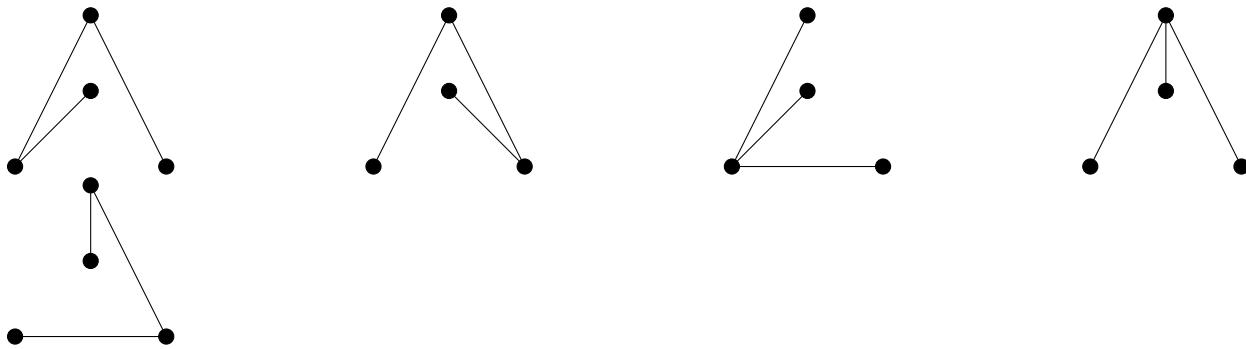


In this case, we don't need to find a circuit, or even a specific path; all we need to do is make sure we can make a call from any office to any other. In other words, we need to be sure there is a path from any vertex to any other vertex.

Definition 12.17: Spanning Tree

A spanning tree is a connected graph using all vertices in which there are no circuits. In other words, there is a path from any vertex to any other vertex, but no circuits.

Some examples of spanning trees are shown below. Notice there are no circuits in the trees, and it is fine to have vertices with degree higher than two.



Usually we have a starting graph to work from, like in the phone example above. In this case, we form our spanning tree by finding a **subgraph** – a new graph formed using all the vertices but only some of the edges from the original graph. No edges will be created where they didn't already exist.

Of course, any random spanning tree isn't really what we want. We want the **minimum cost spanning tree (MCST)**.

Definition 12.18: Minimum Cost Spanning Tree (MCST)

The minimum cost spanning tree is the spanning tree with the smallest total edge weight.

A nearest neighbor style approach doesn't make as much sense here since we don't need a circuit, so instead we will take an approach similar to sorted edges.

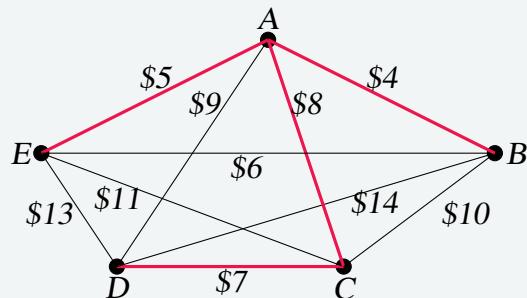
Kruskal's Algorithm

1. Select the cheapest unused edge in the graph.
2. Repeat step 1, adding the cheapest unused edge, unless:
 - adding the edge would create a circuit.
3. Repeat until a spanning tree is formed.

Example 12.19

Using our phone line graph from above, begin adding edges:

AB	\$4	OK
AE	\$5	OK
BE	\$6	reject – closes circuit ABEA
DC	\$7	OK
AC	\$8	OK



At this point we stop – every vertex is now connected, so we have formed a spanning tree with cost \$24 thousand a year.

Remarkably, Kruskal's algorithm is both optimal and efficient; we are guaranteed to always produce the optimal MCST.

Example 12.20

The power company needs to lay updated distribution lines connecting the ten Oregon cities below to the power grid. How can they minimize the amount of new line to lay?

	Ashland	Astoria	Bend	Corvallis	Crater Lake	Eugene	Newport	Portland	Salem	Seaside
Ashland	-	374	200	223	108	178	252	285	240	356
Astoria	374	-	255	166	433	199	135	95	136	17
Bend	200	255	-	128	277	128	180	160	131	247
Corvalis	223	166	128	-	430	47	52	84	40	155
Crater Lake	108	433	277	430	-	453	478	344	389	423
Eugene	178	199	128	47	453	-	91	110	64	181
Newport	252	135	180	52	478	91	-	114	83	117
Portland	285	95	160	84	344	110	114	-	47	78
Salem	240	136	131	40	389	64	83	47	-	118
Seaside	356	17	247	155	423	181	117	78	118	-

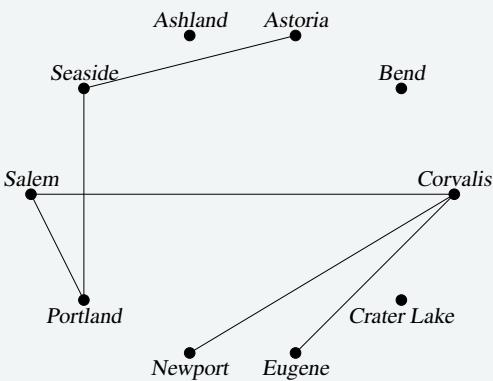
Using Kruskal's algorithm, we add edges from cheapest to most expensive, rejecting any that close a circuit. We stop when the graph is connected.

Seaside to Astoria	17 miles
Corvallis to Salem	40 miles
Portland to Salem	47 miles
Corvallis to Eugene	47 miles
Corvallis to Newport	52 miles
Salem to Eugene	reject – closes circuit
Portland to Seaside	78 miles

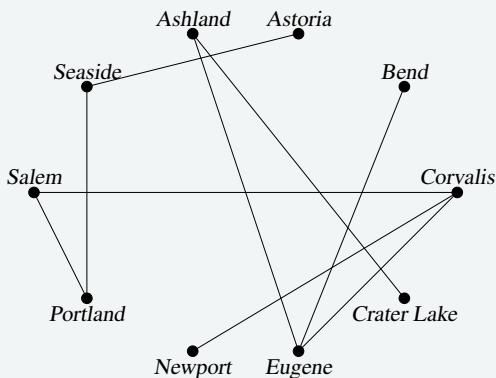
The graph up to this point is shown to the right.

Continuing,

Newport to Salem	reject
Corvallis to Portland	reject
Eugene to Newport	reject
Portland to Astoria	reject
Ashland to Crater Lake	108 miles
Eugene to Portland	reject
Newport to Portland	reject
Newport to Seaside	reject
Salem to Seaside	reject
Bend to Eugene	128 miles
Bend to Salem	reject
Astoria to Newport	reject
Salem to Astoria	reject
Corvallis to Seaside	reject
Portland to Bend	reject
Astoria to Corvallis	reject
Eugene to Ashland	178 miles

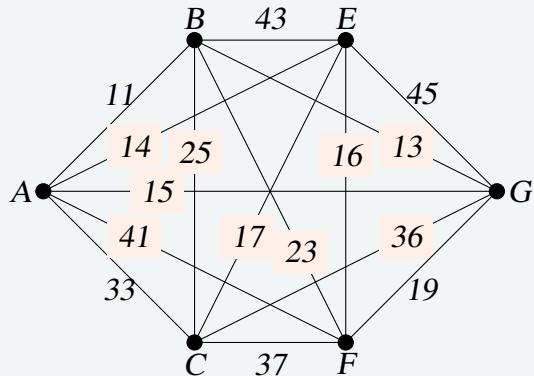


This connects the graph. The total length of cable to lay would be 695 miles.



Exercise 12.21: Min Cost Spanning Tree

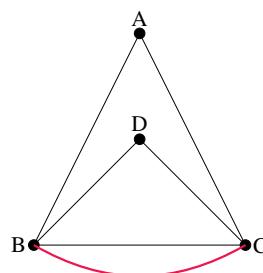
Find a minimum cost spanning tree on the graph below using Kruskal's algorithm.



12.6 Exercise Answers

1. (a) 5 vertices, 10 edges
 (b) Yes, it is connected.
 (c) The vertex is degree 4.
 (d) A path
 (e) A circuit
2. The shortest path is ABDEG, with length 13.
3. Yes, all vertices have even degree so this graph has an Euler Circuit. There are several possibilities. One is: ABEGFCDDFEDBCA
- 4.

This graph can be eulerized by duplicating the edge BC, as shown. One possible Euler circuit on the eulerized graph is ACDBCBA.



5. At each step, we look for the nearest location we haven't already visited. From B the nearest computer is E with time 24.

From E, the nearest computer is D with time 11.

From D the nearest is A with time 12.

From A the nearest is C with time 34.

From C, the only computer we haven't visited is F with time 27.

From F, we return back to B with time 50.

The NNA circuit from B is BEDACFB with time 158 milliseconds.

Using NNA again from other starting vertices:

Starting at A: ADEBCFA: time 146

Starting at C: CDEBAFC: time 167

Starting at D: DEBCFAD: time 146

Starting at E: EDACFBE: time 158

Starting at F: FDEBCAF: time 158

The RNN found a circuit with time 146 milliseconds: ADEBCFA. We could also write this same circuit starting at B if we wanted: BCFADEB or BEDAFCB.

6.

AB: Add, cost 11

BG: Add, cost 13

AE: Add, cost 14

EF: Add, cost 15

EC: Skip (degree 3 at E)

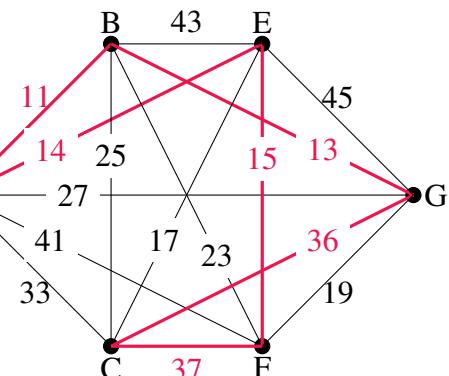
FG: Skip (would create a circuit not including C)

BF, BC, AG, AC: Skip (would cause a vertex to have degree 3)

GC: Add, cost 36

CF: Add, cost 37, completes the circuit

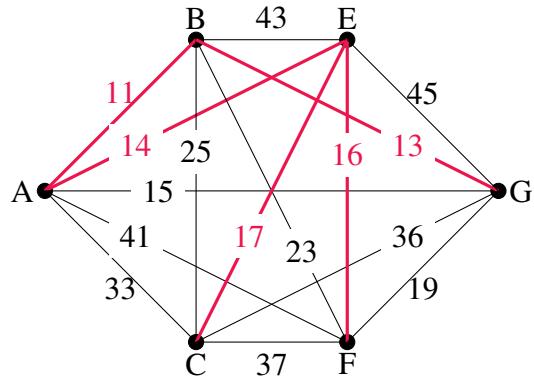
Final circuit: ABGCFEA



7. (??)

AB: Add, cost 11
 BG: Add, cost 13
 AE: Add, cost 14
 AG: Skip, would create circuit ABGA
 EF: Add, cost 16
 EC: Add, cost 17

This completes the spanning tree.



12.7 Prim's Algorithm

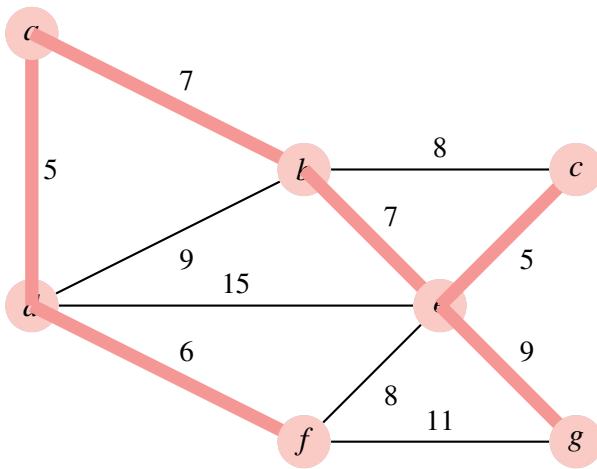
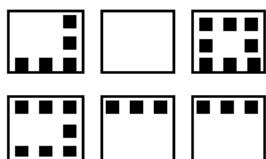


Figure 12.1: Prim's algorithm

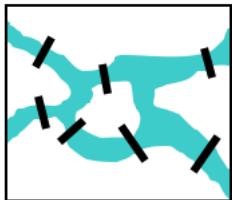
12.8 Additional Exercises

Skills

- To deliver mail in a particular neighborhood, the postal carrier needs to walk along each of the streets with houses (the dots). Create a graph with edges showing where the carrier must walk to deliver the mail.



2. Suppose that a town has 7 bridges as pictured below. Create a graph that could be used to determine if there is a path that crosses all bridges once.



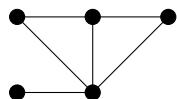
3. The table below shows approximate driving times (in minutes, without traffic) between five cities in the Dallas area. Create a weighted graph representing this data.

	Plano	Mesquite	Arlington	Denton
Fort Worth	54	52	19	42
Plano		38	53	41
Mesquite			43	56
Arlington				50

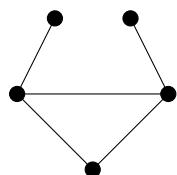
4. Shown in the table below are the one-way airfares between 5 cities². Create a graph showing this data.

	Honolulu	London	Moscow	Cairo
Seattle	\$159	\$370	\$654	\$684
Honolulu		\$830	\$854	\$801
London			\$245	\$323
Moscow				\$329

5. Find the degree of each vertex in the graph below.

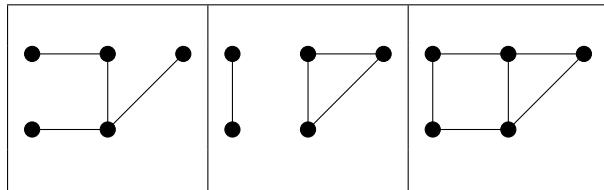


6. Find the degree of each vertex in the graph below.

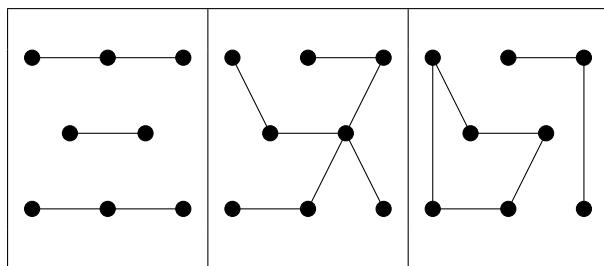


²Cheapest fares found when retrieved Sept. 1, 2009 for travel Sept. 22, 2009

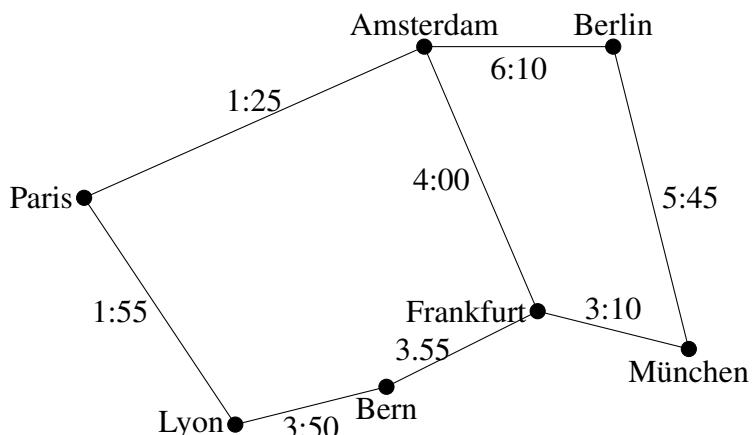
7. Which of these graphs are connected?



8. Which of these graphs are connected?

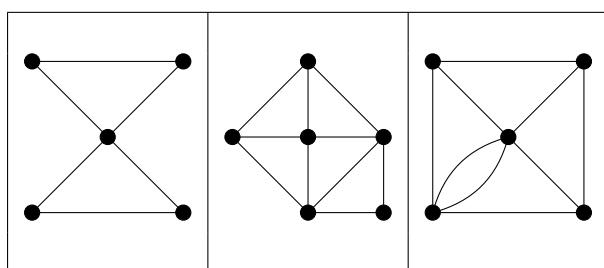


9. Travel times by rail for a segment of the Eurail system is shown below with travel times in hours and minutes. Find path with shortest travel time from Bern to Berlin by applying Dijkstra's algorithm.

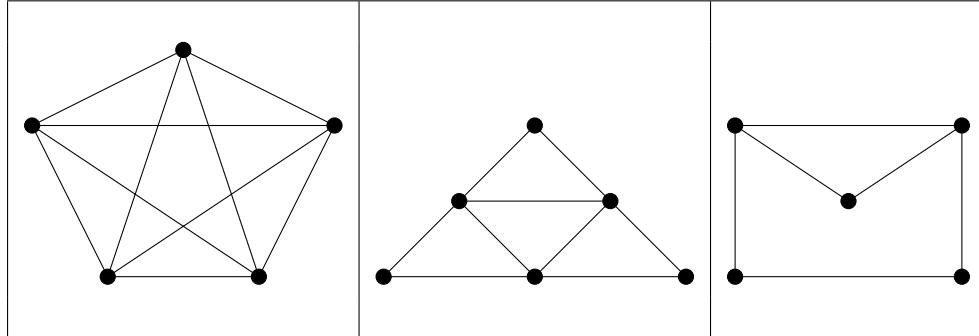


10. Using the graph from the previous problem, find the path with shortest travel time from Paris to München.

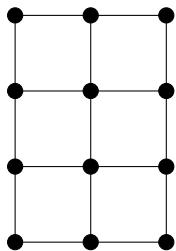
11. Does each of these graphs have an Euler circuit? If so, find it.



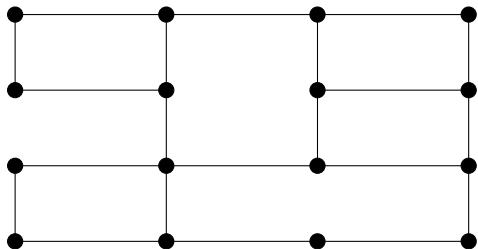
12. Does each of these graphs have an Euler circuit? If so, find it.



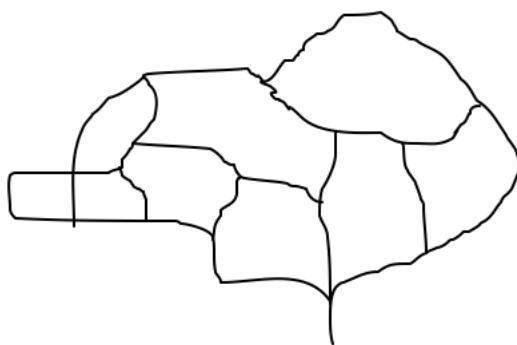
13. Eulerize this graph using as few edge duplications as possible. Then, find an Euler circuit.



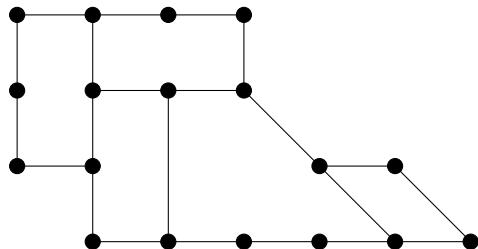
14. Eulerize this graph using as few edge duplications as possible. Then, find an Euler circuit.



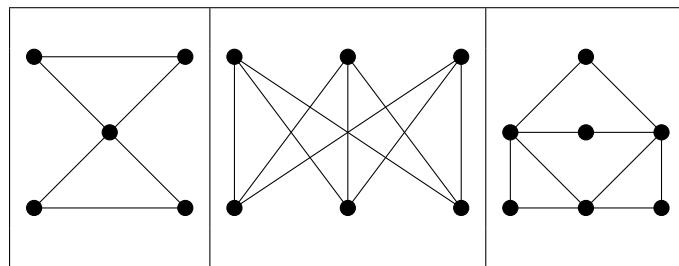
15. The maintenance staff at an amusement park need to patrol the major walkways, shown in the graph below, collecting litter. Find an efficient patrol route by finding an Euler circuit. If necessary, eulerize the graph in an efficient way.



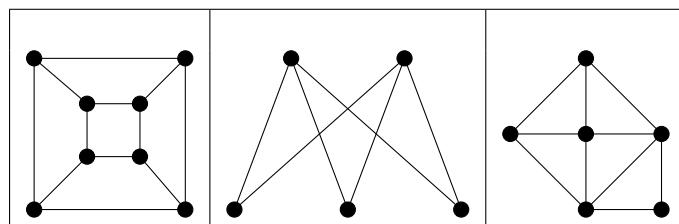
16. After a storm, the city crew inspects for trees or brush blocking the road. Find an efficient route for the neighborhood below by finding an Euler circuit. If necessary, eulerize the graph in an efficient way.



17. Does each of these graphs have at least one Hamiltonian circuit? If so, find one.



18. Does each of these graphs have at least one Hamiltonian circuit? If so, find one.



19. A company needs to deliver product to each of their 5 stores around the Dallas, TX area. Driving distances between the stores are shown below. Find a route for the driver to follow, returning to the distribution center in Fort Worth:

- (a) Using Nearest Neighbor starting in Fort Worth
- (b) Using Repeated Nearest Neighbor
- (c) Using Sorted Edges

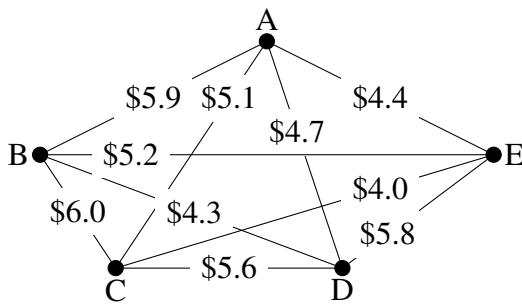
	Plano	Mesquite	Arlington	Denton
Fort Worth	54	52	19	42
Plano		38	53	41
Mesquite			43	56
Arlington				50

20. A salesperson needs to travel from Seattle to Honolulu, London, Moscow, and Cairo. Use the table of flight costs from problem #4 to find a route for this person to follow:

- (a) Using Nearest Neighbor starting in Seattle
- (b) Using Repeated Nearest Neighbor
- (c) Using Sorted Edges

21. When installing fiber optics, some companies will install a sonet ring; a full loop of cable connecting multiple locations. This is used so that if any part of the cable is damaged it does not interrupt service, since there is a second connection to the hub. A company has 5 buildings. Costs (in thousands of dollars) to lay cables between pairs of buildings are shown below. Find the circuit that will minimize cost:

- (a) Using Nearest Neighbor starting at building A
- (b) Using Repeated Nearest Neighbor
- (c) Using Sorted Edges



22. A tourist wants to visit 7 cities in Israel. Driving distances, in kilometers, between the cities are shown below³. Find a route for the person to follow, returning to the starting city:

- (a) Using Nearest Neighbor starting in Jerusalem
- (b) Using Repeated Nearest Neighbor
- (c) Using Sorted Edges

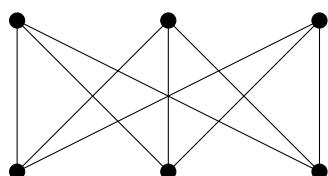
³From <http://www.ddtravel-acc.com/Israel-cities-distance.htm>

	Jerusalem	Tel Aviv	Haifa	Tiberias	Beer Sheba	Eilat
Jerusalem	—					
Tel Aviv	58	—				
Haifa	151	95	—			
Tiberias	152	134	69	—		
Beer Sheba	81	105	197	233	—	
Eilat	309	346	438	405	241	—
Nazareth	131	102	35	29	207	488

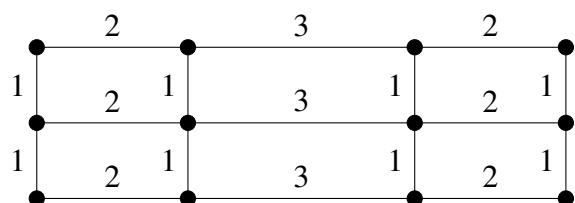
23. Find a minimum cost spanning tree for the graph you created in problem #3.
24. Find a minimum cost spanning tree for the graph you created in problem #22.
25. Find a minimum cost spanning tree for the graph from problem #21.

Concepts

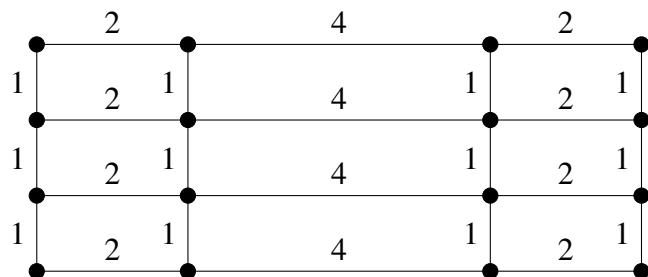
26. Can a graph have one vertex with odd degree? If not, are there other values that are not possible? Why?
27. A complete graph is one in which there is an edge connecting every vertex to every other vertex. For what values of n does complete graph with n vertices have an Euler circuit? A Hamiltonian circuit?
28. Create a graph by drawing n vertices in a row, then another n vertices below those. Draw an edge from each vertex in the top row to every vertex in the bottom row. An example when $n = 3$ is shown below. For what values of n will a graph created this way have an Euler circuit? A Hamiltonian circuit?



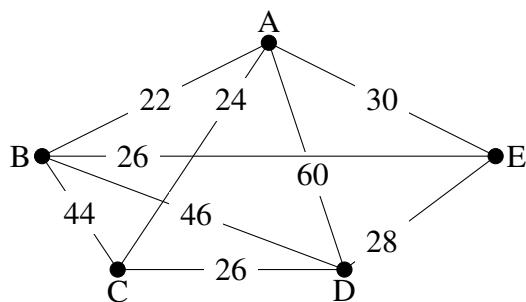
29. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



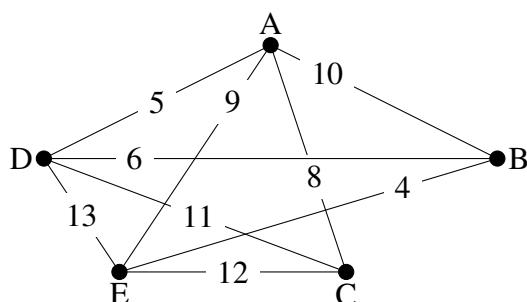
30. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



31. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



32. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



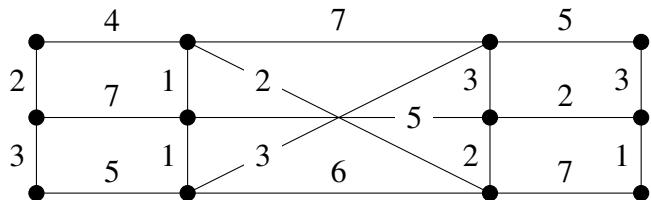
Explorations

33. Social networks such as Facebook and LinkedIn can be represented using graphs in which vertices represent people and edges are drawn between two vertices when those people are “friends.” The table below shows a friendship table, where an X shows that two people are friends.

	A	B	C	D	E	F	G	H	I
A	X	X			X	X			
B		X		X					
C			X		X				
D				X				X	
E					X		X		
F						X		X	
G							X		
H								X	

- (a) Create a graph of this friendship table
- (b) Find the shortest path from A to D. The length of this path is often called the “degrees of separation” of the two people.
- (c) Extension: Split into groups. Each group will pick 10 or more movies, and look up their major actors (www.imdb.com is a good source). Create a graph with each actor as a vertex, and edges connecting two actors in the same movie (note the movie name on the edge). Find interesting paths between actors, and quiz the other groups to see if they can guess the connections.
34. A spell checker in a word processing program makes suggestions when it finds a word not in the dictionary. To determine what words to suggest, it tries to find similar words. One measure of word similarity is the Levenshtein distance, which measures the number of substitutions, additions, or deletions that are required to change one word into another. For example, the words spit and spot are a distance of 1 apart; changing spit to spot requires one substitution (i for o). Likewise, spit is distance 1 from pit since the change requires one deletion (the s). The word spite is also distance 1 from spit since it requires one addition (the e). The word soot is distance 2 from spit since two substitutions would be required.
- (a) Create a graph using words as vertices, and edges connecting words with a Levenshtein distance of 1. Use the misspelled word “moke” as the center, and try to find at least 10 connected dictionary words. How might a spell checker use this graph?
- (b) Improve the method from above by assigning a weight to each edge based on the likelihood of making the substitution, addition, or deletion. You can base the weights on any reasonable approach: proximity of keys on a keyboard, common language errors, etc. Use Dijkstra’s algorithm to find the length of the shortest path from each word to “moke”. How might a spell checker use these values?
35. The graph below contains two vertices of odd degree. To eulerize this graph, it is necessary to duplicate edges connecting those two vertices.
- (a) Use Dijkstra’s algorithm to find the shortest path between the two vertices with odd degree. Does this produce the most efficient eulerization and solve the Chinese Postman Problem for

this graph?



- (b) Suppose a graph has n odd vertices. Using the approach from part a, how many shortest paths would need to be considered? Is this approach going to be efficient?

12.8.1. Notes

A paper entitled 'A Note on Two Problems in Connexion with Graphs' was published in the journal 'Numerische Mathematik' in 1959. It was in this paper where the computer scientist named Edsger W. Dijkstra proposed the Dijkstra's Algorithm for the shortest path problem; a fundamental graph theoretic problem. This algorithm can be used to find the shortest path between two nodes or a more common variant of this algorithm is to find the shortest path between a specific 'source' node to any other nodes in the network. <https://www.overleaf.com/project/62472837411e2ce1b881337f>

Notes, References, and Resources

Resources

Youtube! Video of many graph algorithms by Google engineer (6+ hours)

Part III

Integer Programming

13. 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.

13.1 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 13.1), but has applications in many contexts.

Binary Knapsack Problem:

NP-Complete

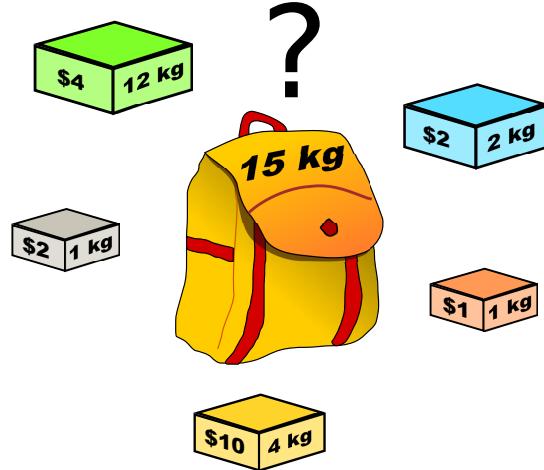
Given a non-negative weight vector $a \in \mathbb{Q}_+^n$, a capacity $b \in \mathbb{Q}_+$, and objective coefficients $c \in \mathbb{Q}^n$,

$$\begin{aligned} & \max c^\top x \\ & \text{s.t. } a^\top x \leq b \\ & \quad x \in \{0, 1\}^n \end{aligned} \tag{13.1}$$

Example: Knapsack

Gurobipy

¹wiki/File/knapsack, from wiki/File/knapsack. wiki/File/knapsack, wiki/File/knapsack.



© wiki/File/knapsack¹

Figure 13.1: 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 13.1.

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 $c \in \mathbb{Q}^n$,

$$\begin{aligned} & \max c^\top x \\ & \text{s.t. } a^\top x \leq b \\ & \quad x \in \mathbb{Z}_+^n \end{aligned} \tag{13.2}$$

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 $c \in \mathbb{Q}^n$,

$$\max c^\top x \tag{13.3}$$

$$\text{s.t. } a^\top x = b \tag{13.4}$$

$$x \in \mathbb{Z}_+^n \tag{13.5}$$

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

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

13.2 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:

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

13.3 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

$$\begin{aligned} \min \quad & 1^\top x \\ \text{s.t.} \quad & \sum_{v \in V_i} x_v \geq 1 \text{ for all } i = 1, \dots, l \\ & x_v \in \{0, 1\} \text{ for all } v \in V \end{aligned} \tag{13.1}$$

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:

$$\begin{aligned}
 \min \quad & 1^\top x \\
 \text{s.t.} \quad & x_u + x_v \geq 1 \text{ for all } (u, v) \in E \\
 & x_v \in \{0, 1\} \text{ for all } v \in V.
 \end{aligned} \tag{13.2}$$

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 13.2 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:

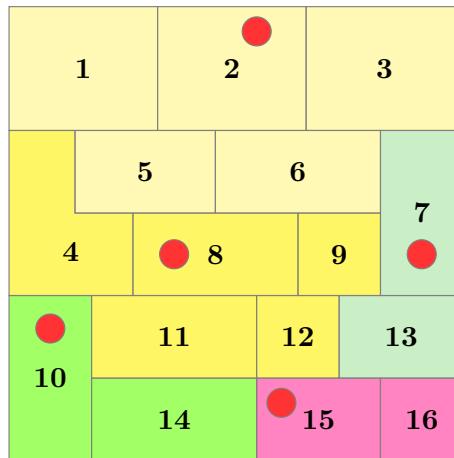
- Let V be the set of districts ($V = \{1, \dots, 16\}$)
- Let V_i be the set of districts that neighbor district i (e.g. $V_1 = \{2, 4, 5\}$).

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 13.2: 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

$$\begin{aligned}
 \max \quad & c^\top x \\
 \text{s.t..} \quad & Ax \geq 1 \\
 & x \in \{0, 1\}^n.
 \end{aligned} \tag{13.3}$$

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.

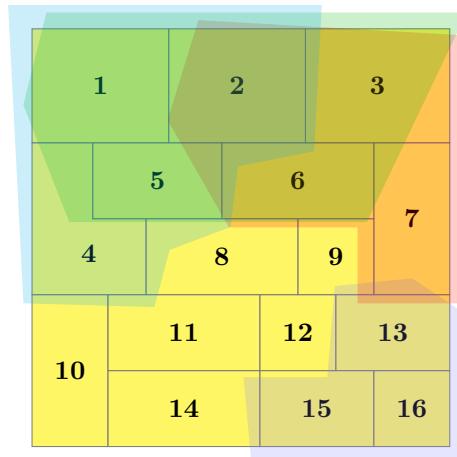
© tikz/Illustration2.pdf³

Figure 13.3: Set cover representation of fire station problem. For example, choosing district 16 to have a fire station covers districts 13, 15, and 16.

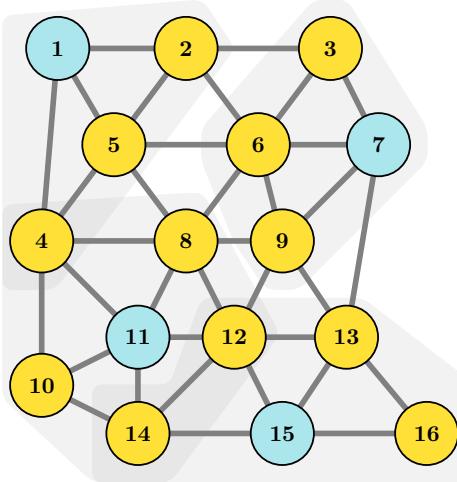
© tikz/Illustration3.pdf⁴

Figure 13.4: 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.

13.3.1. 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

$$\begin{aligned} & \max c^\top x \\ & \text{s.t.. } Ax \geq b \\ & \quad x \in \mathbb{Z}_+^n. \end{aligned} \tag{13.4}$$

13.4 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

Gurobipy

Sets:

- 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:

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

13.5 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*.

13.5.1. 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:

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

Alternative model!

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

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.

13.5.2. Uncapacitated Facility Location

Uncapacitated Facility Location:

NP-Complete

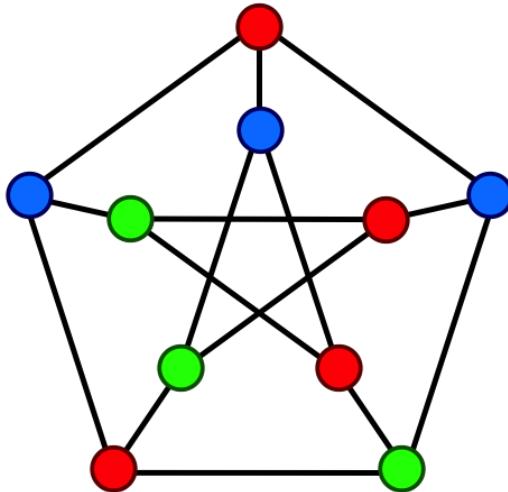
Given costs connections c_{ij} and fixed building costs f_i , the uncapacitated facility location problem is

$$\begin{aligned}
 \min \quad & \sum_{i=1}^n \sum_{j=1}^m c_{ij} z_{ij} + \sum_{i=1}^n f_i x_i \\
 \text{s.t.} \quad & \sum_{i=1}^n z_{ij} = 1 \text{ for all } j = 1, \dots, m \\
 & \sum_{j=1}^m z_{ij} \leq M x_i \text{ for all } i = 1, \dots, n \\
 & z_{ij} \in \{0, 1\} \text{ for all } i = 1, \dots, n \text{ and } j = 1, \dots, m \\
 & x_i \in \{0, 1\} \text{ for all } i = 1, \dots, n
 \end{aligned} \tag{13.1}$$

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!

$$\begin{aligned} \min \quad & \sum_{i=1}^n \sum_{j=1}^m c_{ij} z_{ij} + \sum_{i=1}^n f_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n z_{ij} = 1 \text{ for all } j = 1, \dots, m \\ & z_{ij} \leq x_i \text{ for all } i = 1, \dots, n \text{ for all } j = 1, \dots, m \\ & z_{ij} \in \{0, 1\} \text{ for all } i = 1, \dots, n \text{ and } j = 1, \dots, m \\ & x_i \in \{0, 1\} \text{ for all } i = 1, \dots, n \end{aligned} \tag{13.2}$$



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

Figure 13.5: wiki/File/Petersen-graph-3-coloring.png

13.6 Graph Coloring

Graph coloring Figure 13.5 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.

13.7 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.

$$x_i \leq x_j \quad (13.1)$$

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

$$x_1 + x_2 + \dots + x_5 \leq 5 \cdot x_6 \quad (13.2)$$

Alternatively!

$$x_i \leq x_6 \quad \text{for all } i = 1, \dots, 5 \quad (13.3)$$

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

$$x_i \leq x_j \quad (13.4)$$

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

$$x_1 + x_2 + \dots + x_i \leq i \cdot x_j \quad (13.5)$$

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

$$x_i \leq x_j \quad (13.6)$$

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

$$x_i + x_j = 1 \quad (13.7)$$

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

$$x_i + x_j \geq 1 \quad (13.8)$$

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

$$x_j \leq (1 - x_i) \quad (13.9)$$

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

$$x_i + x_j + x_k \leq 1 \quad (13.10)$$

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

$$x_i + x_j + x_k \leq 2 \quad (13.11)$$

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

$$x_i + x_j + x_k = 1 \quad (13.12)$$

These tricks can be connected to create different function values.

Example 13.1: 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 13.2: 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.

13.7.1. 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

$$x_A + x_B + x_C + x_D + x_E \leq 3 + M\delta. \quad (13.13)$$

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

$$\text{If } \delta = 0, \text{ then } a^\top x \leq b. \quad (13.14)$$

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

$$\begin{aligned} a^\top x - b &\leq M\delta \\ \delta &\in \{0, 1\} \end{aligned} \quad (13.15)$$

13.7.2. 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:

$$\text{Either } a^\top x \leq b \text{ or } c^\top x \leq d \text{ holds} \quad (13.16)$$

can be modeled as

$$\begin{aligned} a^\top x - b &\leq M_1 \delta \\ c^\top x - d &\leq M_2(1 - \delta) \\ \delta &\in \{0, 1\}, \end{aligned} \quad (13.17)$$

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 13.3

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

$$\begin{aligned} x - 2 &\leq 5\delta \\ y - 10 &\leq 20(1 - \delta) \\ \delta &\in \{0, 1\}, \end{aligned} \quad (13.18)$$

13.7.3. 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

$$\sum_{i \in I} x_i \geq 10 + 1 + M\delta. \quad (13.19)$$

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

$$\text{If } a^\top x \leq b \text{ then } \delta = 1 \quad (13.20)$$

can be modeled as

$$a^\top x - b \geq \varepsilon(1 - \delta) + m\delta. \quad (13.21)$$

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

$$\text{If } \delta = 0, \text{ then } a^\top x - b > 0. \quad (13.22)$$

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 13.1: 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 13.2: 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 $a^\top x \leq b + M\delta$.

13.7.4. 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 $i \in \{1, \dots, n\}$:

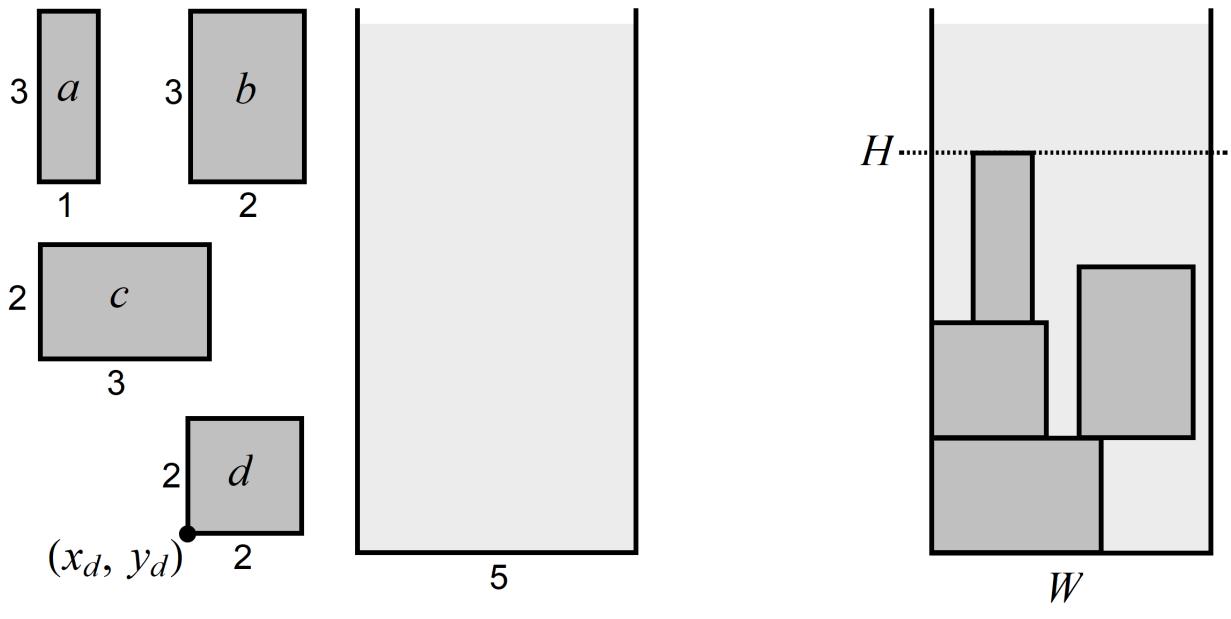
$$\begin{aligned} \mathbf{a}_1^\top \mathbf{x} &\leq b_1 + M(1 - \delta_1) \\ \mathbf{a}_2^\top \mathbf{x} &\leq b_2 + M(1 - \delta_2) \\ &\vdots \\ \mathbf{a}_n^\top \mathbf{x} &\leq b_n + M(1 - \delta_n) \\ \sum_{i=1}^n \delta_i &\geq k \end{aligned}$$

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.

13.7.4.1. 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 (13.23a) overall height is minimized, (13.23b) overall width is less than W , and (13.23c) none of the rectangles overlap. See Figure 13.6 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 (13.23c) 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 13.6: An Example of the Strip Packing Problem

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

$$\text{Minimize } \max_{i \in I} \{y_i + h_i\} \quad (13.23a)$$

$$\text{s.t. } \max_{i \in I} \{x_i + w_i\} \leq W \quad (13.23b)$$

$$\left. \begin{array}{l} x_i + w_i \leq x_j \\ x_j + w_j \leq x_i \\ y_i + h_i \leq y_j \\ y_j + h_j \leq y_i \end{array} \right\} \begin{array}{l} \text{At least one of these} \\ \text{for every distinct pairs} \\ \text{of rectangles } i, j \in I \end{array} \quad (13.23c)$$

$$x_i, y_i \geq 0 \quad \forall i \in I \quad (13.23d)$$

Constraint (13.23c) 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 (13.23c):

$$\begin{aligned} \text{Minimize} \quad & H \\ \text{s.t.} \quad & y_i + h_i \leq H \quad \forall i \in I \quad (13.23a) \\ & x_i + w_i \leq W \quad \forall i \in I \quad (13.23b) \\ & x_i + w_i \leq x_j + M(1 - \delta_{ij1}) \quad \forall i, j \in I: i > j \\ & x_j + w_j \leq x_i + M(1 - \delta_{ij2}) \quad \forall i, j \in I: i > j \\ & y_i + h_i \leq y_j + M(1 - \delta_{ij3}) \quad \forall i, j \in I: i > j \quad (13.23c) \\ & y_j + h_j \leq y_i + M(1 - \delta_{ij4}) \quad \forall i, j \in I: i > j \\ & \sum_{k=1}^4 \delta_{ijk} \geq 1 \quad \forall i, j \in I: i > j \\ & x_i, y_i \geq 0 \quad \forall i \in I \\ & \delta_{ij} \in \{0, 1\}^4 \quad \forall i, j \in I: i > j \end{aligned}$$

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 13.7; it was found via GurobiPy.

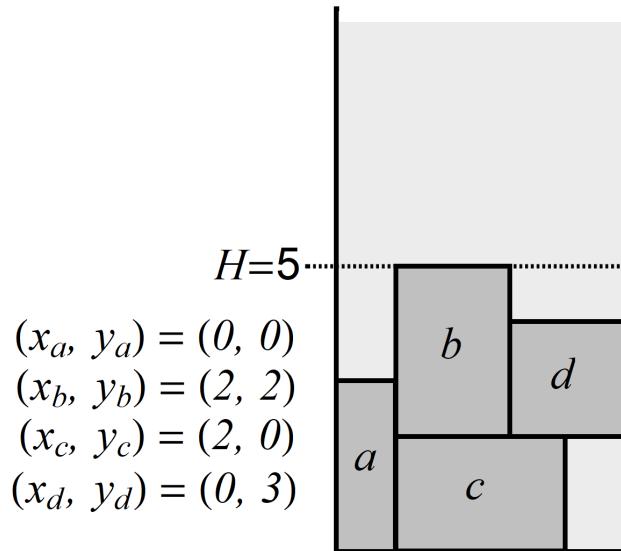


Figure 13.7: The Optimal Solution to the example problem

This problem is considered strongly NP-Hard.

13.7.5. SOS1 Constraints

Definition 13.4: 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:

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

13.7.6. SOS2 Constraints

Definition 13.5: 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.

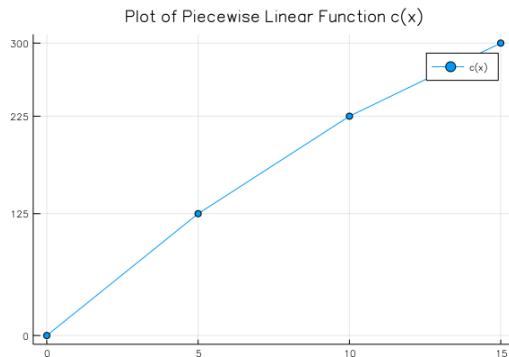
We next give an example of how to use binary variables to model this and then show how much simpler it can be coded using the SOS2 constraint.

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 13.8: scale = 0.35

13.7.7. 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 13.8 .

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 17:

$$\max z = 12x_{11} + 12x_{21} + 14x_{12} + 14x_{22} - c(x) \quad (13.24)$$

$$\text{s.t. } x_{11} + x_{12} \leq x + 5 \quad (13.25)$$

$$x_{21} + x_{22} \leq 10 \quad (13.26)$$

$$0.5x_{11} - 0.5x_{21} \geq 0 \quad (13.27)$$

$$0.4x_{12} - 0.6x_{22} \geq 0 \quad (13.28)$$

$$x_{ij} \geq 0 \quad (13.29)$$

$$0 \leq x \leq 15 \quad (13.30)$$

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

$$\begin{aligned}
 & \max \quad 12X_{1,1} + 12X_{2,1} + 14X_{1,2} + 14X_{2,2} - y \\
 \text{Subject to} \quad & x - 5z_2 - 10z_3 - 15z_4 = 0 \\
 & y - 125z_2 - 225z_3 - 300z_4 = 0 \\
 & z_1 + z_2 + z_3 + z_4 = 1 \\
 & X_{1,1} + X_{1,2} - x \leq 5 \\
 & X_{2,1} + X_{2,2} \leq 10 \\
 & 0.5X_{1,1} - 0.5X_{2,1} \geq 0 \\
 & 0.4X_{1,2} - 0.6X_{2,2} \geq 0 \\
 & \text{SOS2 : } \{z_1, z_2, z_3, z_4\} \\
 & \quad \begin{array}{ll} X_{i,j} \geq 0 & \forall i \in \{1, 2\}, j \in \{1, 2\} \\ 0 \leq z_i \leq 1 & \forall i \in \{1, 2, 3, 4\} \\ 0 \leq x \leq 15 & \\ y \text{ free} & \end{array} \\
 \end{aligned} \quad (13.31)$$

13.7.7.1. 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.

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

13.7.8. 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

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

13.7.9. 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,

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

13.7.10. 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.

13.7.10.1. Exact 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:

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

13.7.10.2. 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:

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

13.8 Network Flow

13.8.1. 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.

13.8.2. 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:

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

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.

13.8.3. 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:

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

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.

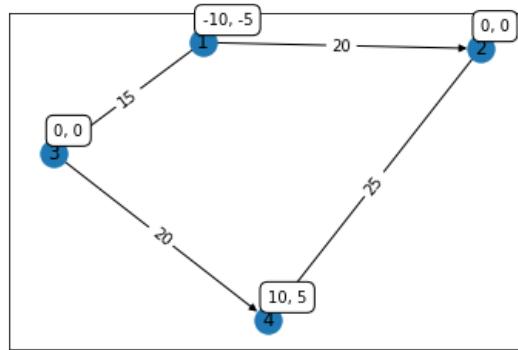
- Flow Conservation: The second set of constraints ensures that the flow into each node minus the flow out of the node is equal to the demand of each commodity at the node.

- Integrality Constraints: The third set of constraints ensures that the flow on each arc for each commodity is an integer.

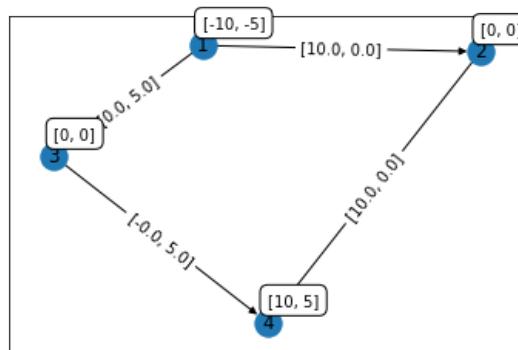
Example: Multi-commodity network flow

Gurobipy

```
# Sample data
nodes = [1, 2, 3, 4]
arcs = [(1, 2), (1, 3), (2, 4), (3, 4)]
```



© multi-network-flow-data.png⁷
Figure 13.9: multi-network-flow-data.png



© multi-network-flow-solution.png⁸
Figure 13.10: multi-network-flow-solution.png

```

commodities = [1, 2]
capacities = {(1, 2): 20, (1, 3): 15, (2, 4): 25, (3, 4): 20}
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.

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

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

13.9 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.

13.9.1. 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:

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

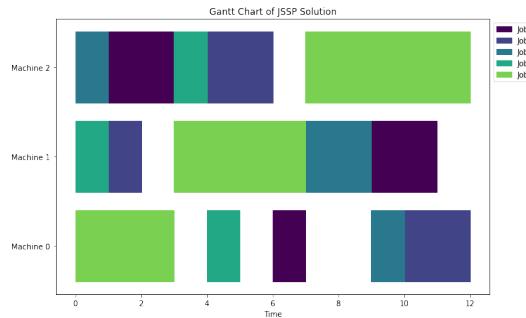
Here, the sequence of machines for each job is given by the rows of the matrix. For instance, job 1 is processed first on machine 2, then on machine 0, and finally on machine 1.

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 13.11

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



© jssp.png⁹
Figure 13.11: 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.

13.9.2. 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

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

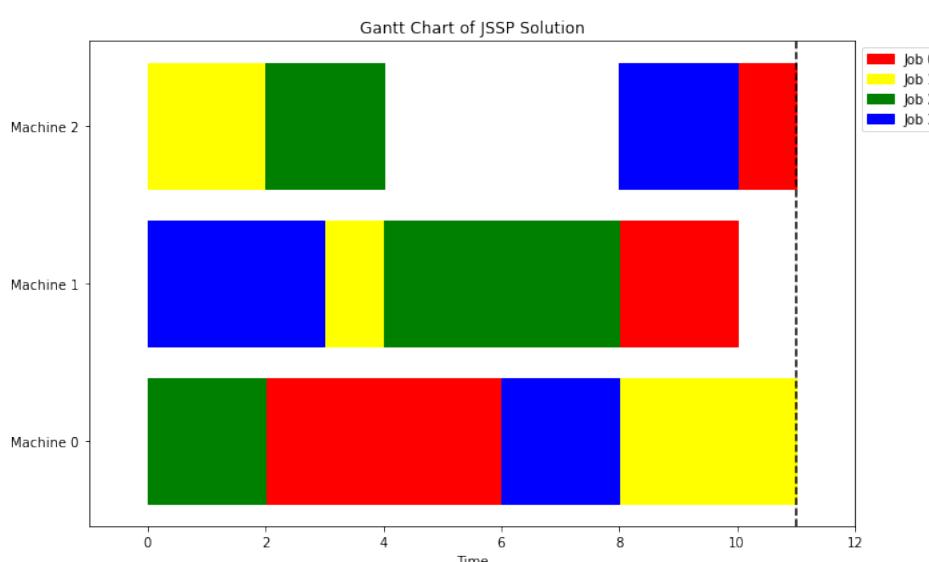
(13.1a)

Example: Duplo Scheduling

Gurobipy

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

The optimal value is 11.





13.9.3. 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.

13.10 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:

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

13.11 Generalized 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.

13.11.1. 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.

13.11.2. 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:

$$\text{maximize } \sum_{i=1}^m \sum_{j=1}^n p_{ij}x_{ij}. \quad (13.1)$$

$$\text{subject to } \sum_{j=1}^n w_{ij}x_{ij} \leq t_i \quad i = 1, \dots, m; \quad (13.2)$$

$$\sum_{i=1}^m x_{ij} = 1 \quad j = 1, \dots, n; \quad (13.3)$$

$$x_{ij} \in \{0, 1\} \quad i = 1, \dots, m, \quad j = 1, \dots, n; \quad (13.4)$$

13.12 Other material

13.12.1. 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

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \quad (13.1)$$

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

$$\begin{aligned} x &= \sum_{i=1}^u i z_i = z_1 + 2z_2 + \dots + uz_u \\ 1 &\geq \sum_{i=1}^u z_i = z_1 + z_2 + \dots + z_u \\ z_i &\in \{0, 1\} \quad \text{for } i = 1, \dots, u \end{aligned} \quad (13.2)$$

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:

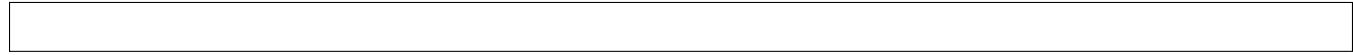
$O(\log u)$ many binary variables

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

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \quad (13.3)$$

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

$$\begin{aligned} x &= \sum_{i=0}^{\lfloor \log(u) \rfloor} 2^i z_i = z_0 + 2z_1 + 4z_2 + 8z_3 + \dots + 2^{\lfloor \log(u) \rfloor} z_{\lfloor \log(u) \rfloor} \\ z_i &\in \{0, 1\} \quad \text{for } i = 1, \dots, \lfloor \log(u) \rfloor \end{aligned} \quad (13.4)$$



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.

13.13 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](#)

13.14 MIP Solvers and Modeling Tools

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

13.14.1. Tools for Solving Job Shop Scheduling Problems

Job Shop Scheduling (JSS) is a classic optimization problem in operations research. There are several tools and techniques available to tackle this problem, ranging from exact algorithms to heuristic and meta-heuristic methods.

1. Exact Algorithms:

- **Branch and Bound:** This is a general algorithm for finding optimal solutions. It systematically searches for the best solution by exploring all possible solutions in a tree-like structure.
- **Integer Linear Programming (ILP):** Many commercial solvers like IBM CPLEX, Gurobi, and SCIP can be used to model and solve JSS as an ILP problem.

2. Heuristic Methods:

- **Priority Dispatching Rules:** These are simple rules like Shortest Processing Time (SPT), Earliest Due Date (EDD), and Longest Processing Time (LPT) that prioritize jobs based on certain criteria.
- **Shifting Bottleneck:** This heuristic focuses on the most constrained machine (the bottleneck) and schedules jobs on it first.

3. Metaheuristic Methods:

- **Genetic Algorithms (GA):** GA is inspired by the process of natural selection. Tools like JGAP (Java Genetic Algorithms Package) can be used to implement GA for JSS.
- **Simulated Annealing (SA):** SA is inspired by the annealing process in metallurgy. It's a probabilistic technique used to find an approximate solution to an optimization problem.
- **Tabu Search:** This is a local search method that uses memory structures to avoid getting trapped in local optima.
- **Particle Swarm Optimization (PSO):** Inspired by the social behavior of birds, PSO is used to find optimal or near-optimal solutions.

4. Hybrid Methods:

- Combining two or more of the above methods can often yield better results. For example, a GA can be combined with SA or Tabu Search to enhance the search process.

5. Software and Libraries:

- **OptaPlanner:** An open-source constraint satisfaction solver in Java that can handle JSS problems.
- **OR-Tools by Google:** Provides a suite of operations research tools, including solvers for routing, linear programming, and constraint programming, which can be applied to JSS.

6. Simulation Software:

- Tools like FlexSim, AnyLogic, and Arena can be used to simulate and optimize job shop scheduling scenarios.

7. Commercial Packages:

- Some ERP (Enterprise Resource Planning) and MES (Manufacturing Execution Systems) software offer built-in tools for job shop scheduling.

When choosing a tool or method, it's essential to consider the specific requirements of the problem, the size of the problem instance, and the available computational resources. For smaller instances, exact methods might be feasible, but for larger, real-world problems, heuristic or metaheuristic methods are often more appropriate.

14. 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.

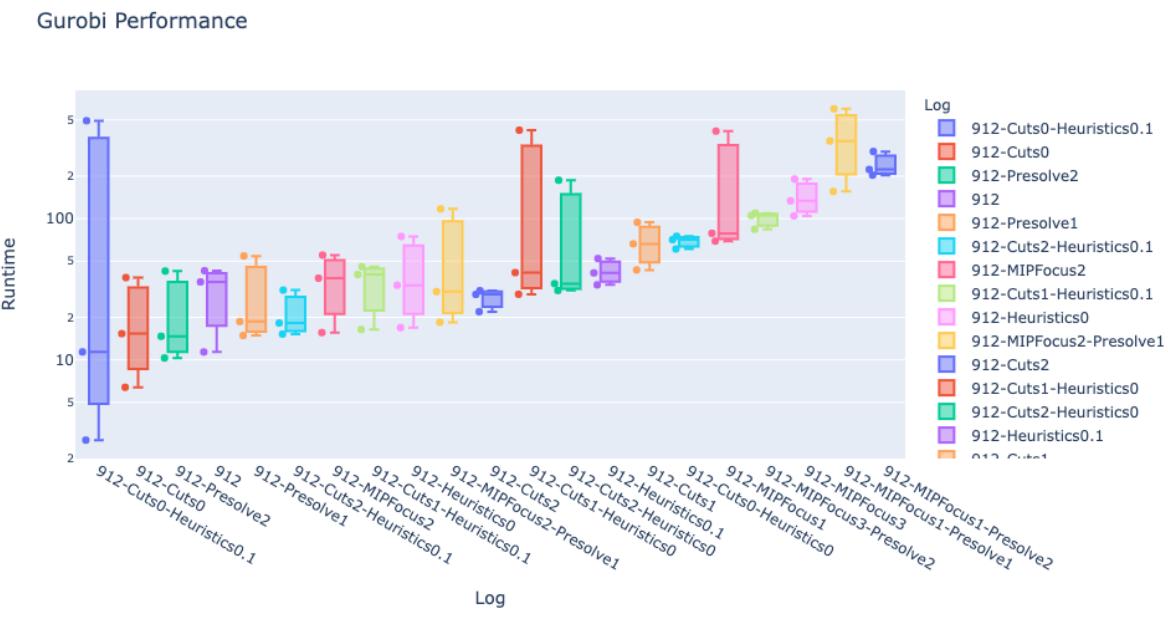


Figure 14.1: GUROBI Performance on a set of problems while varying different possible settings.

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.

14.1 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 14.1: LP Bounds

It always holds that

$$z_{IP}^* \leq z_{LP}^*. \quad (14.1)$$

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

$$x_{LP}^* = x_{IP}^* \text{ and } z_{LP}^* = z_{IP}^*. \quad (14.2)$$

Example 14.2

Consider the problem

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

14.1.1. Rounding LP Solution can be bad!

Consider the two variable knapsack problem

$$\max 3x_1 + 100x_2 \quad (14.3)$$

$$x_1 + 100x_2 \leq 100 \quad (14.4)$$

$$x_i \in \{0, 1\} \text{ for } i = 1, 2. \quad (14.5)$$

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.

14.1.2. 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

$$\min x_1 + 200x_2 \quad (14.6)$$

$$x_1 + 100x_2 \geq 100 \quad (14.7)$$

$$x_i \in \{0, 1\} \text{ for } i = 1, 2. \quad (14.8)$$

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.

14.2 Branch and Bound

14.2.1. 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 5 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.
-

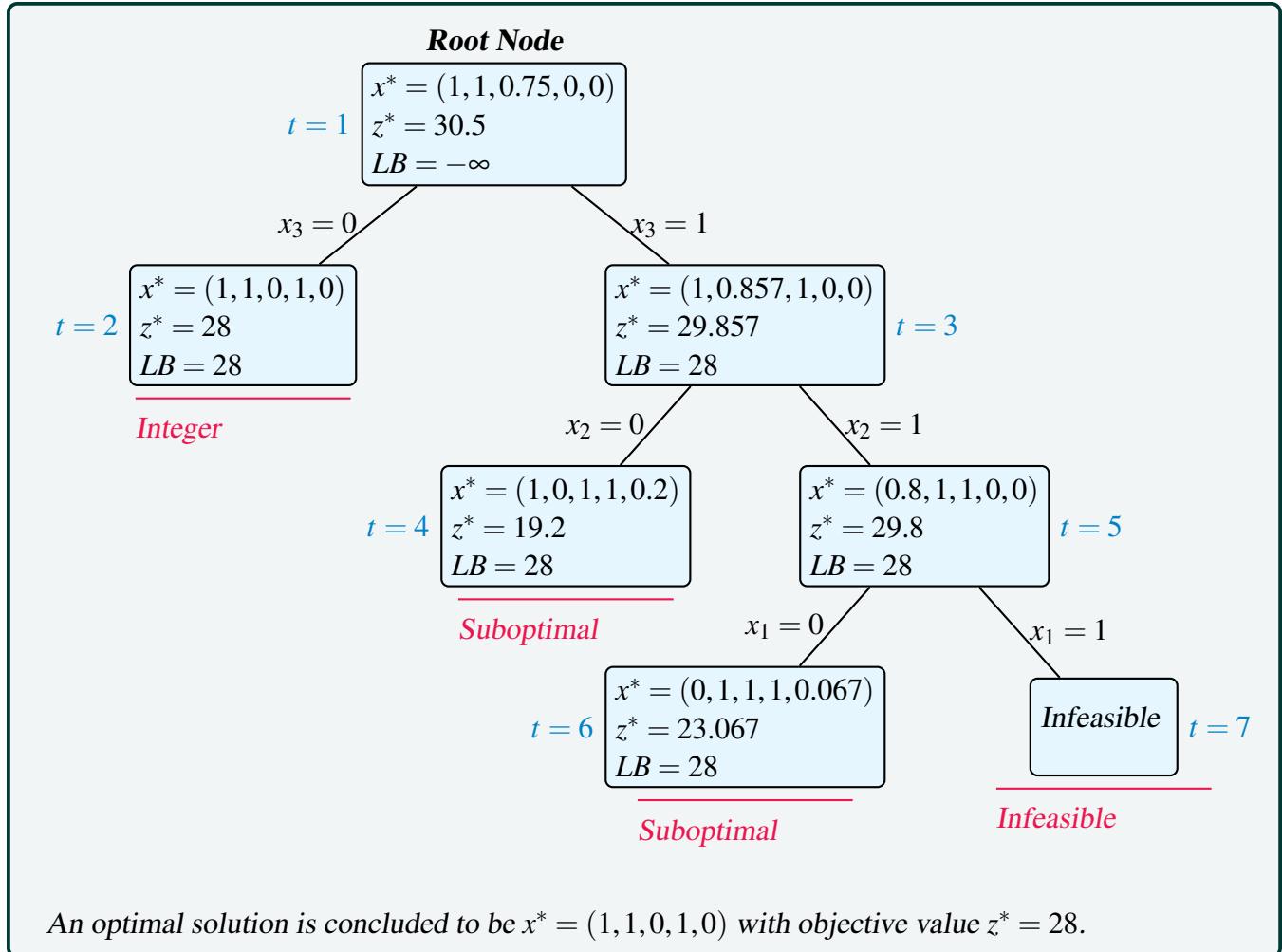
Let's look at a nice example. This is the smallest example that exhibits all the interesting behavior of branch and bound. Typically these branch and bound trees are much much much bigger.

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 14.3: Binary Knapsack Example

Solve the following problem with branch and bound.

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



14.2.2. 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 6 Branch and Bound - Maximization

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

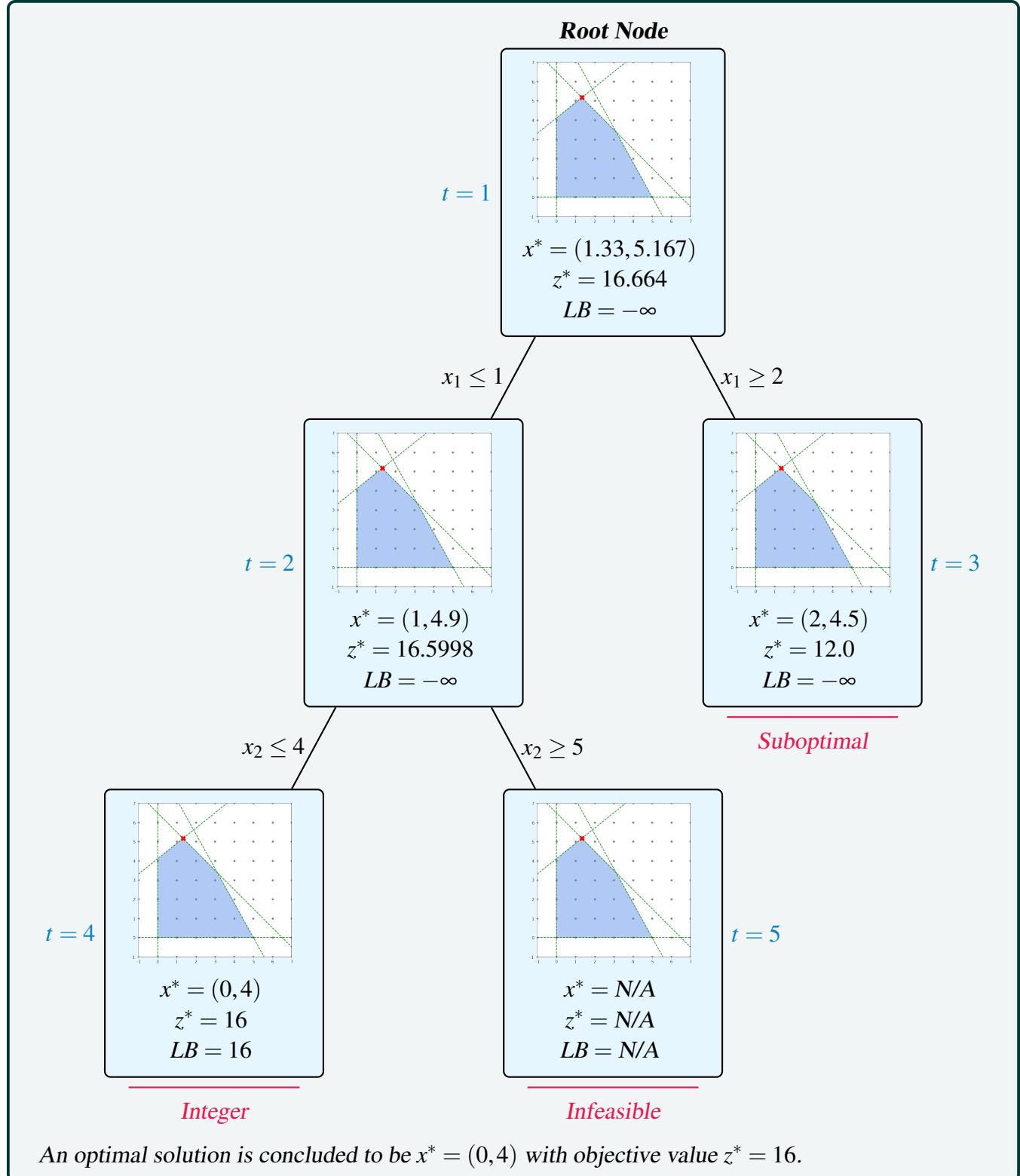
- 1: Set $LB = -\infty$.
 - 2: Solve LP relaxation.
 - 3: **if** x^* is integer **then**
 - 4: Stop!
 - 5: **else**
 - 6: Choose fractional entry x_i^* .
 - 7: Branch onto subproblems:
 - (i) $x_i \leq \lfloor x_i^* \rfloor$
 - (ii) $x_i \geq \lceil x_i^* \rceil$
 - 8: Solve LP relaxation of any subproblem.
 - 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 14.4: Branching on general integer variables

Consider the two variable example with

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



14.3 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.

14.3.1. Cover Inequalities

Consider the binary knapsack problem

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

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*

$$\sum_{i \in S} x_i \leq |S| - 1 \Rightarrow x_1 + x_2 + x_3 + x_4 \leq 4 - 1 = 3. \quad (14.1)$$

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

$$x_1 + x_2 \leq 2 - 1 = 1 \quad (14.2)$$

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

$$S = \{1, 4\} \Rightarrow x_1 + x_4 \leq 1 \quad (14.3)$$

and

$$S = \{2, 3, 4\} \Rightarrow x_2 + x_3 + x_4 \leq 2 \quad (14.4)$$



14.3.2. Chvátal Cuts

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

Chvátal Cuts:

Suppose

$$a_1x_1 + \cdots + a_nx_n \leq d \quad (14.5)$$

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

$$\lfloor a_1 \rfloor x_1 + \cdots + \lfloor a_n \rfloor x_n \leq \lfloor d \rfloor \quad (14.6)$$

is valid for the integer points in P , that is, it is valid for the set $P \cap \mathbb{Z}^n$. Equation (14.6) is called a Chvátal Cut.

We will illustrate this idea with an example.

Example 14.5

Recall example 7. 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.

14.3.3. 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 7 (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 7.

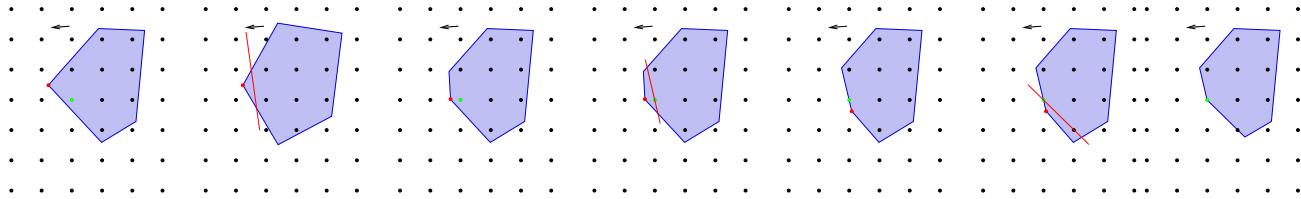


Figure 14.2: The cutting plane procedure.

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>

14.3.4. 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

$$x_j + \sum_{i \in N} \tilde{a}_i x_i = \tilde{b}_j \quad (14.7)$$

is an equation in the optimal simplex tableau.

Gomory Cut:

The Gomory cut corresponding to the tableau row (14.7) is

$$\sum_{i \in N} (\tilde{a}_i - \lfloor \tilde{a}_i \rfloor) x_i \geq \tilde{b}_j - \lfloor \tilde{b}_j \rfloor \quad (14.8)$$

We will solve the following problem using only Gomory Cuts.

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

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

$$\begin{array}{lll} \min & x_1 - 2x_2 \\ \text{s.t.} & -4x_1 + 6x_2 + s_1 & = 9 \\ & x_1 + x_2 + s_2 & = 4 \\ & x \geq 0 & , \quad x_1, x_2 \in \mathbb{Z} \end{array} \quad (14.9)$$

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 14.6

(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

$$x_1 - 0.1s_1 + -0.6s_2 = 1.5. \quad (14.10)$$

Applying the Gomory Cut (14.8), we have the inequality

$$0.9s_1 + 0.4s_2 \geq 0.5. \quad (14.11)$$

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 14.7: (Gomory Cut in Original Space)

The Gomory Cut (14.11) can be rewritten in the original variables using the equations from (14.9). That is, we can use the equations

$$\begin{aligned} s_1 &= 9 + 4x_1 - 6x_2 \\ s_2 &= 4 - x_1 - x_2, \end{aligned} \quad (14.12)$$

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

$$-3.2x_1 + 5.8x_2 \leq 9.2. \quad (14.13)$$

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

Example 14.8: (Gomory cuts plus new tableau)

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

$$0.9s_1 + 0.4s_2 - s_3 = 0.5. \quad (14.14)$$

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

14.4 Interpreting Output Information and Progress

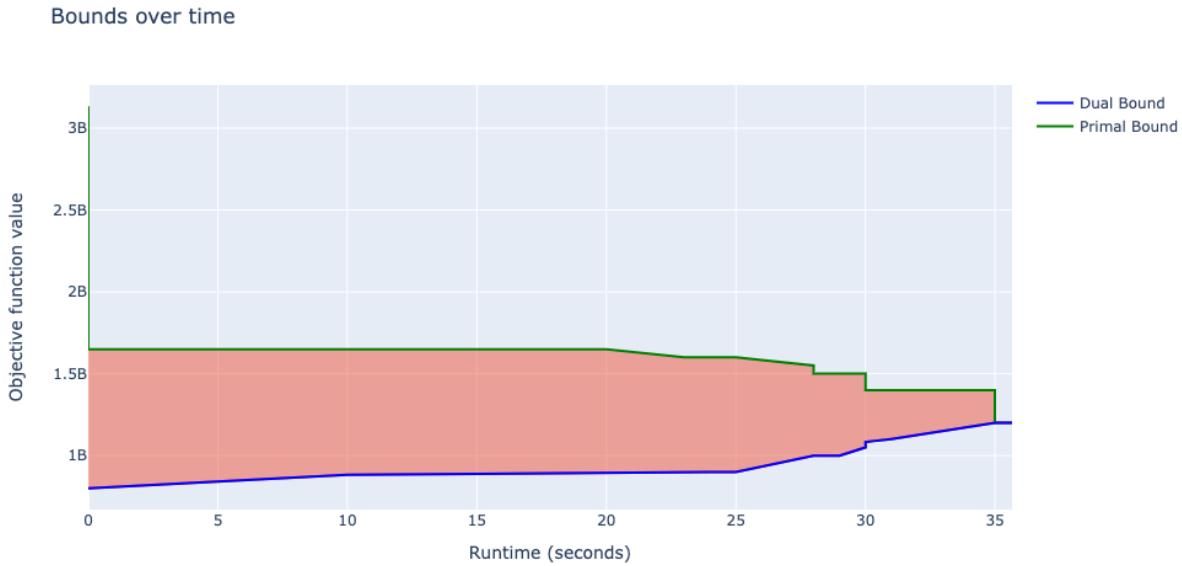


Figure 14.3: This shows the progress of the solver over time.

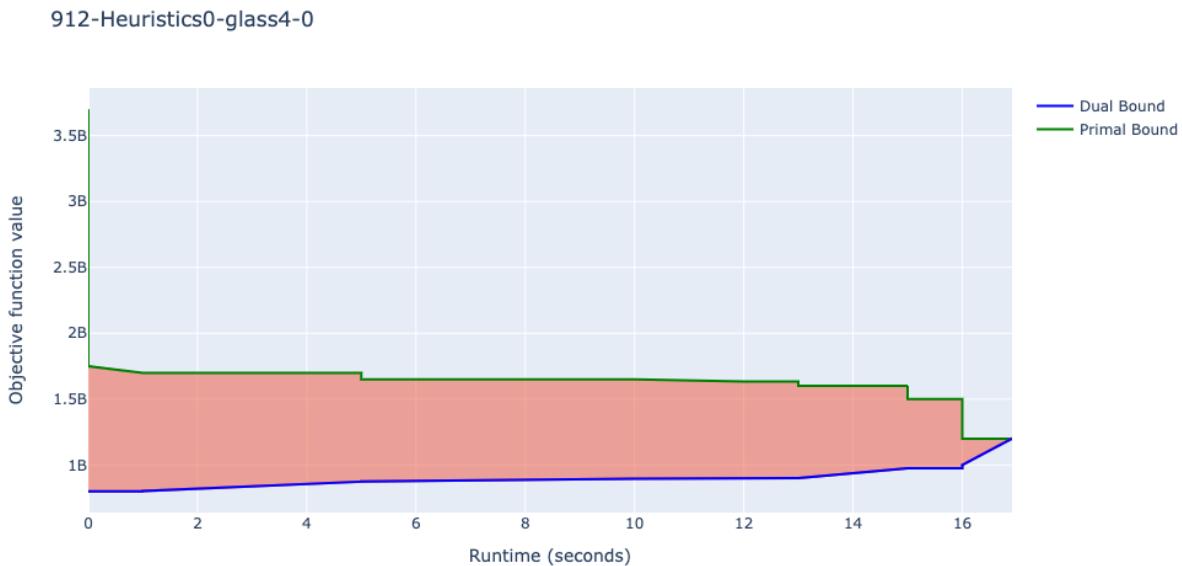


Figure 14.4: This shows the progress of the solver over time.

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.

14.5 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

14.6 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.

14.6.1. 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.

14.6.2. 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.

14.6.3. 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.

14.7 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*

15. 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.

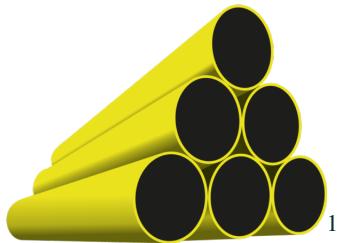
Although typically models need to be a reasonable size in order for us to code them and send them to a solver, there are some ways that we can allow having models of exponential size. The first example here is the cutting stock problem, where we will model with exponentially many variables. The second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

15.1 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 15.1: 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

$$\begin{aligned}
 & \min \sum_{j=1}^N z_j \\
 \text{s.t. } & \sum_{i=1}^k L_i x_{ij} \leq L z_j \quad \text{for } j = 1, \dots, N \\
 & \sum_{j=1}^N x_{ij} \geq d_i \quad \text{for } i = 1, \dots, k \\
 & z_j \in \{0, 1\} \quad \text{for } j = 1, \dots, N \\
 & x_{ij} \in \mathbb{Z}_+ \quad \text{for } i = 1, \dots, k, j = 1, \dots, N
 \end{aligned} \tag{15.1}$$

Exercise 15.2: 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 15.1: 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

$$\begin{aligned}
 & \min \sum_{j=1}^{16} z_j \\
 \text{s.t. } & 4x_{1j} + 5x_{2j} + 6x_{3j} \leq 19z_j \\
 & \sum_{j=1}^{16} x_{1j} \geq 12 \\
 & \sum_{j=1}^{16} x_{2j} \geq 15 \\
 & \sum_{j=1}^{16} x_{3j} \geq 22 \\
 & z_j \in \{0, 1\} \text{ for } j = 1, \dots, 16 \\
 & x_{ij} \in \mathbb{Z}_+ \text{ for } i = 1, \dots, 3, j = 1, \dots, 16
 \end{aligned} \tag{15.2}$$

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

$$z_1 \geq z_2 \geq z_3 \geq \dots \geq z_N. \tag{15.3}$$

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:

15.1.1. 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

$$\sum_{i=1}^k L_i a_i \leq L \quad (15.4)$$

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

In our running example, we have

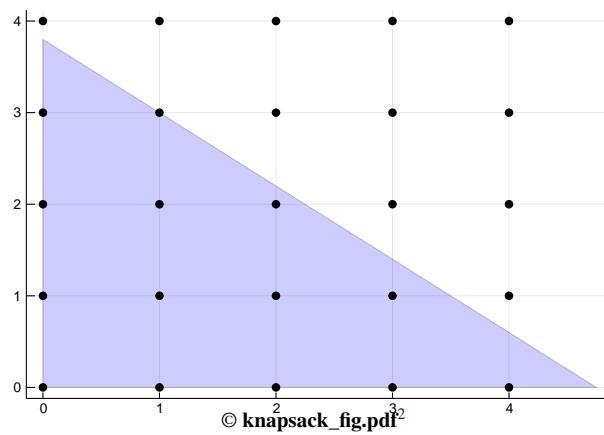
$$4a_1 + 5a_2 + 6a_3 \leq 19 \quad (15.5)$$

$a_i \in \mathbb{Z}_+$ 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

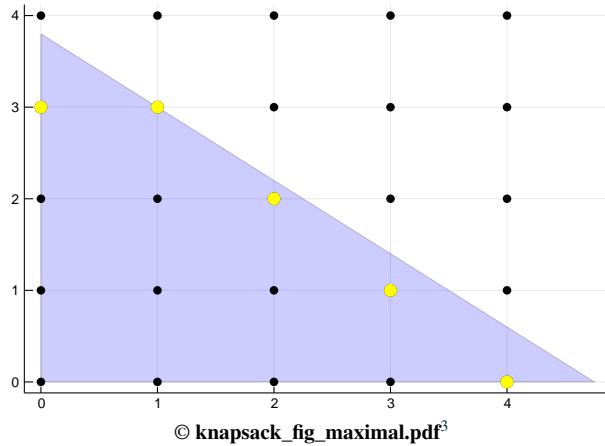
$$P = \{(a_1, a_2) : 4a_1 + 5a_2 \leq 19, a_1 \geq 0, a_2 \geq 0\} \quad (15.6)$$

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 15.3: 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.

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 2 & 1 & 2 & 3 & 4 & 1 \\ 0 & 1 & 0 & 2 & 1 & 2 & 2 & 1 & 0 & 3 \\ 3 & 2 & 2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad (15.7)$$

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

$$\min \sum_{i=1}^p x_i \quad (15.8)$$

$$\text{such that } Ax \geq \begin{bmatrix} 12 \\ 15 \\ 22 \end{bmatrix} \quad (15.9)$$

$$x \in \mathbb{Z}_+^p \quad (15.10)$$

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

15.1.2. Column Generation

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

Thus we can write it as

$$\begin{aligned} \min \quad & (c_N - c_B A_B^{-1} A_N) x_N + c_B A_B^{-1} b \\ \text{s.t.} \quad & x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ & x \geq 0 \end{aligned} \tag{15.11}$$

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

$$\begin{aligned} \min \quad & (1_N - 1_B A_B^{-1} N) x_N + 1_B A_B^{-1} b \\ \text{s.t.} \quad & x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ & x \geq 0 \end{aligned} \tag{15.12}$$

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

$$(1 - 1_B A_B^{-1} A_N^i) x_i \tag{15.13}$$

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

$$1 - 1_B A_B^{-1} a < 0 \Rightarrow 1 < 1_B A_B^{-1} a \tag{15.14}$$

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.

$$\begin{aligned} \max \quad & 1_B A_B^{-1} a \\ \text{s.t.} \quad & \sum_{i=1}^k L_i a_i \leq L \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k \end{aligned} \tag{15.15}$$

Example 15.4: 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

$$A = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{pmatrix} \quad (15.16)$$

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.

$$A = \begin{pmatrix} 4 & 0 & 0 & a_1 \\ 0 & 3 & 0 & a_2 \\ 0 & 0 & 3 & a_3 \end{pmatrix} \quad (15.17)$$

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

$$\begin{aligned} \max \quad & \frac{1}{4}a_1 + \frac{1}{3}a_2 + \frac{1}{3}a_3 \\ \text{s.t.} \quad & 4a_1 + 5a_2 + 6a_3 \leq 19 \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k \end{aligned} \quad (15.18)$$

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.

15.1.3. 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

15.2 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!

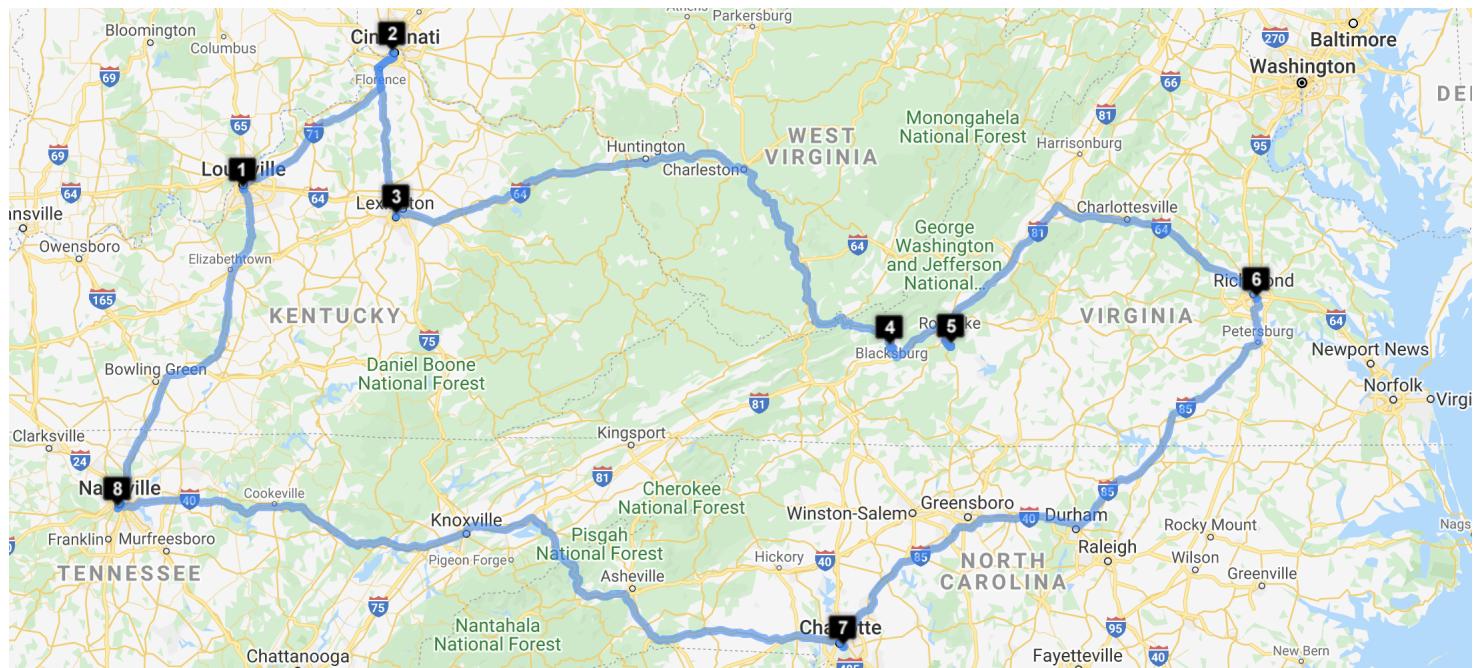


Figure 15.1: Optimal tour through 8 cities. Generated by Gebweb - Optimap. See it also on Google Maps!.

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:

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (15.1)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (15.2)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (15.3)$$

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

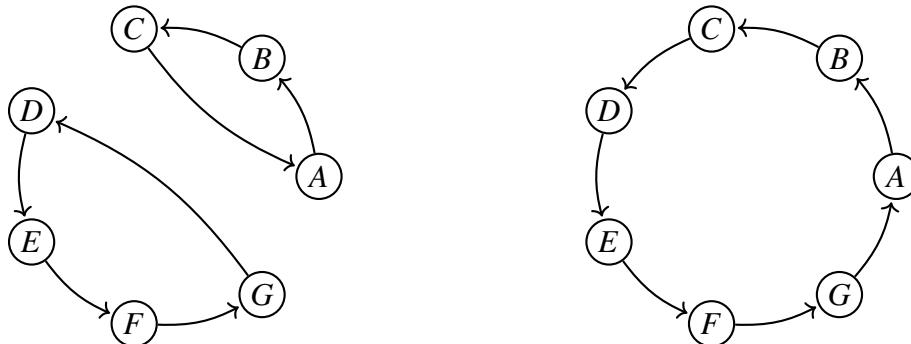


Figure 15.2: Left: An assignment solution where each node has an incoming arc and an outgoing arc. But in this case, there are 2 subtours. **Right:** An assignment that achieves a proper tour. Note that both of these assignments have the same number of arcs!

15.2.1. 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 (15.1)-(15.3) 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

$$\text{If } x_{ij} = 1, \text{ then } u_i + 1 \leq u_j. \quad (15.4)$$

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

$$u_i + 1 \leq u_j + n(1 - x_{ij}) \quad (15.5)$$

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 (15.5), we have that

$$u_2 < u_5 < u_3 < u_2, \quad (15.6)$$

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:

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (15.7)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (15.8)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (15.9)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (15.10)$$

$$u_i + 1 \leq u_j + n(1 - x_{ij}) \quad \text{for all } i, j \in N, i, j \neq 1 \quad [\text{prevents subtours}] \quad (15.11)$$

$$u_1 = 1 \quad (15.12)$$

$$2 \leq u_i \leq n \quad \text{for all } i \in N, i \neq 1 \quad (15.13)$$

$$u_i \in \mathbb{Z} \quad \text{for all } i \in N \quad (15.14)$$

$$x_{ij} \in \{0, 1\} \quad \text{for all } i, j \in N \quad (15.15)$$

Example 15.5: 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 15.6: 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 15.7: 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 15.8: 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

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

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

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

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

15.2.2. 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

$$2 \rightarrow 5 \rightarrow 3 \rightarrow 2. \quad (15.16)$$

We can prevent this subtour by adding the constraint

$$x_{25} + x_{53} + x_{32} \leq 2 \quad (15.17)$$

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*

$$\sum_{(i,j) \in S} x_{ij} \leq |S| - 1 \quad \text{Subtour Elimination Constraint.} \quad (15.18)$$

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:

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (15.19)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (15.20)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (15.21)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (15.22)$$

$$\sum_{(i,j) \in S} x_{ij} \leq |S| - 1 \quad \text{for all subtours } S \quad [\text{prevents subtours}] \quad (15.23)$$

$$x_{ij} \in \{0,1\} \quad \text{for all } i, j \in N \quad (15.24)$$

Example 15.9: 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 15.10

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*.

15.2.3. Traveling Salesman Problem - Branching Solution

We will see in the next section

1. That the constraint (15.1)-(15.3) 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.

15.2.4. Traveling Salesman Problem Variants**15.2.4.1. Many salespersons (m-TSP)**

m-Traveling Salesman Problem - DFJ Model:

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (15.25)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (15.26)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \text{ [incoming arc]} \quad (15.27)$$

$$\sum_{j \in N} x_{Dj} = m \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (15.28)$$

$$\sum_{i \in N} x_{iD} = m \quad \text{for all } j \in N \text{ [incoming arc]} \quad (15.29)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \cup \{D\} \text{ [no self loops]} \quad (15.30)$$

$$\sum_{(i,j) \in S} x_{ij} \leq |S| - 1 \quad \text{for all subtours } S \subseteq N \text{ [prevents subtours]} \quad (15.31)$$

$$x_{ij} \in \{0, 1\} \quad \text{for all } i, j \in N \cup \{D\} \quad (15.32)$$

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

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (15.33)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (15.34)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \text{ [incoming arc]} \quad (15.35)$$

$$\sum_{j \in N} x_{Dj} = m \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (15.36)$$

$$\sum_{i \in N} x_{iD} = m \quad \text{for all } j \in N \text{ [incoming arc]} \quad (15.37)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \cup \{D\} \text{ [no self loops]} \quad (15.38)$$

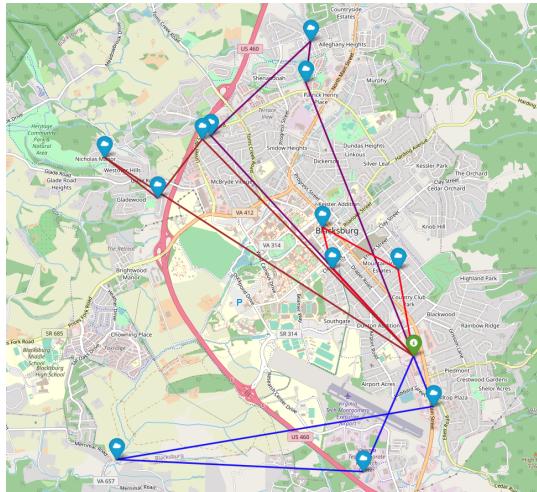
$$u_i + 1 \leq u_j + n(1 - x_{ij}) \quad \text{for all } i, j \in N, i, j \neq 1 \text{ [prevents subtours]} \quad (15.39)$$

$$u_1 = 1 \quad (15.40)$$

$$2 \leq u_i \leq T \quad \text{for all } i \in N, i \neq 1 \quad (15.41)$$

$$u_i \in \mathbb{Z} \quad \text{for all } i \in N \quad (15.42)$$

$$x_{ij} \in \{0, 1\} \quad \text{for all } i, j \in N \cup \{D\} \quad (15.43)$$



© m-tsp_solution⁴

[Image html](#)

15.2.4.2. 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

$$u_2 + 1 = u_3.$$

15.2.5. 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:

$$\min \sum_{i,j} c_{ij} \sum_k x_{ijk} \quad (15.44)$$

⁴m-tsp_solution, from m-tsp_solution. m-tsp_solution, m-tsp_solution.

Subject to:

$$\sum_i \sum_k x_{ijk} = 1 \quad \forall j \neq \text{depot} \quad (\text{Exactly one vehicle in}) \quad (15.45)$$

$$\sum_j \sum_k x_{ijk} = 1 \quad \forall i \neq \text{depot} \quad (\text{Exactly one vehicle out}) \quad (15.46)$$

$$\sum_i \sum_k x_{ihk} - \sum_j \sum_k x_{hjk} = 0 \quad \forall k, h \quad (\text{It's the same vehicle}) \quad (15.47)$$

$$\sum_i q_i \sum_j x_{ijk} \leq Q_k \quad \forall k \quad (\text{Capacity constraint}) \quad (15.48)$$

$$\sum_{ijk} x_{ijk} = |S| - 1 \quad \forall S \subseteq P(N), 0 \notin S \quad (\text{Subtour elimination}) \quad (15.49)$$

$$x_{ijk} \in \{0, 1\} \quad (15.50)$$

15.3 TSP with Time Windows

Let $G = (V, A)$ be a directed graph where V is the set of nodes (including the depot) and A is the set of arcs. Each arc $(i, j) \in A$ has a cost c_{ij} associated with it which represents the cost of traveling from node i to node j . The problem is to find a minimum-cost tour that visits each node within its time window.

Parameters

- n - the number of nodes.
- c_{ij} - cost of traveling from node i to node j .
- $[a_i, b_i]$ - time window of node i , where a_i and b_i are the earliest and latest time, respectively, that node i can be visited.
- M - a large positive number (big-M).

Decision Variables

- $x_{ij} = \begin{cases} 1, & \text{if arc } (i, j) \text{ is used in the solution} \\ 0, & \text{otherwise} \end{cases}$
- t_i - time of service beginning at node i .

Objective Function

$$\text{Minimize } Z = \sum_{i=1}^n \sum_{j=1, j \neq i}^n c_{ij} x_{ij} \quad (15.1)$$

Constraints:

Time Window Constraints

$$a_i \leq t_i \leq b_i \quad \forall i \in V \quad (15.2)$$

Subtour Elimination Constraints

$$t_i + (c_{ij} + s_i - t_j)x_{ij} \leq M(1 - x_{ij}) \quad \forall (i, j) \in A, i \neq j \quad (15.3)$$

where s_i is the service time at node i .

Arrival Time Constraints

$$t_j \geq t_i + s_i + c_{ij} - M(1 - x_{ij}) \quad \forall (i, j) \in A, i \neq j \quad (15.4)$$

Flow Conservation Constraints

$$\sum_{j=1, j \neq i}^n x_{ij} = 1 \quad \forall i \in V \quad (15.5)$$

$$\sum_{i=1, i \neq j}^n x_{ij} = 1 \quad \forall j \in V \quad (15.6)$$

Integer Constraints

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in A \quad (15.7)$$

15.4 Case study - Traveling Salesman Problem Applications for Supply Chain Optimization

While the traveling salesman problem was built predominantly for applications of a salesman going to distinct locations, a problem that I believe the TSP could be applied to would be supply chain optimization. We could consider the ‘salesman’ in this situation to be a specific product which has to have numerous components appended to the product for the product to be valuable. The iPhone has to go through numerous manufacturing facilities and have numerous components (including antennas, chips and micro-processors) for the phone to be valuable. This problem could be formulated to be a traveling salesman problem. The supply chain manager in this situation would want to ensure that the product gets to each manufacturing facility for the components to be attached to the product. However, we would want to minimize geographic distance traveled. We would furthermore have to consider aspects including whether the manufacturing facility is reliable (would be quantified by some sort of quality score), the amount of product that the manufacturing facility could supply (sometimes special components including materials could be difficult to obtain) and the cost of the manufacturing process (some manufacturing processes would be expensive compared to others). There are a couple of interesting features of this supply chain optimization problem which distinguish this against a normal traveling salesman problem. To start, we have to consider numerous factors other than the geographic distance that the product travels. We have to consider factors including cost of process, time to ship (shipping by boat requires time compared to shipping by air) and quality of the manufacturing process. For a phone, electronic robot manufacturing would be reliable compared to human manufacturing, however the cost of human manufacturing in poorer countries would be lower. Phones manufactured in Asia would be cheaper than phones manufactured in

the USA, however shipping times would be longer. These features would mathematically have to be built into the model through constraints. Each manufacturing facility could have a specific group of features assigned to the facility. Some electronic manufacturing facilities in the USA could have a quality (Q) = 8, cost (C) = 9 and shipment time (S) = 3. A human manufacturing facility in India could have a quality (Q) = 4, cost (C) = 3 and shipment time (S) = 8. The shipment time scores could be built into the objective function of the model and the quality and cost scores could be built into the constraints. We could have constraints which would decide if a location would be selected if the cost and quality score are there inside a specific target number. I would mathematically incorporate a DFJ model. Some other feature that would have to be considered would be if we could have numerous ‘salesmen’ for the supply chain. For highly complex supply chains including cars, there are numerous supply chains for each component of the car. For this supply chain, we could disassemble the car into numerous components (including engines, wheels and electronic components) and build out traveling salesmen algorithms for each of those components. This could be considered a vehicle routing algorithm because of the fact that we are allowed to have numerous supply chains. Furthermore, I could consider the situation if numerous supply chains each require a specific manufacturing facility. We would have to combine the traveling salesman problem and the network flow problem to create a method for managing this. The network flow could figure out if the manufacturing facility has the capacity to create the product.

Data I have attached to this paper a Excel data that would be good for building a supply chain optimization model through the TSP. The data includes data of shipment times, expense and throughput rates for manufacturing facilities and which things could be produced by which manufacturing facility. I would build the traveling salesman model to start by figuring out which manufacturing facility is required before the others. I would create the model for the product to go through each manufacturing facility while minimizing cost through shipments and manufacturing cost. I have attached some simple code for the supply chain that includes the cost and capacity constraints, however this model could be extended for other features. We could include that a certain quality score has to be maintained for a facility to be selected. The code demonstrates that two subtours could be generated. One subtour is going to be Singapore to Thailand to Mumbai. The other subtour is NY to Chicago to LA. We could then attach a route for LA to Singapore to attach the subtours together. This would lead to a total distance of around 15000 kilometers. I built constraints to this model to ensure that shipment time, cost and quality satisfied constraints.

This supply chain optimization problem differs from a standard TSP in several ways. It involves factors beyond geographic distance, including manufacturing cost, shipping time (e.g., air vs. sea transport), and manufacturing process quality. For instance, electronic robot manufacturing may offer higher reliability but at a greater cost compared to human manufacturing in lower-wage countries. These factors must be incorporated mathematically through constraints. Each manufacturing facility can be assigned specific quality (Q), cost (C), and shipment time (S) scores, which influence the model’s objective function and constraints.

MATHEMATICAL MODEL The model can incorporate a Decision, Features, and Juxtaposition (DFJ) approach. Constraints determine if a location is selected based on target quality and cost scores. Additional considerations include the possibility of multiple 'salesmen' for complex supply chains, such as in the automotive industry. Each component (e.g., engines, wheels, electronics) can have its TSP algorithm, resembling a vehicle routing problem. Combining TSP and network flow problem techniques can manage these scenarios, with network flow determining manufacturing facility capacity.

DATA Attached to this paper is Excel data suitable for building a TSP-based supply chain optimization model. The data includes shipment times, expenses, throughput rates, and manufacturing facility-product compatibility. The TSP model starts by identifying the required manufacturing facility and then routes the product through each facility, minimizing costs and considering quality constraints.

MODEL

SETS

- $i, j \in \text{Facilities}$: Set of facilities to visit, indexed by i and j .

PARAMETERS

- $x_{ij} \in \{0, 1\}$: Binary decision variable representing whether to travel from facility i to facility j .
- d_{ij} : Haversine distance between facility i and facility j (calculated based on latitude and longitude).
- Q_{ij} : Quality score associated with traveling from facility i to facility j .
- C_{ij} : Cost associated with traveling from facility i to facility j .
- S_{ij} : Shipment time associated with traveling from facility i to facility j .

OBJECTIVE The objective is to minimize the total distance traveled while considering the quality, cost, and shipment time associated with each facility visit:

$$\text{Minimize} \quad \sum_{i \in \text{Facilities}} \sum_{j \in \text{Facilities}, i \neq j} d_{ij} \cdot x_{ij} \quad (15.1)$$

CONSTRAINTS

- Facility Visit Constraints:

$$\sum_{j \in \text{Facilities}, j \neq i} x_{ij} = 1, \quad \forall i \in \text{Facilities} \quad (15.2)$$

- Facility Leave Constraints:

$$\sum_{i \in \text{Facilities}, i \neq j} x_{ij} = 1, \quad \forall j \in \text{Facilities} \quad (15.3)$$

- Quality Score Constraint (Sum of quality scores for selected routes):

$$\sum_{i \in \text{Facilities}} \sum_{j \in \text{Facilities}, i \neq j} Q_{ij} \cdot x_{ij} \leq 150 \quad (15.4)$$

- Cost Constraint (Sum of costs for selected routes):

$$\sum_{i \in \text{Facilities}} \sum_{j \in \text{Facilities}, i \neq j} C_{ij} \cdot x_{ij} \leq 250 \quad (15.5)$$

- Shipment Time Constraint (Sum of shipment times for selected routes):

$$\sum_{i \in \text{Facilities}} \sum_{j \in \text{Facilities}, i \neq j} S_{ij} \cdot x_{ij} \leq 100 \quad (15.6)$$

SOLUTION The Gurobi optimizer is used to solve the TSP model. The optimal solution provides the sequence of facility visits that minimizes the total distance traveled, while meeting quality, cost, and shipment time constraints.

CODE A simple code sample for the supply chain, incorporating cost and capacity constraints, is provided. This code can be extended to include other features, such as maintaining a specific quality score for facility selection. The code generates two subtours: Singapore to Thailand to Mumbai and NY to Chicago to LA. To connect these subtours, an additional route from LA to Singapore is added, resulting in a total distance of approximately 15,000 kilometers. Constraints ensure that shipment time, cost, and quality meet specified criteria.

ENHANCEMENTS To improve this implementation, the TSP should account for international travel by increasing shipment time constraints.

15.5 Vehicle Routing Problem (VRP)

The VRP is a generalization of the TSP and comes in 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.

15.5.1. 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.

15.5.2. 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.

15.5.3. Saving Calculation

The saving s_{ij} of merging the route of customer i and customer j is given by:

$$s_{ij} = c(0,i) + c(0,j) - c(i,j) \quad (15.1)$$

Where:

- $c(x,y)$ represents the cost (or distance) between node x and node y .
- 0 is the depot.

15.5.4. 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.

15.5.5. 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.

15.6 Tools to solve VRP

There are several tools and software packages available to solve VRPs, ranging from commercial software to open-source libraries. Here are some recommended tools:

1. Commercial Software:

- **Lingo:** A mathematical optimization tool that can be used to model and solve VRPs.
- **IBM ILOG CPLEX Optimization Studio:** Offers powerful optimization modeling capabilities and can solve large-scale VRPs.
- **Gurobi Optimizer:** Another commercial optimization solver that's known for its speed and efficiency.

2. Open-Source Solvers:

- **OR-Tools:** Developed by Google, OR-Tools is a set of optimization tools that includes solvers for the VRP. It's one of the most popular open-source tools for solving VRPs.

- **JSprit**: A Java-based, open-source toolkit for solving rich VRPs. It's lightweight and can handle various constraints and objectives.
- **VROOM**: A vehicle routing optimization library that focuses on real-world applications with fast execution times.

3. Frameworks and Libraries:

- **OptaPlanner**: An open-source constraint solver in Java that can tackle VRPs among other planning problems.
- **AequilibraE**: A Python library and QGIS plugin for transportation modeling, including tools for solving VRPs.

4. Simulation-Based:

- **AnyLogic**: A simulation modeling tool that can be used to model and solve complex VRPs using agent-based, discrete event, and system dynamics simulation methodologies.

5. Online Platforms:

- **Routific**: A cloud-based route optimization solution that can handle various real-world constraints.
- **OptimoRoute**: An online platform for route planning and optimization suitable for small to medium-sized delivery operations.

6. Heuristic and Metaheuristic Approaches:

- Many researchers and practitioners use heuristic and metaheuristic algorithms like Genetic Algorithms, Simulated Annealing, Tabu Search, and Ant Colony Optimization to solve VRPs. There are various libraries and toolkits in languages like Python, Java, and C++ that can be used to implement these algorithms.

When choosing a tool, consider the following factors:

- **Problem Size**: Some tools are better suited for large-scale problems, while others are designed for smaller instances.
- **Complexity**: If your VRP has many constraints or special requirements, you'll need a more flexible and powerful tool.
- **Budget**: Commercial software can be expensive, but they often come with support and advanced features. Open-source tools are free but might require more time to set up and customize.
- **Programming Skills**: Some tools require programming knowledge, while others offer a graphical interface.

Remember that the VRP is NP-hard, which means that as the problem size grows, the time required to find an optimal solution can increase exponentially. In many cases, especially for large problems, heuristic or metaheuristic approaches are used to find good (but not necessarily optimal) solutions in a reasonable amount of time.

15.7 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

15.7.1. TSP In Excel

15.7.2. Resources

TSP

Resources

See math Waterloo.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!](#)
[TSP with excel solver](#)

Google maps data: [Blog - Python | 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](#)

16. 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).

16.1 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 16.1: 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

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0. \quad (16.1)$$

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 16.2: 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

$$\limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty. \quad (16.2)$$

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 16.3: 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

$$0 \leq f(n) \leq c g(n) \quad \text{for all } n \geq n_0. \quad (16.3)$$

Example 16.4:

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 16.3 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

$$2. \ n \leq n^2 \text{ and therefore } 10n \leq 10n^2$$

$$3. \ 1 \leq n^2 \text{ and therefore } 7 \leq 7n^2$$

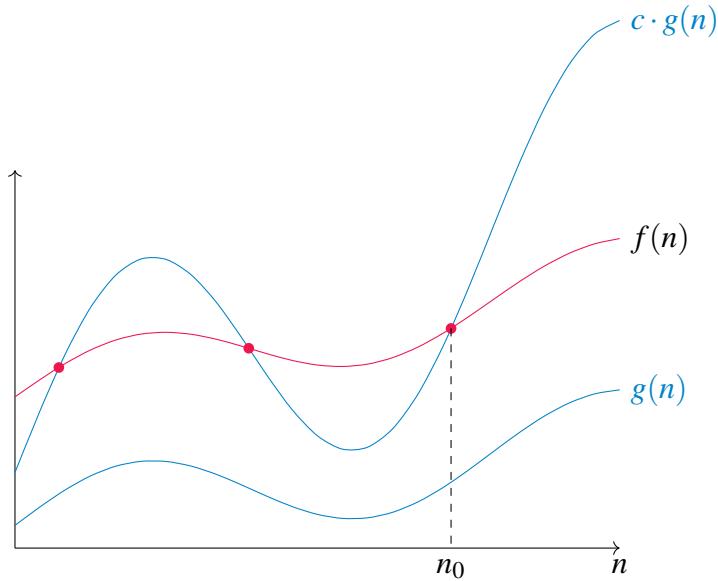


Figure 16.1: Example of Big-O notation: $f(n) = O(g(n))$. We see that for all $n \geq n_0$, we have $c \cdot g(n) \geq f(n)$.

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

$$5n^2 + 10n + 7 \leq 5n^2 + 10n^2 + 7n^2 \quad \text{for all } n \geq 1, \quad (16.4)$$

$$\Rightarrow 5n^2 + 10n + 7 \leq 22n^2 \quad \text{for all } n \geq 1. \quad (16.5)$$

Hence, we have shown that Equation 16.3 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 16.2: time-of-algorithms

16.2 Algorithms - Example with Bubble Sort

The following definition comes from Merriam-Webster's dictionary.

Definition 16.5: 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.

16.2.1. 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 16.6: 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](#).

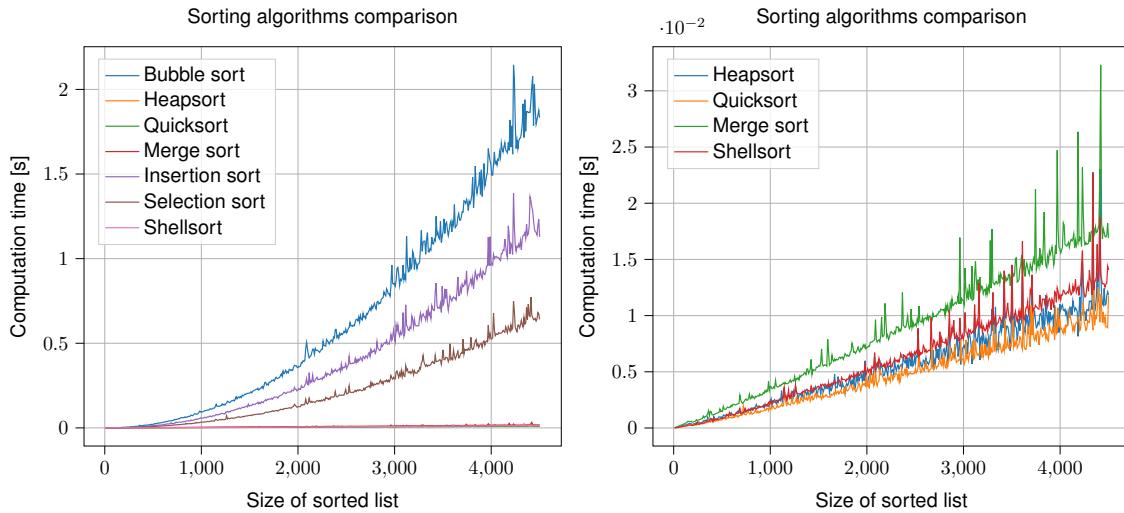
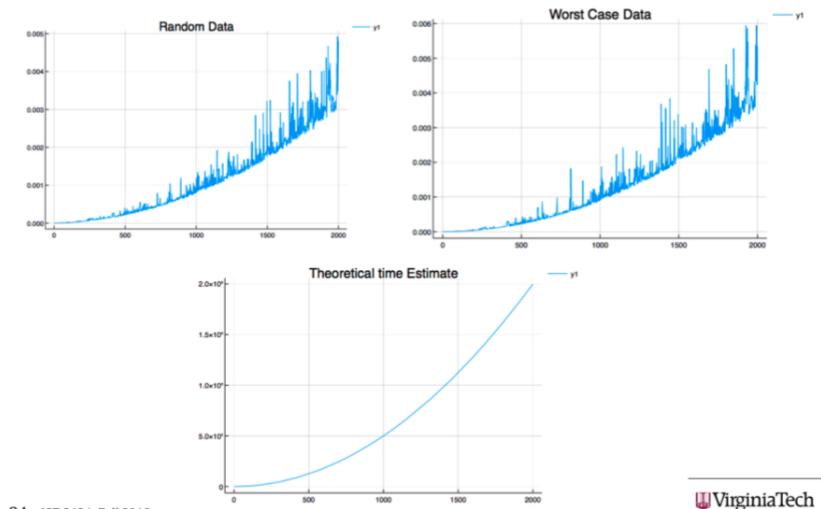


Figure 16.3: Comparison of runtimes of sorting algorithms.

Time elapsed in computer for bubble sort

Figure 16.4: bubble-sort-computational-example²

Theorem 16.7: 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

- $\sum_{i=1}^n i^k = O(n^{k+1})$.

16.3 Problem, instance, size

16.3.1. Problem, instance

Definition 16.8: 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 16.9: 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).

16.3.2. 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.

16.3.3. 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 16.10: 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 16.11: 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.

16.4 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 16.5 .

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.

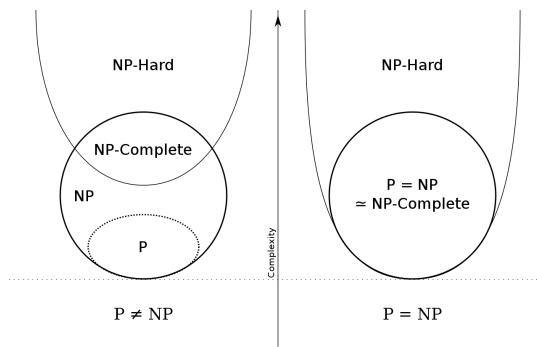


Figure 16.5: Complexity class possibilities. Most academics agree that the case $P \neq NP$ is more likely.

16.4.1. P

Definition 16.12: 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 16.13: Sorting

The sorting problem can be solved in $O(n^2)$ time. Thus, this problem is in *P*

Example 16.14: 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 16.15: 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.

16.4.2. 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 16.16: 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 16.17: 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 16.18

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.

16.4.3. 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 16.19: 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 16.20: 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.

16.4.4. 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.

16.4.5. Examples of Problem Reductions

16.4.5.1. 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.

16.4.5.2. 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.

16.4.5.3. 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.

16.4.6. 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.

16.4.7. 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

$$\text{true} \wedge \text{true} = \text{true} \quad (16.1)$$

$$\text{true} \wedge \text{false} = \text{false} \quad (16.2)$$

$$\text{false} \wedge \text{false} = \text{false} \quad (16.3)$$

$$\text{false} \wedge \text{true} = \text{false} \quad (16.4)$$

$$\text{true} \vee \text{true} = \text{true} \quad (16.5)$$

$$\text{true} \vee \text{false} = \text{true} \quad (16.6)$$

$$\text{false} \vee \text{false} = \text{false} \quad (16.7)$$

$$\text{false} \vee \text{true} = \text{true} \quad (16.8)$$

$$\neg \text{true} = \text{false} \quad (16.9)$$

$$\neg \text{false} = \text{true} \quad (16.10)$$

For example, **Missing code here** A *logical expression* is a sequence of logical operations on variables X_1, \dots, X_n , such that

$$(X_1 \wedge \neg X_2 \vee X_3) \wedge (X_1 \vee \neg X_3) \vee (X_1 \wedge X_2 \wedge X_3). \quad (16.11)$$

A *clause* is a logical expression that only contains the operations \vee and \neg and is not nested (with parentheses), such as

$$X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4. \quad (16.12)$$

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

$$(X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4) \wedge (X_1 \vee X_2 \vee X_3) \wedge (X_2 \vee \neg X_3 \vee \neg X_4 \vee X_5). \quad (16.13)$$

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.

16.5 Problems and Algorithms

We will discuss the following concepts:

- Feasible solutions
- Optimal solutions
- Approximate solutions
- Heuristics
- Exact Algorithms
- Approximation Algorithms
- Complexity class relations

16.5.1. Matching Problem

Definition 16.21: 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' .

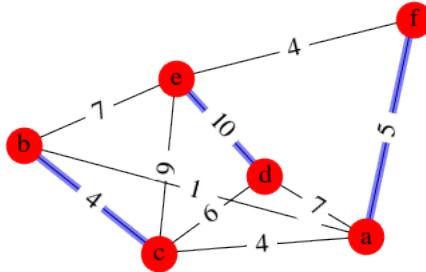
A perfect matching is a matching where every vertex is connected to an edge in E' .

A maximal matching is a matching E' such that there is no matching E'' that strictly contains it.

Figure 16.6: Two possible matchings. On the left, we have a perfect matching (all nodes are matched). On the right, a feasible matching, but not a perfect matching since not all nodes are matched.

Definition 16.22: 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 16.7: graph-for-matching-maximal

16.5.1.1. 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 16.23: 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.

16.5.1.2. Other algorithms to look at

1. Improved Algorithm [[DRAKE2003211](#)]
2. Blossom Algorithm [Wikipedia](#)

16.5.2. Minimum Spanning Tree

Definition 16.24: Spanning Tree

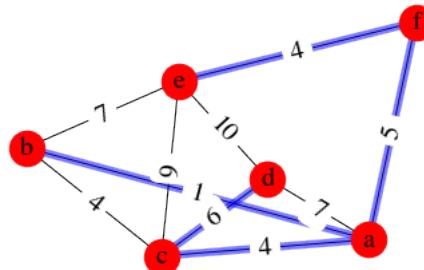
Given a graph $G = (V, E)$, a spanning tree connected, acyclic subgraph T that contains every node in V .

^{© spanning-tree⁵}

Figure 16.8: spanning-tree

Definition 16.25: 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 16.9: spanning-tree-MST

Lemma 16.26: 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](#).

16.5.3. 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

16.5.3.1. Prim's Algorithm

16.5.4. Traveling Salesman Problem

See Section 15.2 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 ??.

16.5.4.1. Nearest Neighbor - Construction Heuristic

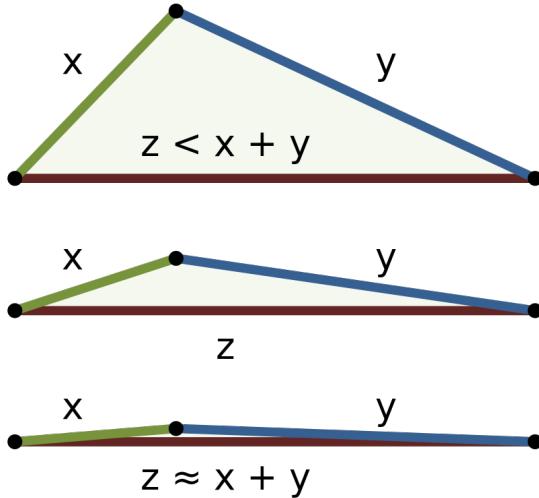
We will discuss heuristics more later in this book. For now, present this construction heuristic as a simple algorithmic example.

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 16.10: wiki/File/triangle_inequality.png

16.5.4.2. Double Spanning Tree - 2-Apx

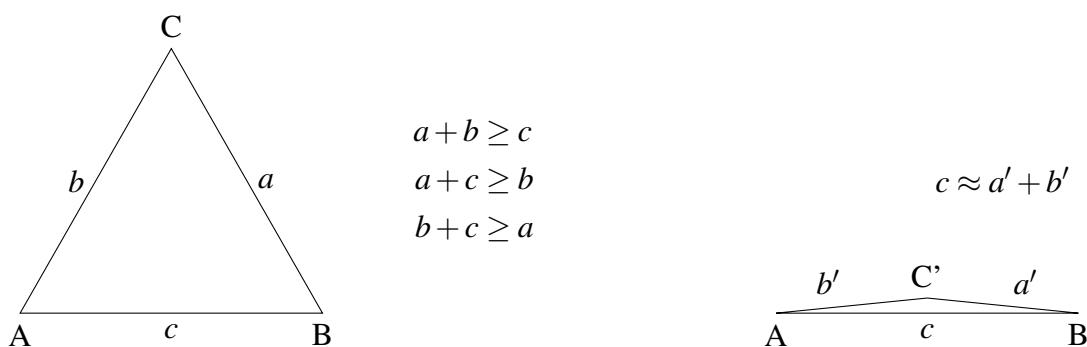
We can use a minimum spanning tree algorithm to find a provably okay solution to the TSP, provided certain properties of the graph are satisfied.

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 16.10

Definition 16.27: 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

$$w(e_{ab}) + w(e_{bc}) \geq w(e_{ac}).$$



⁷wiki/File/triangle_inequality.png, from wiki/File/triangle_inequality.png, wiki/File/triangle_inequality.png, wiki/File/triangle_inequality.png.

Algorithm 8 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.

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

$$w(S) \leq 2w(T),$$

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.

16.5.4.3. 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.

16.6 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

16.6.1. 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.

16.7 Other material for Integer Linear Programming

Recall the problem on lemonade and lemon juice from Chapter 4.6:

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 16.28

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

$$\begin{array}{llllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 \geq 1 \\ & -x_1 & - & x_2 & + & 2x_3 \geq 0 \\ & -x_1 & + & 5x_2 & - & x_3 = 3 \\ & & & & x_3 \leq 0 \\ & x_1, & x_2, & x_3 \geq 0, \end{array}$$

and (P3) is the problem

$$\begin{array}{llllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 \geq 1 \\ & -x_1 & - & x_2 & + & 2x_3 \geq 0 \\ & -x_1 & + & 5x_2 & - & x_3 = 3 \\ & & & & x_3 \geq 1 \\ & x_1, & x_2, & x_3 \geq 0. \end{array}$$

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).
- $\mathbf{x}_{\text{best}}^*$ 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 $\mathbf{x}_{\text{best}}^*$.
- If step 7 is reached, then \mathbf{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:

$$\begin{aligned} \min \quad & x_1 \\ \text{s.t.} \quad & x_1 + 2x_2 - 2x_3 = 1 \\ & x_1, x_2, x_3 \geq 0 \\ & x_1, x_2, x_3 \in \mathbb{Z}. \end{aligned}$$

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	$\mathbf{x}_{\text{best}}^*$	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	active problems	$\mathbf{x}_{\text{best}}^*$	bestbound
1	(P1)	optimal $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{2}{3} \\ \frac{3}{3} \\ \frac{1}{3} \end{bmatrix}$	(P2): $x_3 \leq 0$, (P3): $x_3 \geq 1$	(P2), (P3)	N/A	∞
2	(P2)	infeasible	—	(P3)	N/A $\begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$	∞
3	(P3)	optimal $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$	—	—	$\begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$	1

Exercises

- Suppose that (MP) in Example 16.7 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 $\mathbf{d}^\top \mathbf{A} \in \mathbb{Z}^n$. Prove that every \mathbf{x} satisfying S also satisfies $\mathbf{d}^\top \mathbf{Ax} \geq \lceil \mathbf{d}^\top \mathbf{b} \rceil$. (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

1. An optimal solution to the modified problem is given by $x^* = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.
2. 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.

16.7.1. Other discrete problems

16.7.2. Assignment Problem and the Hungarian Algorithm

Assignment Problem:

Polynomial time (P)

$$\begin{aligned}
 & \min \langle C, X \rangle \\
 \text{s.t. } & \sum_i X_{ij} = 1 \text{ for all } j \\
 & \sum_j X_{ij} = 1 \text{ for all } i \\
 & X_{ij} \in \{0, 1\} \text{ for all } i = 1, \dots, n, j = 1, \dots, m
 \end{aligned} \tag{16.1}$$

This problem is efficiently solved by the Hungarian Algorithm.

16.7.3. History of Computation in Combinatorial Optimization

Book: Computing in Combinatorial Optimization by William Cook, 2019

17. 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.

17.1 Construction Heuristics

17.1.1. 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.

17.1.2. 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.

17.1.3. Insertion Method

Algorithm 9 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.
-

17.2 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.

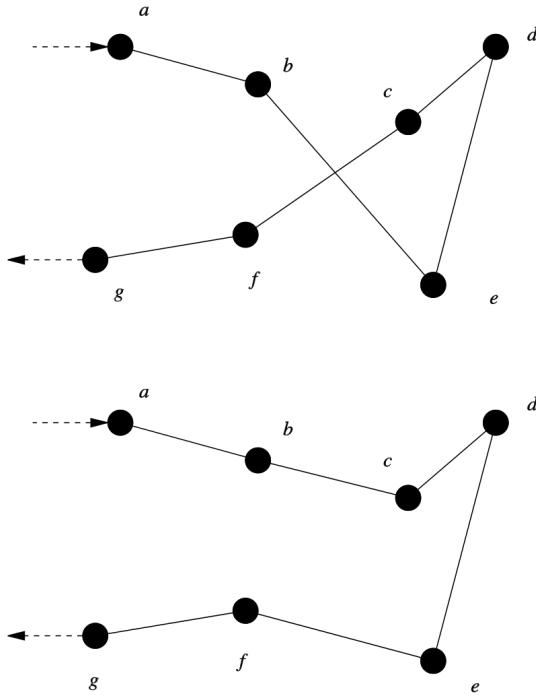
17.2.1. 2-Opt (Subtour Reversal)

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 17.1: 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 17.1

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.

17.2.2. 3-Opt

17.2.3. k -Opt

This is a generalization of 2-Opt and 3-Opt.

17.3 Meta-Heuristics

17.3.1. Hill Climbing (2-Opt for TSP)

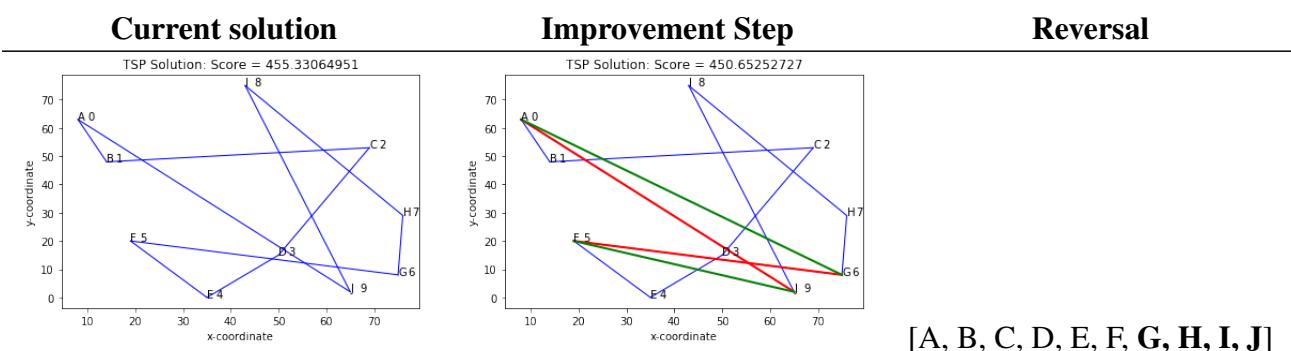
The *Hill Climbing* algorithm finds an improving neighboring solution and climbs in that direction. It continues this process until there is no other neighbor that is improving.

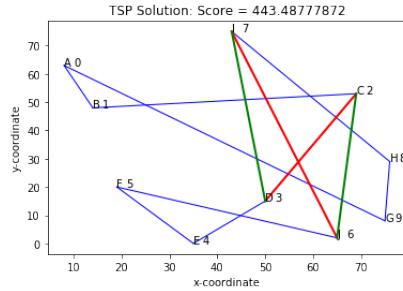
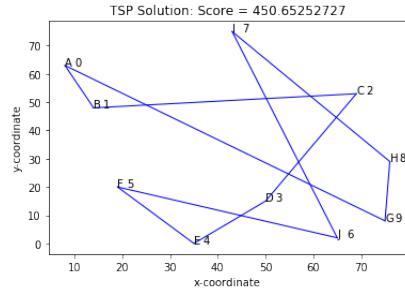
In the context of TSP, we will consider 2-Opt improving moves and the Hill Climbing algorithm for TSP in this case is referred to as the 2-Opt algorithm (also known as the Subtour Reversal Algorithm).

Algorithm 10 Hill Climbing

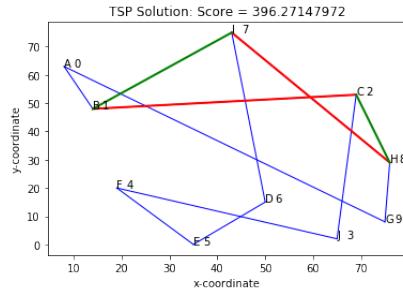
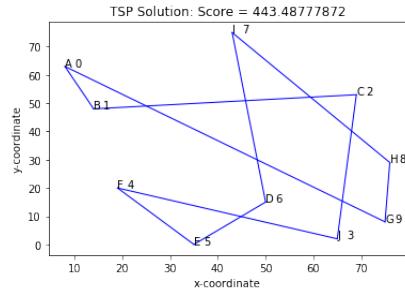
- 1: **Input:** Initial feasible solution, etc.
 - 2: **Output:** Best solution found.
 - 3: Start with an initial feasible solution, label it as the current solution.
 - 4: **repeat**
 - 5: List all neighbors of the current solution.
 - 6: Evaluate all neighbors to find the best one.
 - 7: **if** no 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:

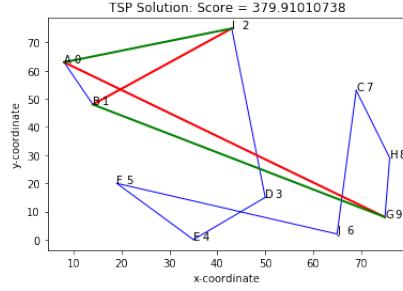
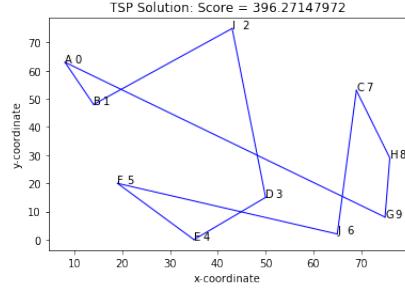




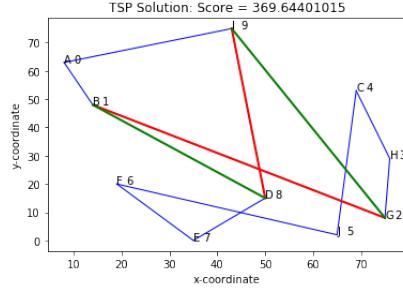
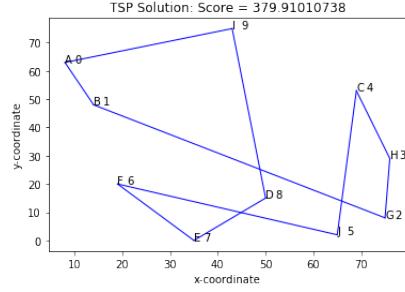
[A, B, C, D, E, F, J, I, H, G]



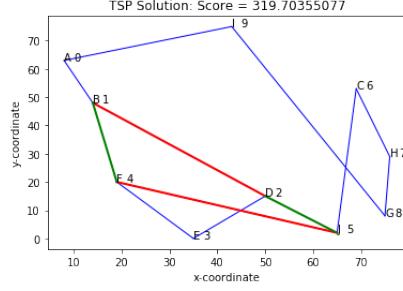
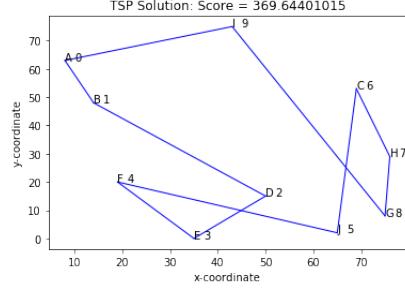
[A, B, C, J, F, E, D, I, H, G]



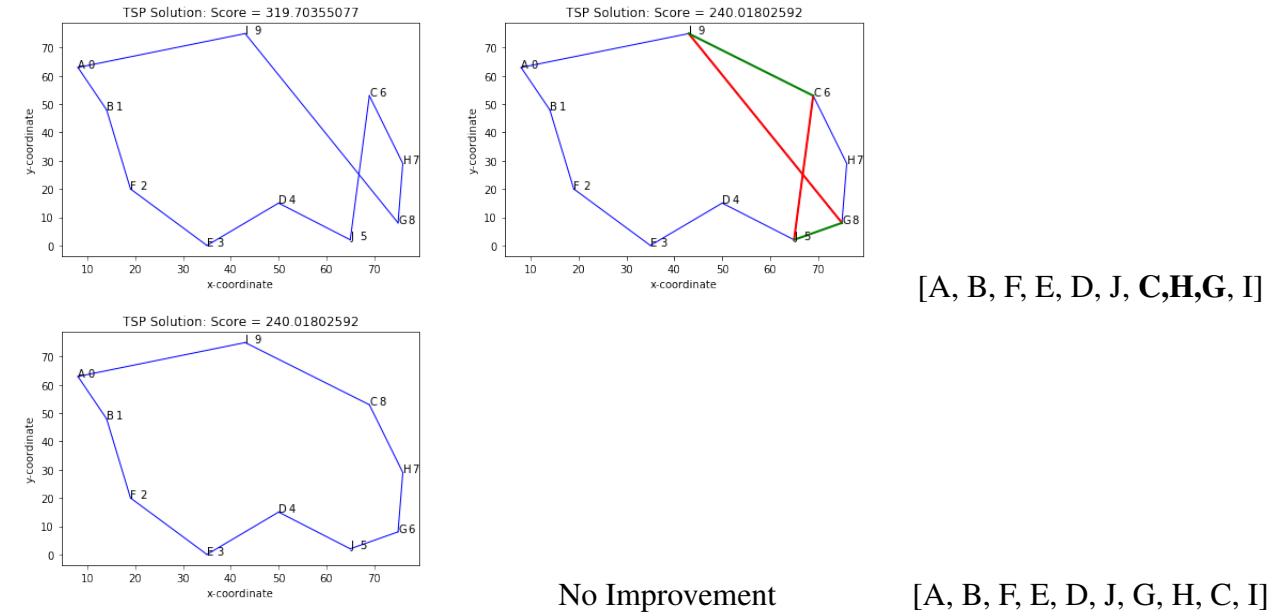
[A, B, I, D, E, F, J, C, H, G]



[A, B, G, H, C, J, F, E, D, I]



[A, B, D, E, F, J, C, H, G, I]



17.3.2. 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 17.2

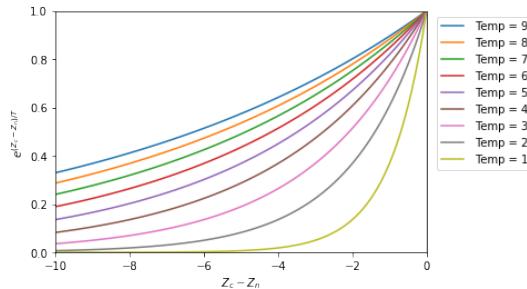
²[simulated_annealing_temperatures.png](#), [simulated_annealing_temperatures.png](#), [simulated_annealing_temperatures.png](#), [simulated_annealing_temperatures.png](#).

Algorithm 11 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 17.2: simulated_annealing_temperatures.png****17.3.3. 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.

17.3.4. 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](#)

17.3.5. 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..

17.3.6. Ant Colony Optimization

[Wikipedia - Ant Colony Optimization](#)

17.4 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 17.2: 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 17.3: 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 17.4: 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

17.4.1. 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.

Part IV

Nonlinear Programming

18. Nonlinear Programming (NLP)

In our last chapter, we covered integer programming, focusing on problems with discrete variables. This chapter will explore *nonlinear (continuous) optimization*, where we deal with continuous variables and now the objective and/or constraints might be nonlinear.

The foundation of continuous optimization is multi-variable calculus. By understanding its basic concepts, we can analyze whether problems are convex or non-convex. This understanding helps in creating efficient algorithms to tackle them.

Our approach in this chapter will be structured. We'll start with key definitions and then move to applications, especially in machine learning. It's worth noting that the problems in this area can vary widely in scale. Some might have millions of variables, while others only a few dozen. This means the choice of algorithm and approach can vary significantly based on the problem.

Our goals for this chapter are:

- Seeing practical applications of continuous optimization.
- Understanding the structure of problems.
- Recognizing the strengths and limits of different algorithms.
- Implementing solutions using Python tools like `scipy.optimize` and `scikitlearn`.

In the next chapter, we'll combine the continuous optimization techniques from this chapter with the integer programming methods from the last chapter. This combination will help us tackle more advanced problems and techniques.

We will provide a number of proofs throughout this chapter for completeness, although, memorizing these proofs may not be essential to fulfilling the outcomes of this chapter.

18.1 Definitions and Theorems

We consider a general mathematical formulation for the optimization problem. We will begin with the context of unconstrained optimization and then discuss later how some of the theorems can be altered to manage constrained versions of the problem.

Definition 18.1: Unconstrained Minimization

Consider a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. The problem is given by:

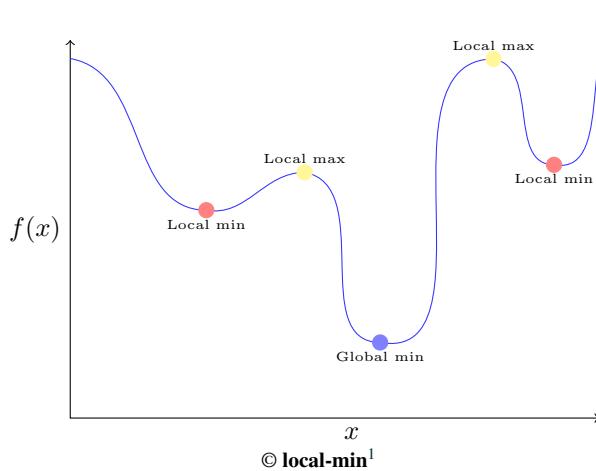
$$\min_{x \in \mathbb{R}^n} f(x)$$

Note that we may change any maximization problem to a minimization problem by simply minimizing the function $g(x) := -f(x)$.

Definition 18.2: Global and local optima

For Problem 18.1, a vector x^* is a

- global minimizer if $f(x^*) \leq f(x)$ for all $x \in \mathbb{R}^n$.
- local minimizer if $f(x^*) \leq f(x)$ for all x satisfying $\|x - x^*\| \leq \varepsilon$ for some $\varepsilon > 0$.
- strict local minimizer if $f(x^*) < f(x)$ for all $x \neq x^*$ satisfying $\|x - x^*\| \leq \varepsilon$ for some $\varepsilon > 0$.

**Theorem 18.3: Attaining a minimum**

Let S be a nonempty set that is closed and bounded. Suppose that $f : S \rightarrow \mathbb{R}$ is continuous. Then the problem $\min\{f(x) : x \in S\}$ attains its minimum.

Proof. By the Extreme Value Theorem, if a function f is continuous on a closed and bounded interval, then f attains both its maximum and minimum values, i.e., there exist points $c, d \in S$ such that:

$$f(c) \leq f(x) \leq f(d) \quad \text{for all } x \in S$$

Given that S is nonempty, closed, and bounded, and f is continuous on S , it follows from the theorem that f attains its minimum on S .

¹local-min, from local-min. local-min, local-min.

Thus, there exists an $x^* \in S$ such that:

$$f(x^*) \leq f(x) \quad \text{for all } x \in S$$

This means that the problem $\min\{f(x) : x \in S\}$ attains its minimum at x^* .

This completes the proof. ♠

18.1.1. Calculus: Derivatives

We generalize the notion a derivative from single variable calculus to multi-variable calculus.

Definition 18.4: Partial Derivative

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a function defined on an open set containing a point $\mathbf{a} = (a_1, a_2, \dots, a_n)$. The partial derivative of f with respect to its i -th variable, x_i , at the point \mathbf{a} is defined as:

$$\frac{\partial f}{\partial x_i}(\mathbf{a}) = \lim_{h \rightarrow 0} \frac{f(a_1, \dots, a_{i-1}, a_i + h, a_{i+1}, \dots, a_n) - f(a_1, \dots, a_n)}{h}$$

provided the limit exists. It represents the rate of change of the function with respect to the variable x_i while keeping all other variables constant.

We can then combine these into a vector called the *Gradient*.

Definition 18.5: Gradient

Given a scalar-valued differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the gradient of f at a point $x \in \mathbb{R}^n$ is a vector of its first order partial derivatives. It is denoted by $\nabla f(x)$ and is given by:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

The gradient points in the direction of the steepest ascent of the function at the point x .

Definition 18.6: Critical Point

A critical point is a point \bar{x} where $\nabla f(\bar{x}) = 0$.

Theorem 18.7

Suppose that $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is differentiable. If $\min\{f(x) : x \in \mathbb{R}^n\}$ has an optimizer x^* , then x^* is a critical point of f (i.e., $\nabla f(x^*) = 0$).

Proof. Suppose x^* is an optimizer of f but $\nabla f(x^*) \neq 0$. Without loss of generality, let's assume that the first component of $\nabla f(x^*)$ is positive (the argument for a negative component is similar).

Then, the directional derivative of f at x^* in the direction of the standard basis vector e_1 (which is the vector with a 1 in the first position and 0 elsewhere) is positive. Formally, this is:

$$D_{e_1}f(x^*) = \nabla f(x^*) \cdot e_1 > 0$$

This means that for sufficiently small $t > 0$, we have:

$$f(x^* + te_1) < f(x^*)$$

This contradicts the assumption that x^* is an optimizer of f , since we've found a point $x^* + te_1$ where f takes a smaller value than at x^* .

Therefore, our initial assumption that $\nabla f(x^*) \neq 0$ must be incorrect, and we conclude that $\nabla f(x^*) = 0$.

This completes the proof. ♠

18.1.2. Calculus: Second derivatives

We can also generalize multi-variate valuva

Definition 18.8: Hessian

For a scalar-valued function $f: \mathbb{R}^n \rightarrow \mathbb{R}$ that has continuous second order partial derivatives, the Hessian matrix of f at a point $x \in \mathbb{R}^n$ is a square matrix of its second order partial derivatives. It is denoted by $\nabla^2 f(x)$ or $H_f(x)$ and is defined as:

$$H_f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

The Hessian provides information about the local curvature of the function at the point x .

Definition 18.9: Positive Semidefinite

A square matrix A is said to be positive semidefinite if, for all non-zero vectors $x \in \mathbb{R}^n$, the quadratic form $x^\top Ax$ is non-negative, i.e.,

$$x^\top Ax \geq 0$$

for all $x \in \mathbb{R}^n$. Equivalently, all the eigenvalues of A are non-negative.

18.1.3. Multivariate Calculus Examples

Example 18.10: 2D Function: Gradient and Hessian

Consider the function:

$$f(x, y) = x^2 + 2xy + y^2$$

1. Gradient of f :

$$\nabla f(x, y) = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x + 2y \\ 2x + 2y \end{bmatrix}$$

2. Hessian of f :

$$H_f(x, y) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

The eigenvalues of this Hessian are 4 and 0, both non-negative. Therefore, the Hessian is positive semidefinite.

Example 18.11: Non-Quadratic 2D Function: Gradient and Hessian

Consider the function:

$$h(x, y) = x^3y + xy^2$$

1. Gradient of h :

$$\nabla h(x, y) = \begin{bmatrix} \frac{\partial h}{\partial x} \\ \frac{\partial h}{\partial y} \end{bmatrix} = \begin{bmatrix} 3x^2y + y^2 \\ x^3 + 2xy \end{bmatrix}$$

2. Hessian of h :

$$H_h(x, y) = \begin{bmatrix} \frac{\partial^2 h}{\partial x^2} & \frac{\partial^2 h}{\partial x \partial y} \\ \frac{\partial^2 h}{\partial y \partial x} & \frac{\partial^2 h}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 6xy + 2y & 3x^2 + 2x \\ 3x^2 + 2x & 2x \end{bmatrix}$$

Note that the entries of the Hessian are not constant and vary depending on the point (x, y) . The positive semidefiniteness of the Hessian will vary depending on the values of x and y , and cannot be determined globally for this function without further analysis.

Example 18.12: 3D Function: Gradient and Hessian

Consider the function:

$$g(x, y, z) = x^2 + y^2 + z^2 + 2xz$$

1. Gradient of g :

$$\nabla g(x, y, z) = \begin{bmatrix} \frac{\partial g}{\partial x} \\ \frac{\partial g}{\partial y} \\ \frac{\partial g}{\partial z} \end{bmatrix} = \begin{bmatrix} 2x + 2z \\ 2y \\ 2z + 2x \end{bmatrix}$$

2. Hessian of g :

$$H_g(x, y, z) = \begin{bmatrix} \frac{\partial^2 g}{\partial x^2} & \frac{\partial^2 g}{\partial x \partial y} & \frac{\partial^2 g}{\partial x \partial z} \\ \frac{\partial^2 g}{\partial y \partial x} & \frac{\partial^2 g}{\partial y^2} & \frac{\partial^2 g}{\partial y \partial z} \\ \frac{\partial^2 g}{\partial z \partial x} & \frac{\partial^2 g}{\partial z \partial y} & \frac{\partial^2 g}{\partial z^2} \end{bmatrix} = \begin{bmatrix} 2 & 0 & 2 \\ 0 & 2 & 0 \\ 2 & 0 & 2 \end{bmatrix}$$

The eigenvalues of this Hessian are approximately 4.73, 2, and -0.73. Since one eigenvalue is negative, the Hessian is not positive semidefinite.

Lemma 18.13: Gradient of Quadratic in Matrix Notation

Given a multivariate quadratic function in the form:

$$f(x) = x^\top Qx$$

where $x = [x_1, x_2, \dots, x_n]^\top$ and Q is an $n \times n$ symmetric matrix. The gradient is given by

$$\nabla f(x) = 2Qx$$

Proof. We can expand the function in terms of individual elements as:

$$f(x) = \sum_{i=1}^n \sum_{j=1}^n Q_{ij} x_i x_j$$

Now, let's differentiate with respect to x_k , for some $k \in \{1, 2, \dots, n\}$:

$$\frac{\partial f}{\partial x_k} = \frac{\partial}{\partial x_k} \left(\sum_{i=1}^n \sum_{j=1}^n Q_{ij} x_i x_j \right)$$

For terms where $i = k$:

$$\frac{\partial}{\partial x_k} (Q_{kj} x_k x_j) = Q_{kj} x_j$$

And for terms where $j = k$:

$$\frac{\partial}{\partial x_k} (Q_{ik} x_i x_k) = Q_{ik} x_i$$

Given that Q is symmetric, we have $Q_{kj} = Q_{jk}$. So, the sum of the two derivatives above for any given k is:

$$Q_{kj}x_j + Q_{jk}x_i = 2Q_{kj}x_j$$

Thus, the k -th component of the gradient vector is:

$$\frac{\partial f}{\partial x_k} = 2 \sum_{j=1}^n Q_{kj}x_j$$

In matrix notation, the gradient is:

$$\nabla f(x) = 2Qx$$



18.1.4. Taylor's Theorem

Theorem 18.14: Taylor's Theorem: Univariate Case

Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a function such that f and its first n derivatives are continuous on an interval containing c and x , and its $(n+1)$ -th derivative exists on this interval. Then, for each x in the interval, there exists a point z between c and x such that:

$$f(x) = f(c) + f'(c)(x - c) + \frac{f''(c)}{2!}(x - c)^2 + \cdots + \frac{f^{(n)}(c)}{n!}(x - c)^n + R_n(x)$$

where the remainder term is given by:

$$R_n(x) = \frac{f^{(n+1)}(z)}{(n+1)!}(x - c)^{n+1}$$

This can be generalized to the multivariate case. We present here just the version up to quadratic terms since this tends to be the most used version of the result.

Theorem 18.15: Taylor's Theorem: Multivariate Case (Quadratic Terms)

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a scalar-valued function with continuous partial derivatives up to order 3 in a neighborhood of a point \mathbf{a} in \mathbb{R}^n . Then, for a vector \mathbf{h} in this neighborhood, the function can be expressed as:

$$f(\mathbf{a} + \mathbf{h}) = f(\mathbf{a}) + \nabla f(\mathbf{a})^\top \mathbf{h} + \frac{1}{2} \mathbf{h}^\top \nabla^2 f(\mathbf{a}) \mathbf{h} + R(\mathbf{h})$$

where the remainder term is:

$$R(\mathbf{h}) = O(\|\mathbf{h}\|^3)$$

and $R(\mathbf{h})$ depends on the third order derivatives of f , which are evaluated at some point between \mathbf{a} and $\mathbf{a} + \mathbf{h}$.

Lemma 18.16: 2nd Derivative and Critical Points

If $f''(a) > 0$ at a critical point a of a function f , then a is a local minimum of f .

Proof. A point a is a critical point of f if and only if $f'(a) = 0$. Using Taylor's theorem, we can expand f about the point $x = a$:

$$f(x) = f(a) + f'(a)(x - a) + \frac{f''(c)}{2!}(x - a)^2$$

for some c between a and x .

Given that $f'(a) = 0$, the expansion reduces to:

$$f(x) = f(a) + \frac{f''(c)}{2!}(x - a)^2$$

Now, if $f''(a) > 0$, then by the continuity of the second derivative, there exists an open interval $(a - \delta, a + \delta)$ around a such that for all x in this interval (except possibly at $x = a$), $f''(x) > 0$.

For x in this interval, the term $(x - a)^2$ is always non-negative. Given that $f''(c) > 0$ for c between a and x , we deduce that:

$$f(x) - f(a) = \frac{f''(c)}{2}(x - a)^2 > 0$$

This implies that $f(x) > f(a)$ for x in $(a - \delta, a + \delta)$, except possibly at $x = a$. Therefore, a is a local minimum of f . ♠

Now let's look at the multivariate version of this.

Lemma 18.17: Hessian and Local Minima

Let $f: \mathbb{R}^n \rightarrow \mathbb{R}$ be a twice continuously differentiable function. If $\nabla f(\mathbf{a}) = 0$ (i.e., \mathbf{a} is a critical point) and the Hessian matrix $\nabla^2 f(\mathbf{a})$ is positive semidefinite at \mathbf{a} , then \mathbf{a} is a local minimum of f .

Proof. At the point \mathbf{a} , since $\nabla f(\mathbf{a}) = 0$, the first order Taylor expansion of f about \mathbf{a} is simply $f(\mathbf{a})$.

The second order Taylor expansion of f about \mathbf{a} is:

$$f(\mathbf{x}) \approx f(\mathbf{a}) + \frac{1}{2}(\mathbf{x} - \mathbf{a})^\top \nabla^2 f(\mathbf{a})(\mathbf{x} - \mathbf{a})$$

Given that the Hessian $\nabla^2 f(\mathbf{a})$ is positive semidefinite, for any vector \mathbf{v} , we have:

$$\mathbf{v}^\top \nabla^2 f(\mathbf{a}) \mathbf{v} \geq 0$$

Choosing $\mathbf{v} = \mathbf{x} - \mathbf{a}$, we get:

$$(\mathbf{x} - \mathbf{a})^\top \nabla^2 f(\mathbf{a}) (\mathbf{x} - \mathbf{a}) \geq 0$$

Thus, for \mathbf{x} near \mathbf{a} :

$$f(\mathbf{x}) \geq f(\mathbf{a})$$

This shows that $f(\mathbf{a})$ is less than or equal to the values of f near \mathbf{a} , and hence \mathbf{a} is a local minimum of f .



18.1.5. Constrained Minimization and the KKT Conditions

Definition 18.18: Constrained Minimization

Given functions $f: \mathbb{R}^n \rightarrow \mathbb{R}$ and $g_i: \mathbb{R}^n \rightarrow \mathbb{R}$ for $i = 1, \dots, m$. The problem is formulated as:

$$\begin{array}{ll} \min_{x \in \mathbb{R}^n} & f(x) \\ \text{subject to} & g_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \end{array}$$

The KKT conditions use the augmented Lagrangian problem to describe sufficient conditions for optimality of a convex program.

KKT Conditions for Optimality:

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

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & g_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d \end{array} \tag{18.1}$$

Given $(\bar{x}, \bar{\lambda})$ with $\bar{x} \in \mathbb{R}^d$ and $\bar{\lambda} \in \mathbb{R}^m$, if the KKT conditions hold, then \bar{x} is optimal for the convex programming problem.

The KKT conditions are

1. (Stationary).

$$-\nabla f(\bar{x}) = \sum_{i=1}^m \bar{\lambda}_i \nabla g_i(\bar{x}) \quad (18.2)$$

2. (Complimentary Slackness).

$$\bar{\lambda}_i g_i(\bar{x}) = 0 \text{ for } i = 1, \dots, m \quad (18.3)$$

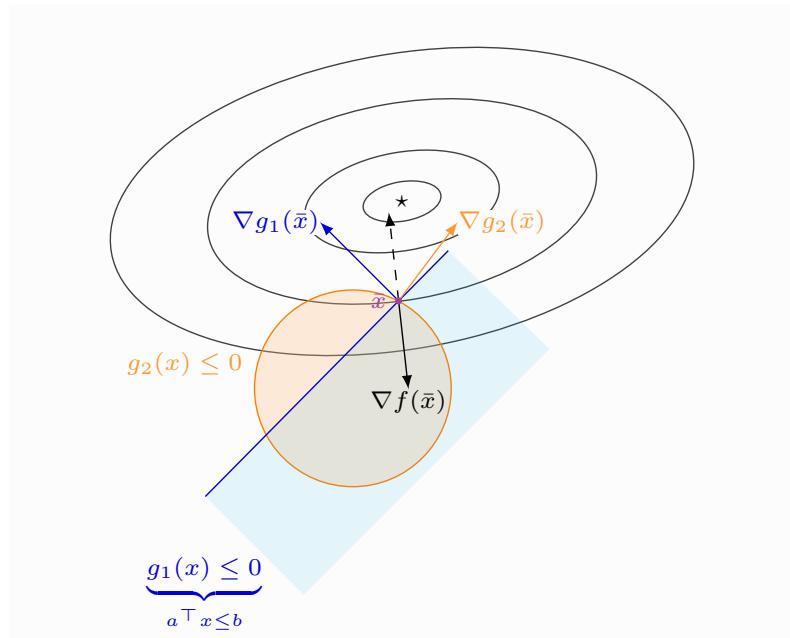
3. (Primal Feasibility).

$$g_i(\bar{x}) \leq 0 \text{ for } i = 1, \dots, m \quad (18.4)$$

4. (Dual Feasibility).

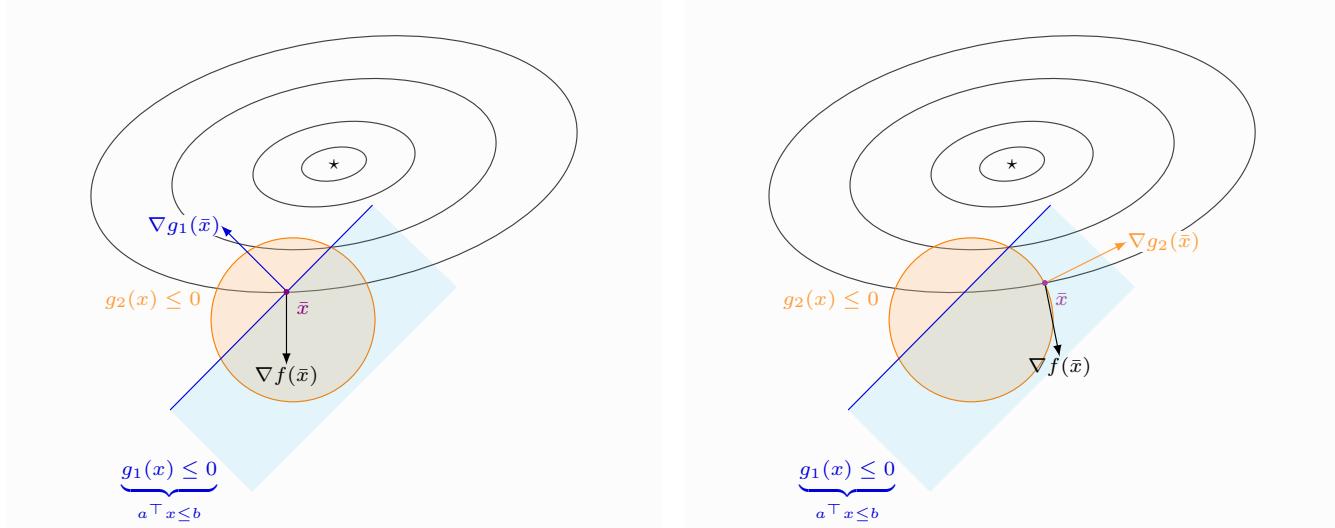
$$\bar{\lambda}_i \geq 0 \text{ for } i = 1, \dots, m \quad (18.5)$$

If certain properties are true of the convex program, then every optimizer has these properties. In particular, this holds for Linear Programming.



© tikz/kkt-optimal²

²tikz/kkt-optimal, from tikz/kkt-optimal. tikz/kkt-optimal, tikz/kkt-optimal.

© tikz/kkt-non-optimal1³© tikz/kkt-non-optimal2⁴

tikz/kkt-non-optimal1, from tikz/kkt-non-optimal1.
 tikz/kkt-non-optimal1, tikz/kkt-non-optimal1.

tikz/kkt-non-optimal2, from tikz/kkt-non-optimal2.
 tikz/kkt-non-optimal2, tikz/kkt-non-optimal2.

19. NLP Algorithms

19.1 Algorithms Introduction

We will begin with unconstrained optimization and consider several different algorithms based on what is known about the objective function. In particular, we will consider the cases where we use

- Only function evaluations (also known as *derivative free optimization*),
- Function and gradient evaluations,
- Function, gradient, and hessian evaluations.

We will first look at these algorithms and their convergence rates in the 1-dimensional setting and then extend these results to higher dimensions.

19.2 1-Dimensional Algorithms

We suppose that we solve the problem

$$\min f(x) \tag{19.1}$$

$$x \in [a, b]. \tag{19.2}$$

That is, we minimize the univariate function $f(x)$ on the interval $[l, u]$.

For example,

$$\min(x^2 - 2)^2 \tag{19.3}$$

$$0 \leq x \leq 10. \tag{19.4}$$

Note, the optimal solution lies at $x^* = \sqrt{2}$, which is an irrational number. Since we will consider algorithms using floating point precision, we will look to return a solution \bar{x} such that $\|x^* - \bar{x}\| < \varepsilon$ for some small $\varepsilon > 0$, for instance, $\varepsilon = 10^{-6}$.

Golden Section Search

A function $f : [a, b] \rightarrow \mathbb{R}$ satisfies the *unimodal property* if it has exactly one local minimum and is monotonic on either side of the minimizer. In other words, f decreases from a to its minimizer x^* , then increases up to b (see Figure 22.1). The *golden section search* method optimizes a unimodal function f by iteratively defining smaller and smaller intervals containing the unique minimizer x^* . This approach is especially useful if the function's derivative does not exist, is unknown, or is very costly to compute.

By definition, the minimizer x^* of f must lie in the interval $[a, b]$. To shrink the interval around x^* , we test the following strategically chosen points.

$$\tilde{a} = b - \frac{b-a}{\varphi} \quad \tilde{b} = a + \frac{b-a}{\varphi}$$

Here $\varphi = \frac{1+\sqrt{5}}{2}$ is the *golden ratio*. At each step of the search, $[a, b]$ is refined to either $[a, \tilde{b}]$ or $[\tilde{a}, b]$, called the *golden sections*, depending on the following criteria.

If $f(\tilde{a}) < f(\tilde{b})$, then since f is unimodal, it must be increasing in a neighborhood of \tilde{b} . The unimodal property also guarantees that f must be increasing on $[\tilde{b}, b]$ as well, so $x^* \in [a, \tilde{b}]$ and we set $b = \tilde{b}$. By similar reasoning, if $f(\tilde{a}) > f(\tilde{b})$, then $x^* \in [\tilde{a}, b]$ and we set $a = \tilde{a}$. If, however, $f(\tilde{a}) = f(\tilde{b})$, then the unimodality of f does not guarantee anything about where the minimizer lies. Assuming either $x^* \in [a, \tilde{b}]$ or $x^* \in [\tilde{a}, b]$ allows the iteration to continue, but the method is no longer guaranteed to converge to the local minimum.

At each iteration, the length of the search interval is divided by φ . The method therefore converges linearly, which is somewhat slow. However, the idea is simple and each step is computationally inexpensive.

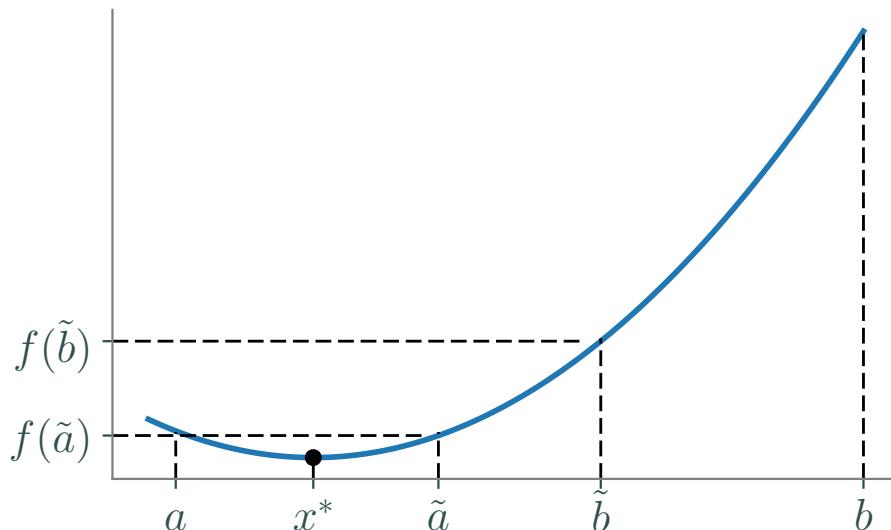


Figure 19.1: The unimodal $f : [a, b] \rightarrow \mathbb{R}$ can be minimized with a golden section search. For the first iteration, $f(\tilde{a}) < f(\tilde{b})$, so $x^* \in [a, \tilde{b}]$. New values of \tilde{a} and \tilde{b} are then calculated from this new, smaller interval.

Algorithm 12 The Golden Section Search

```

1: procedure GOLDEN_SECTION( $f$ ,  $a$ ,  $b$ ,  $\text{tol}$ ,  $\text{maxiter}$ )
2:    $x_0 \leftarrow (a + b)/2$             $\triangleright$  Set the initial minimizer approximation as the interval midpoint.
3:    $\varphi = (1 + \sqrt{5})/2$ 
4:   for  $i = 1, 2, \dots, \text{maxiter}$  do            $\triangleright$  Iterate only  $\text{maxiter}$  times at most.
5:      $c \leftarrow (b - a)/\varphi$ 
6:      $\tilde{a} \leftarrow b - c$ 
7:      $\tilde{b} \leftarrow a + c$ 
8:     if  $f(\tilde{a}) \leq f(\tilde{b})$  then            $\triangleright$  Get new boundaries for the search interval.
9:        $b \leftarrow \tilde{b}$ 
10:    else
11:       $a \leftarrow \tilde{a}$ 
12:     $x_1 \leftarrow (a + b)/2$             $\triangleright$  Set the minimizer approximation as the interval midpoint.
13:    if  $|x_0 - x_1| < \text{tol}$  then            $\triangleright$  Stop iterating if the approximation stops changing enough.
14:      break
15:     $x_0 \leftarrow x_1$ 
16:  return  $x_1$ 

```

Problem 19.1: Implement golden search.

rite a function that accepts a function $f : \mathbb{R} \rightarrow \mathbb{R}$, interval limits a and b , a stopping tolerance tol , and a maximum number of iterations maxiter . Use Algorithm 13 to implement the golden section search. Return the approximate minimizer x^* , whether or not the algorithm converged (true or false), and the number of iterations computed.

Test your function by minimizing $f(x) = e^x - 4x$ on the interval $[0, 3]$, then plotting the function and the computed minimizer together. Also compare your results to SciPy's golden section search, `scipy.optimize.golden()`.

```

>>> from scipy import optimize as opt
>>> import numpy as np

>>> f = lambda x : np.exp(x) - 4*x
>>> opt.golden(f, brack=(0,3), tol=.001)
1.3862578679031485          # ln(4) is the minimizer.

```

19.2.1. Golden Search Method - Derivative Free Algorithm

Resources

- *Youtube! - Golden Section Search Method*

Suppose that $f(x)$ is unimodal on the interval $[a, b]$, that is, it is a continuous function that has a single minimizer on the interval.

Without any extra information, our best guess for the optimizer is $\bar{x} = \frac{a+b}{2}$ with a maximum error of $\varepsilon = \frac{b-a}{2}$. Our goal is to reduce the size of the interval where we know x^* to be, and hence improve our best guess and the maximum error of our guess.

Now we want to choose points in the interior of the interval to help us decide where the minimizer is. Let x_1, x_2 such that

$$a < x_2 < x_1 < b.$$

Next, we evaluate the function at these four points. Using this information, we would like to argue a smaller interval in which x^* is contained. In particular, since f is unimodal, it must hold that

1. $x^* \in [a, x_2]$ if $f(x_1) \leq f(x_2)$,
2. $x^* \in [x_1, b]$ if $f(x_2) < f(x_1)$,

After comparing these function values, we can reduce the size of the interval and hence reduce the region where we think x^* is.

We will now discuss how to chose x_1, x_2 in a way that we can

1. Reuse function evaluations,
2. Have a constant multiplicative reduction in the size of the interval.

We consider the picture:

To determine the best d , we want to decrease by a constant factor. Hence, we decrease be a factor γ , which we will see is the golden ration (GR). To see this, we assume that $(b - a) = 1$, and ask that $d = \gamma$. Thus, $x_1 - a = \gamma$ and $b - x_2 = \gamma$. If we are in case 1, then we cut off $b - x_1 = 1 - \gamma$. Now, if we iterate and do this again, we will have an initial length of γ and we want to cut off the interval $x_2 - x_1$ with this being a proportion of $(1 - \gamma)$ of the remaining length. Hence, the second time we will cut of $(1 - \gamma)\gamma$, which we set as the length between x_1 and x_2 .

Considering the geometry, we have

$$\text{length } a \text{ to } x_1 + \text{length } x_2 \text{ to } b = \text{total length} + \text{length } x_2 \text{ to } x_1$$

hence

$$\gamma + \gamma = 1 + (1 - \gamma)\gamma.$$

Simplifying, we have

$$\gamma^2 + \gamma - 1 = 0.$$

Applying the quadratic formula, we see

$$\gamma = \frac{-1 \pm \sqrt{5}}{2}.$$

Since we want $\gamma > 0$, we take

$$\gamma = \frac{-1 + \sqrt{5}}{2} \approx 0.618$$

This is exactly the Golden Ratio (or, depending on the definition, the golden ration minus 1).

19.2.1.1. Example:

We can conclude that the optimal solution is in $[1.4, 3.8]$, so we would guess the midpoint $\bar{x} = 2.6$ as our approximate solution with a maximum error of $\epsilon = 1.2$.

Convergence Analysis of Golden Search Method:

After t steps of the Golden Search Method, the interval in question will be of length

$$(b - a)(GR)^t \approx (b - a)(0.618)^t$$

Hence, by guessing the midpoint, our worst error could be

$$\frac{1}{2}(b - a)(0.618)^t.$$

19.2.2. Bisection Method - 1st Order Method (using Derivative)

19.2.2.1. Minimization Interpretation

Assumptions: f is convex, differentiable

We can look for a minimizer of the function $f(x)$ on the interval $[a, b]$.

19.2.2.2. Root finding Interpretation

Instead of minimizing, we can look for a root of $f'(x)$. That is, find x such that $f'(x) = 0$.

Assumptions: $f'(a) < 0 < f'(b)$, OR, $f'(b) < 0 < f'(a)$. f' is continuous

The goal is to find a root of the function $f'(x)$ on the interval $[a, b]$. If f is convex, then we know that this root is indeed a global minimizer.

Note that if f is convex, it only makes sense to have the assumption $f'(a) < 0 < f'(b)$.

Convergence Analysis of Bisection Method:

After t steps of the Bisection Method, the interval in question will be of length

$$(b-a) \left(\frac{1}{2}\right)^t.$$

Hence, by guessing the midpoint, our worst error could be

$$\frac{1}{2}(b-a) \left(\frac{1}{2}\right)^t.$$

19.2.3. Gradient Descent - 1st Order Method (using Derivative)

Input: $f(x)$, $\nabla f(x)$, initial guess x^0 , learning rate α , tolerance ε

Output: An approximate solution x

1. Set $t = 0$
2. While $\|f(x^t)\|_2 > \varepsilon$:
 - (a) Set $x^{t+1} \leftarrow x^t - \alpha \nabla f(x^t)$.
 - (b) Set $t \leftarrow t + 1$.
3. Return x^t .

Newton's Method

Newton's method is an important root-finding algorithm that can also be used for optimization. Given $f : \mathbb{R} \rightarrow \mathbb{R}$ and a good initial guess x_0 , the sequence $(x_k)_{k=1}^{\infty}$ generated by the recursive rule

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

converges to a point \bar{x} satisfying $f(\bar{x}) = 0$. The first-order necessary conditions from elementary calculus state that if f is differentiable, then its derivative evaluates to zero at each of its local minima and maxima. Therefore using Newton's method to find the zeros of f' is a way to identify potential minima or maxima of f . Specifically, starting with an initial guess x_0 , set

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{19.5}$$

and iterate until $|x_k - x_{k-1}|$ is satisfactorily small. Note that this procedure does not use the actual function f at all, but it requires many evaluations of its first and second derivatives. As a result, Newton's method converges in few iterations, but it can be computationally expensive.

Each step of (22.1) can be thought of approximating the objective function f by a quadratic function q and finding its unique extrema. That is, we first approximate f with its second-degree Taylor polynomial centered at x_k .

$$q(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2$$

This quadratic function satisfies $q(x_k) = f(x_k)$ and matches f fairly well close to x_k . Thus the optimizer of q is a reasonable guess for an optimizer of f . To compute that optimizer, solve $q'(x) = 0$.

$$0 = q'(x) = f'(x_k) + f''(x_k)(x - x_k) \implies x = x_k - \frac{f'(x_k)}{f''(x_k)}$$

This agrees with (22.1) using x_{k+1} for x . See Figure 22.2.

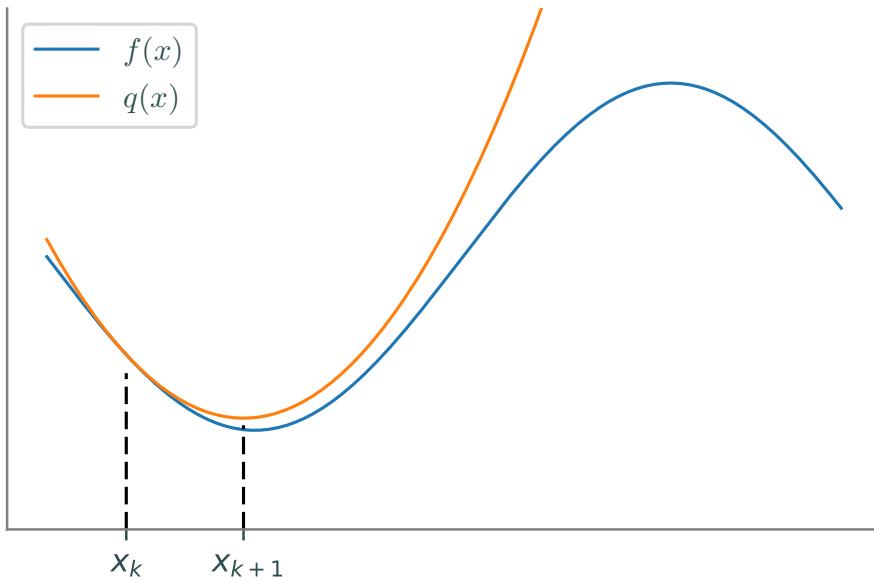


Figure 19.2: A quadratic approximation of f at x_k . The minimizer x_{k+1} of q is close to the minimizer of f .

Newton's method for optimization works well to locate minima when $f''(x) > 0$ on the entire domain. However, it may fail to converge to a minimizer if $f''(x) \leq 0$ for some portion of the domain. If f is not unimodal, the initial guess x_0 must be sufficiently close to a local minimizer x^* in order to converge.

Problem 19.2: Newton's method for optimization

Let $f : \mathbb{R} \rightarrow \mathbb{R}$. Write a function that accepts f' , f'' , a starting point x_0 , a stopping tolerance tol , and a maximum number of iterations $maxiter$. Implement Newton's method using (22.1) to locate a local optimizer. Return the approximate optimizer, whether or not the algorithm converged, and the number of iterations computed.

Test your function by minimizing $f(x) = x^2 + \sin(5x)$ with an initial guess of $x_0 = 0$. Compare your results to `scipy.optimize.newton()`, which implements the root-finding version of Newton's method.

```
>>> df = lambda x : 2*x + 5*np.cos(5*x)
>>> d2f = lambda x : 2 - 25*np.sin(5*x)
>>> opt.newton(df, x0=0, fprime=d2f, tol=1e-10, maxiter=500)
-1.4473142236328096
```

Note that other initial guesses can yield different minima for this function.

CONVERGENCE OF NEWTON'S METHOD We consider the function $f(x) = e^x + e^{-x}$.

	x	$f(x)$	$f'(x)$	$ x^k - x^{k-1} $	$\frac{ x^k - x^{k-1} }{ x^{k-1} - x^{k-2} }$
0	6.100000e+00	445.860013	4.458555e+02	NaN	NaN
0	5.100010e+00	164.029654	1.640175e+02	9.999899e-01	NaN
0	4.100084e+00	60.361952	6.032881e+01	9.999257e-01	9.999357e-01
0	3.100633e+00	22.257038	2.216700e+01	9.994509e-01	9.995252e-01
0	2.104679e+00	8.326354	8.082584e+00	9.959545e-01	9.965016e-01
0	1.133956e+00	3.429685	2.786169e+00	9.707231e-01	9.746662e-01
0	3.215870e-01	2.104313	6.543175e-01	8.123688e-01	8.368697e-01
0	1.064582e-02	2.000113	2.129203e-02	3.109412e-01	3.827587e-01
0	4.021572e-07	2.000000	8.043143e-07	1.064541e-02	3.423609e-02
0	-6.935643e-17	2.000000	-1.110223e-16	4.021572e-07	3.777751e-05
0	-1.384528e-17	2.000000	0.000000e+00	5.551115e-17	1.380335e-10
0	-1.384528e-17	2.000000	0.000000e+00	0.000000e+00	0.000000e+00

19.2.4. Newton's Method - 2nd Order Method (using Derivative and Hessian)

Input: $f(x)$, $\nabla f(x)$, $\nabla^2 f(x)$, initial guess x^0 , learning rate α , tolerance ε

Output: An approximate solution x

1. Set $t = 0$
2. While $\|f(x^t)\|_2 > \varepsilon$:
 - (a) Set $x^{t+1} \leftarrow x^t - \alpha[\nabla^2 f(x^t)]^{-1}\nabla f(x^t)$.
 - (b) Set $t \leftarrow t + 1$.
3. Return x^t .

Theorem 19.3: Rate of Convergence of Second-Order Methods in Continuous Optimization

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a twice continuously differentiable function and x^* be a local minimum of f . If the Hessian matrix $\nabla^2 f(x^*)$ is positive definite, and if the iteration scheme is based on a quadratic model of the objective function (like Newton's method), then the method is locally quadratically convergent. Specifically, there exists a neighborhood N of x^* such that for all starting points x_0 in N , the error at iteration $k + 1$ satisfies:

$$\|x_{k+1} - x^*\| \leq C\|x_k - x^*\|^2$$

for some constant $C > 0$.

Proof. Assuming that we're working with Newton's method, the iteration is given by:

$$x_{k+1} = x_k - (\nabla^2 f(x_k))^{-1} \nabla f(x_k)$$

1. Taylor's expansion of f about x^* gives:

$$f(x) \approx f(x^*) + \nabla f(x^*)^T(x - x^*) + \frac{1}{2}(x - x^*)^T \nabla^2 f(x^*)(x - x^*)$$

2. Since x^* is a local minimum and the gradient vanishes at x^* , the first-order term disappears, and we have:

$$f(x) \approx f(x^*) + \frac{1}{2}(x - x^*)^T \nabla^2 f(x^*)(x - x^*)$$

3. Using Newton's iteration, the update becomes:

$$x_{k+1} - x^* = (x_k - x^*) - (\nabla^2 f(x_k))^{-1} \nabla f(x_k)$$

4. As f is twice continuously differentiable and $\nabla^2 f(x^*)$ is positive definite, there exists a neighborhood N of x^* where the Hessian is invertible and Lipschitz continuous. Thus, for x_k close enough to x^* , the Hessian at x_k will be close to the Hessian at x^* .

5. Consequently, for x_k close enough to x^* , $\nabla^2 f(x_k)$ will also be positive definite, ensuring that Newton's direction is a descent direction.

6. From the Taylor expansion, the gradient $\nabla f(x_k)$ is approximately proportional to $x_k - x^*$, and the Hessian is roughly constant. Hence, the update step in the Newton iteration will be proportional to the square of the error:

$$x_{k+1} - x^* \propto (x_k - x^*)^2$$

From the above, we see that the error at iteration $k + 1$ is bounded by the square of the error at iteration k , proving the local quadratic convergence. ♠

19.3 Multi-Variate Unconstrained Optimization

We will now use the techniques for 1-Dimensional optimization and extend them to multi-variate case. We will begin with unconstrained versions (or at least, constrained to a large box) and then show how we can apply these techniques to constrained optimization.

19.3.1. Descent Methods - Unconstrained Optimization - Gradient, Newton

Outline for Descent Method for Unconstrained Optimization:

Input:

- A function $f(x)$

- Initial solution x^0
- Method for computing step direction d_t
- Method for computing length t of step
- Number of iterations T

Output:

- A point x_T (hopefully an approximate minimizer)

Algorithm

1. For $t = 1, \dots, T$,

$$\text{set } x_{t+1} = x_t + \alpha_t d_t$$

19.3.1.1. Choice of α_t

There are many different ways to choose the step length α_t . Some choices have proofs that the algorithm will converge quickly. An easy choice is to have a constant step length $\alpha_t = \alpha$, but this may depend on the specific problem.

19.3.1.2. Choice of d_t using $\nabla f(x)$

Choice of descent methods using $\nabla f(x)$ are known as *first order methods*. Here are some choices:

1. **Gradient Descent:** $d_t = -\nabla f(x_t)$
2. **Nesterov Accelerated Descent:** $d_t = \mu(x_t - x_{t-1}) - \gamma \nabla f(x_t + \mu(x_t - x_{t-1}))$

Here, μ, γ are some numbers. The number μ is called the momentum.

19.3.2. Stochastic Gradient Descent - The mother of all algorithms.

A popular method is called *stochastic gradient descent* (SGD). This has been described as "The mother of all algorithms". This is a method to **approximate the gradient** typically used in machine learning or stochastic programming settings.

Stochastic Gradient Descent:

Suppose we want to solve

$$\min_{x \in \mathbb{R}^n} F(x) = \sum_{i=1}^N f_i(x). \quad (19.1)$$

We could use *gradient descent* and have to compute the gradient $\nabla F(x)$ at each iteration. But! We see that in the **cost to compute the gradient** is roughly $O(nN)$, that is, it is very dependent on the number of function N , and hence each iteration will take time dependent on N .

Instead! Let i be a uniformly random sample from $\{1, \dots, N\}$. Then we will use $\nabla f_i(x)$ as an approximation of $\nabla F(x)$. Although we lose a bit by using a guess of the gradient, this approximation only takes $O(n)$ time to compute. And in fact, in expectation, we are doing the same thing. That is,

$$N \cdot \mathbb{E}(\nabla f_i(x)) = N \sum_{i=1}^N \frac{1}{N} \nabla f_i(x) = \sum_{i=1}^N \nabla f_i(x) = \nabla \left(\sum_{i=1}^N f_i(x) \right) = \nabla F(x).$$

Hence, the SGD algorithm is:

1. Set $t = 0$
2. While ... (some stopping criterion)
 - (a) Choose i uniformly at random in $\{1, \dots, N\}$.
 - (b) Set $d_t = \nabla f_i(x_t)$
 - (c) Set $x_{t+1} = x_t - \alpha d_t$

There can be many variations on how to decide which functions f_i to evaluate gradient information on. Above is just one example.

Linear regression is an excellent example of this.

Example 19.4: Linear Regression with SGD

Given data points $x^1, \dots, x^N \in \mathbb{R}^d$ and output $y^1, \dots, y^N \in \mathbb{R}$, find $a \in \mathbb{R}^d$ and $b \in \mathbb{R}$ such that $a^\top x^i + b \approx y^i$. This can be written as the optimization problem

$$\min_{a,b} \sum_{i=1}^N g_i(a,b) \tag{19.2}$$

where $g_i(a,b) = (a^\top x^i + b)^2$.

Notice that the objective function $G(a,b) = \sum_{i=1}^N g_i(a,b)$ is a convex quadratic function. The gradient of the objective function is

$$\nabla G(a,b) = \sum_{i=1}^N \nabla g_i(a,b) = \sum_{i=1}^N 2x^i(a^\top x^i + b)$$

Hence, if we want to use gradient descent, we must compute this large sum (think of $N \approx 10,000$). Instead, we can **approximate the gradient!**. Let $\tilde{\nabla}G(a,b)$ be our approximate gradient. We will compute this by randomly choosing a value $r \in \{1, \dots, N\}$ (with uniform probability). Then set

$$\tilde{\nabla}G(a,b) = \nabla g_r(a,b).$$

It holds that the expected value is the same as the gradient, that is,

$$\mathbb{E}(\tilde{\nabla}G(a, b)) = G(a, b).$$

Hence, we can make probabilistic arguments that these two will have the same (or similar) convergence properties (in expectation).

19.3.3. Neural Networks

Resources

- *ML Zero to Hero - Part 1: Intro to machine learning*
- *ML Zero to Hero - Part 2: Basic Computer Vision with ML*

19.3.4. Choice of Δ_k using the hessian $\nabla^2 f(x)$

These choices are called *second order methods*

1. **Newton's Method:** $\Delta_k = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k)$
2. **BFGS (Quasi-Newton):** $\Delta_k = -(B_k)^{-1} \nabla f(x_k)$

Here

$$\begin{aligned}s_k &= x_{k+1} - x_k \\ y_k &= \nabla f(x_{k+1}) - \nabla f(x_k)\end{aligned}$$

and

$$B_{k+1} = B_k - \frac{(B_k s_k)(B_k s_k)^\top}{s_k^\top B_k s_k} + \frac{y_k y_k^\top}{y_k^\top s_k}.$$

This serves as an approximation of the hessian and can be efficiently computed. Furthermore, the inverse can be easily computed using certain updating rules. This makes for a fast way to approximate the hessian.

19.4 Constrained Convex Nonlinear Programming

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

$$\begin{aligned}\min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d\end{aligned}\tag{19.1}$$

19.4.1. Barrier Method

Constrained Convex Programming via Barrier Method:

We convert 19.1 into the unconstrained minimization problem:

$$\begin{aligned} \min \quad & f(x) - \phi \sum_{i=1}^m \log(-f_i(x)) \\ x \in \mathbb{R}^d \end{aligned} \tag{19.2}$$

Here $\phi > 0$ is some number that we choose. As $\phi \rightarrow 0$, the optimal solution $x(\phi)$ to (19.2) tends to the optimal solution of (19.1). That is $x(\phi) \rightarrow x^*$ as $\phi \rightarrow 0$.

Constrained Convex Programming via Barrier Method - Initial solution:

Define a variable $s \in \mathbb{R}$ and add that to the right hand side of the inequalities and then minimize it in the objective function.

$$\begin{aligned} \min \quad & s \\ \text{s.t.} \quad & f_i(x) \leq s \quad \text{for } i = 1, \dots, m \\ x \in \mathbb{R}^d, s \in \mathbb{R} \end{aligned} \tag{19.3}$$

Note that this problem is feasible for all x values since s can always be made larger. If there exists a solution with $s \leq 0$, then we can use the corresponding x solution as an initial feasible solution. Otherwise, the problem is infeasible.

Now, convert this problem into the unconstrained minimization problem:

$$\begin{aligned} \min \quad & f(x) - \phi \sum_{i=1}^m \log(-(f_i(x) - s)) \\ x \in \mathbb{R}^d, s \in \mathbb{R} \end{aligned} \tag{19.4}$$

This problem has an easy time of finding an initial feasible solution. For instance, let $x = 0$, and then $s = \max_i f_i(x) + 1$.

Images below: the value t is the value ϕ discussed above

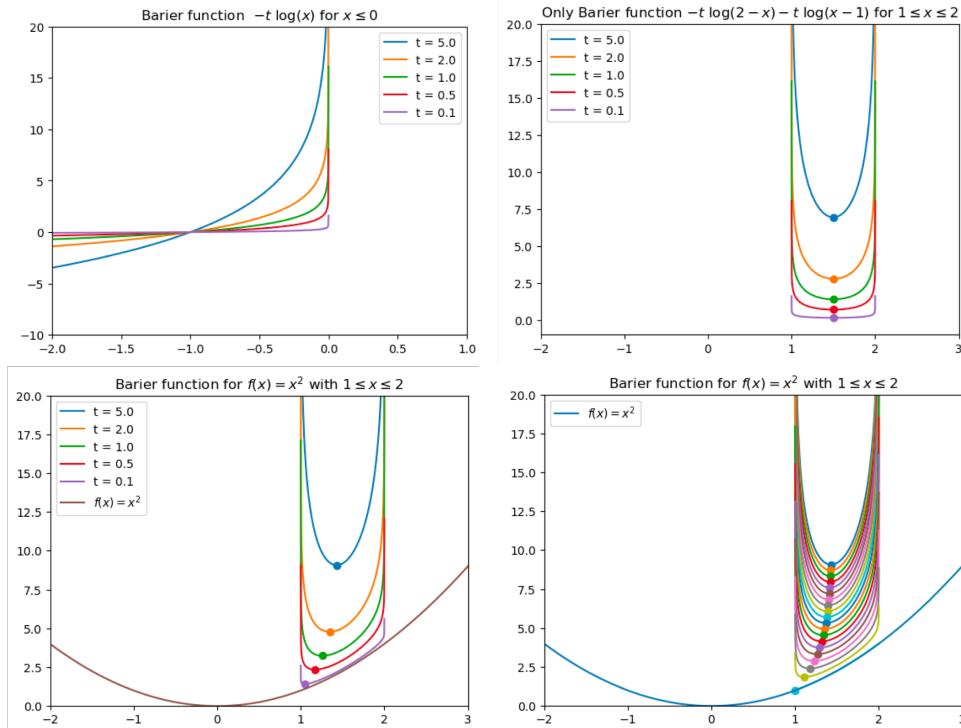


Figure 19.3: Upper Left: Barrier function for varying values of t . Upper Right: Combining barrier functions to create a barrier function on the interval $1 \leq x \leq 2$. Lower left: Combining barrier function and the objective function x^2 for certain choices of t . Lower right: Same picture but more values of t .

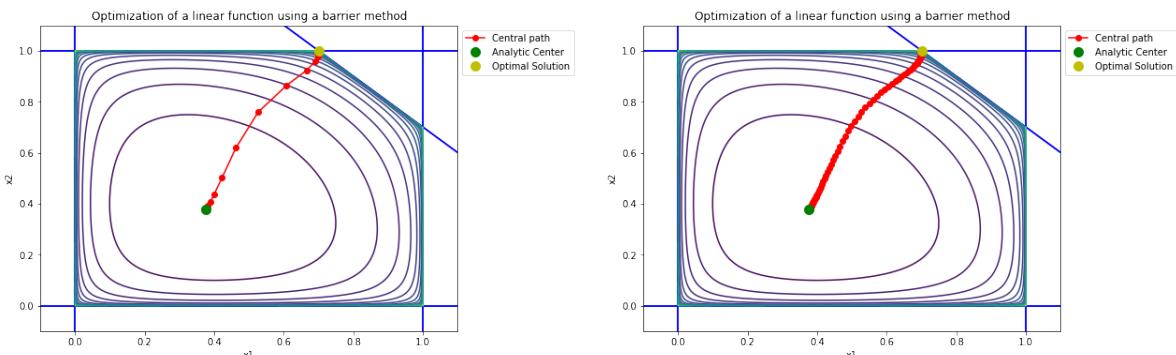


Figure 19.4: Barrier method applied to minimizing a linear program with two variables. Contours of the barrier plotted and the red curve is an approximation of the central path.

Complexity 11.5.3 Total number of Newton iterations We can now give an upper bound on the total number of Newton steps in the barrier method, not counting the initial centering step (which we will analyze later, as part of phase I). We multiply (11.26), which bounds the number of Newton steps per outer iteration, by (11.13), the number of outer steps required, to obtain

$$N = \left\lceil \frac{\log \left(m / \left(t^{(0)} \varepsilon \right) \right)}{\log \mu} \right\rceil \left(\frac{m(\mu - 1 - \log \mu)}{\gamma} + c \right)$$

20. Computational Issues with NLP

We mention a few computational issues to consider with nonlinear programs.

20.1 Irrational Solutions

Consider nonlinear problem (this is even convex)

$$\begin{aligned} \min \quad & -x \\ \text{s.t.} \quad & x^2 \leq 2. \end{aligned} \tag{20.1}$$

The optimal solution is $x^* = \sqrt{2}$, which cannot be easily represented. Hence, we would settle for an **approximate solution** such as $\bar{x} = 1.41421$, which is feasible since $\bar{x}^2 \leq 2$, and it is close to optimal.

20.2 Discrete Solutions

Consider nonlinear problem (not convex)

$$\begin{aligned} \min \quad & -x \\ \text{s.t.} \quad & x^2 = 2. \end{aligned} \tag{20.1}$$

Just as before, the optimal solution is $x^* = \sqrt{2}$, which cannot be easily represented. Furthermore, the only two feasible solutions are $\sqrt{2}$ and $-\sqrt{2}$. Thus, there is no chance to write down a feasible rational approximation.

20.3 Convex NLP Harder than LP

Convex NLP is typically polynomially solvable. It is a generalization of linear programming.

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

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d \end{aligned} \tag{20.1}$$

Example 20.1: C

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

20.4 NLP is harder than IP

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

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \{0, 1\}^n \\ & x_i(1 - x_i) = 0 \quad \text{for } i = 1, \dots, n \end{aligned} \tag{20.1}$$

20.5 Resources

Resources

GRADIENT FREE ALGORITHMS - NEEDLER-MEAD

- [Wikipedia](#)
- [Youtube](#)

BISECTION METHOD AND NEWTON'S METHOD

- See section 4 of the following notes: <http://www.seas.ucla.edu/~vandenbe/133A/133A-notes.pdf>

GRADIENT DESCENT

- <https://www.youtube.com/watch?v=tIpKfDc295M>
- https://www.youtube.com/watch?v=_-02ze7tf08
- https://www.youtube.com/watch?v=N_ZRcLheNv0
- <https://www.youtube.com/watch?v=4RBkIJPG6Yo>

IDEA OF GRADIENT DESCENT

- <https://youtu.be/IHZwWFHwa-w?t=323>

Vectors:

- https://www.youtube.com/watch?v=fNk_zzaMoSs&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab&index=2&t=0s

21. Fairness in Algorithms

Resources

- *Simons Institute - Michael Kearns (University of Pennsylvania) - "The Ethical Algorithm"*

22. One-dimensional Optimization

Lab Objective: *Most mathematical optimization problems involve estimating the minimizer(s) of a scalar-valued function. Many algorithms for optimizing functions with a high-dimensional domain depend on routines for optimizing functions of a single variable. There are many techniques for optimization in one dimension, each with varying degrees of precision and speed. In this lab, we implement the golden section search method, Newton's method, and the secant method, then apply them to the backtracking problem.*

Golden Section Search

A function $f : [a, b] \rightarrow \mathbb{R}$ satisfies the *unimodal property* if it has exactly one local minimum and is monotonic on either side of the minimizer. In other words, f decreases from a to its minimizer x^* , then increases up to b (see Figure 22.1). The *golden section search* method optimizes a unimodal function f by iteratively defining smaller and smaller intervals containing the unique minimizer x^* . This approach is especially useful if the function's derivative does not exist, is unknown, or is very costly to compute.

By definition, the minimizer x^* of f must lie in the interval $[a, b]$. To shrink the interval around x^* , we test the following strategically chosen points.

$$\tilde{a} = b - \frac{b-a}{\varphi} \quad \tilde{b} = a + \frac{b-a}{\varphi}$$

Here $\varphi = \frac{1+\sqrt{5}}{2}$ is the *golden ratio*. At each step of the search, $[a, b]$ is refined to either $[a, \tilde{b}]$ or $[\tilde{a}, b]$, called the *golden sections*, depending on the following criteria.

If $f(\tilde{a}) < f(\tilde{b})$, then since f is unimodal, it must be increasing in a neighborhood of \tilde{b} . The unimodal property also guarantees that f must be increasing on $[\tilde{b}, b]$ as well, so $x^* \in [a, \tilde{b}]$ and we set $b = \tilde{b}$. By similar reasoning, if $f(\tilde{a}) > f(\tilde{b})$, then $x^* \in [\tilde{a}, b]$ and we set $a = \tilde{a}$. If, however, $f(\tilde{a}) = f(\tilde{b})$, then the unimodality of f does not guarantee anything about where the minimizer lies. Assuming either $x^* \in [a, \tilde{b}]$ or $x^* \in [\tilde{a}, b]$ allows the iteration to continue, but the method is no longer guaranteed to converge to the local minimum.

At each iteration, the length of the search interval is divided by φ . The method therefore converges linearly, which is somewhat slow. However, the idea is simple and each step is computationally inexpensive.

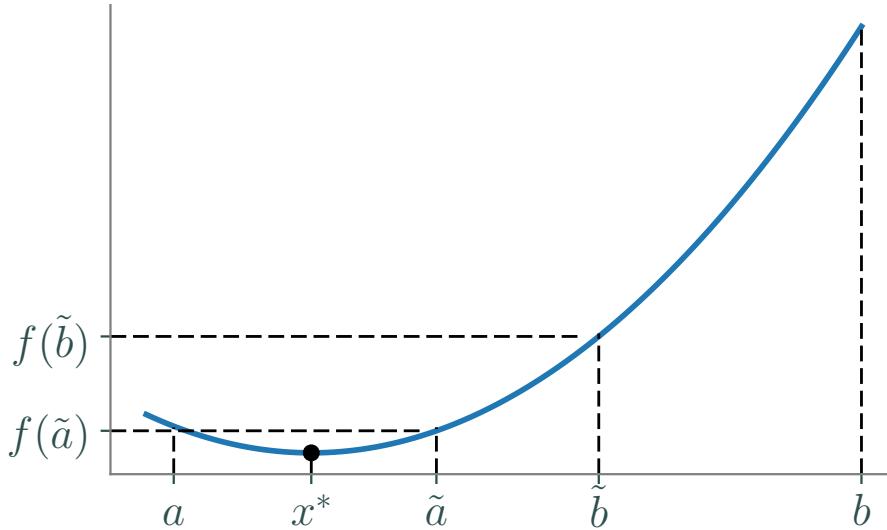


Figure 22.1: The unimodal $f : [a, b] \rightarrow \mathbb{R}$ can be minimized with a golden section search. For the first iteration, $f(\tilde{a}) < f(\tilde{b})$, so $x^* \in [\tilde{a}, \tilde{b}]$. New values of \tilde{a} and \tilde{b} are then calculated from this new, smaller interval.

Algorithm 13 The Golden Section Search

```

1: procedure GOLDEN_SECTION( $f, a, b, \text{tol}, \text{maxiter}$ )
2:    $x_0 \leftarrow (a + b)/2$                                  $\triangleright$  Set the initial minimizer approximation as the interval midpoint.
3:    $\varphi = (1 + \sqrt{5})/2$ 
4:   for  $i = 1, 2, \dots, \text{maxiter}$  do                       $\triangleright$  Iterate only  $\text{maxiter}$  times at most.
5:      $c \leftarrow (b - a)/\varphi$ 
6:      $\tilde{a} \leftarrow b - c$ 
7:      $\tilde{b} \leftarrow a + c$ 
8:     if  $f(\tilde{a}) \leq f(\tilde{b})$  then                   $\triangleright$  Get new boundaries for the search interval.
9:        $b \leftarrow \tilde{b}$ 
10:    else
11:       $a \leftarrow \tilde{a}$ 
12:       $x_1 \leftarrow (a + b)/2$                            $\triangleright$  Set the minimizer approximation as the interval midpoint.
13:      if  $|x_0 - x_1| < \text{tol}$  then
14:        break                                      $\triangleright$  Stop iterating if the approximation stops changing enough.
15:       $x_0 \leftarrow x_1$ 
16:   return  $x_1$ 

```

Problem 22.1: Implement golden search.

rite a function that accepts a function $f : \mathbb{R} \rightarrow \mathbb{R}$, interval limits a and b , a stopping tolerance tol , and a maximum number of iterations $maxiter$. Use Algorithm 13 to implement the golden section search. Return the approximate minimizer x^* , whether or not the algorithm converged (true or false), and the number of iterations computed.

Test your function by minimizing $f(x) = e^x - 4x$ on the interval $[0, 3]$, then plotting the function and the computed minimizer together. Also compare your results to SciPy's golden section search, `scipy.optimize.golden()`.

```
>>> from scipy import optimize as opt
>>> import numpy as np

>>> f = lambda x : np.exp(x) - 4*x
>>> opt.golden(f, brack=(0,3), tol=.001)
1.3862578679031485 # ln(4) is the minimizer.
```

Newton's Method

Newton's method is an important root-finding algorithm that can also be used for optimization. Given $f : \mathbb{R} \rightarrow \mathbb{R}$ and a good initial guess x_0 , the sequence $(x_k)_{k=1}^\infty$ generated by the recursive rule

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

converges to a point \bar{x} satisfying $f(\bar{x}) = 0$. The first-order necessary conditions from elementary calculus state that if f is differentiable, then its derivative evaluates to zero at each of its local minima and maxima. Therefore using Newton's method to find the zeros of f' is a way to identify potential minima or maxima of f . Specifically, starting with an initial guess x_0 , set

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{22.1}$$

and iterate until $|x_k - x_{k-1}|$ is satisfactorily small. Note that this procedure does not use the actual function f at all, but it requires many evaluations of its first and second derivatives. As a result, Newton's method converges in few iterations, but it can be computationally expensive.

Each step of (22.1) can be thought of approximating the objective function f by a quadratic function q and finding its unique extrema. That is, we first approximate f with its second-degree Taylor polynomial centered at x_k .

$$q(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2$$

This quadratic function satisfies $q(x_k) = f(x_k)$ and matches f fairly well close to x_k . Thus the optimizer of q is a reasonable guess for an optimizer of f . To compute that optimizer, solve $q'(x) = 0$.

$$0 = q'(x) = f'(x_k) + f''(x_k)(x - x_k) \implies x = x_k - \frac{f'(x_k)}{f''(x_k)}$$

This agrees with (22.1) using x_{k+1} for x . See Figure 22.2.

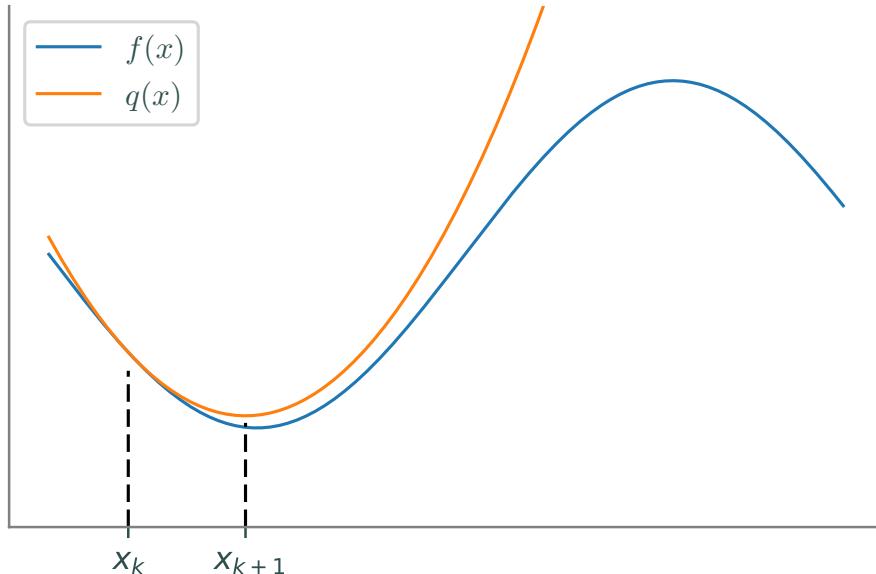


Figure 22.2: A quadratic approximation of f at x_k . The minimizer x_{k+1} of q is close to the minimizer of f .

Newton's method for optimization works well to locate minima when $f''(x) > 0$ on the entire domain. However, it may fail to converge to a minimizer if $f''(x) \leq 0$ for some portion of the domain. If f is not unimodal, the initial guess x_0 must be sufficiently close to a local minimizer x^* in order to converge.

Problem 22.2: Newton's method for optimization

Let $f : \mathbb{R} \rightarrow \mathbb{R}$. Write a function that accepts f' , f'' , a starting point x_0 , a stopping tolerance tol , and a maximum number of iterations maxiter . Implement Newton's method using (22.1) to locate a local optimizer. Return the approximate optimizer, whether or not the algorithm converged, and the number of iterations computed.

Test your function by minimizing $f(x) = x^2 + \sin(5x)$ with an initial guess of $x_0 = 0$. Compare your results to `scipy.optimize.newton()`, which implements the root-finding version of Newton's method.

```
>>> df = lambda x : 2*x + 5*np.cos(5*x)
>>> d2f = lambda x : 2 - 25*np.sin(5*x)
>>> opt.newton(df, x0=0, fprime=d2f, tol=1e-10, maxiter=500)
-1.4473142236328096
```

Note that other initial guesses can yield different minima for this function.

CONVERGENCE OF NEWTON'S METHOD We consider the function $f(x) = e^x + e^{-x}$.

	x	$f(x)$	$f'(x)$	$ x^k - x^{k-1} $	$\frac{ x^k - x^{k-1} }{ x^{k-1} - x^{k-2} }$
0	6.100000e+00	445.860013	4.458555e+02	NaN	NaN
0	5.100010e+00	164.029654	1.640175e+02	9.999899e-01	NaN
0	4.100084e+00	60.361952	6.032881e+01	9.999257e-01	9.999357e-01
0	3.100633e+00	22.257038	2.216700e+01	9.994509e-01	9.995252e-01
0	2.104679e+00	8.326354	8.082584e+00	9.959545e-01	9.965016e-01
0	1.133956e+00	3.429685	2.786169e+00	9.707231e-01	9.746662e-01
0	3.215870e-01	2.104313	6.543175e-01	8.123688e-01	8.368697e-01
0	1.064582e-02	2.000113	2.129203e-02	3.109412e-01	3.827587e-01
0	4.021572e-07	2.000000	8.043143e-07	1.064541e-02	3.423609e-02
0	-6.935643e-17	2.000000	-1.110223e-16	4.021572e-07	3.777751e-05
0	-1.384528e-17	2.000000	0.000000e+00	5.551115e-17	1.380335e-10
0	-1.384528e-17	2.000000	0.000000e+00	0.000000e+00	0.000000e+00

Theorem 22.3: Rate of Convergence of Second-Order Methods in Continuous Optimization

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a twice continuously differentiable function and x^* be a local minimum of f . If the Hessian matrix $\nabla^2 f(x^*)$ is positive definite, and if the iteration scheme is based on a quadratic model of the objective function (like Newton's method), then the method is locally quadratically convergent. Specifically, there exists a neighborhood N of x^* such that for all starting points x_0 in N , the error at iteration $k+1$ satisfies:

$$\|x_{k+1} - x^*\| \leq C \|x_k - x^*\|^2$$

for some constant $C > 0$.

Proof. Assuming that we're working with Newton's method, the iteration is given by:

$$x_{k+1} = x_k - (\nabla^2 f(x_k))^{-1} \nabla f(x_k)$$

1. Taylor's expansion of f about x^* gives:

$$f(x) \approx f(x^*) + \nabla f(x^*)^T (x - x^*) + \frac{1}{2} (x - x^*)^T \nabla^2 f(x^*) (x - x^*)$$

2. Since x^* is a local minimum and the gradient vanishes at x^* , the first-order term disappears, and we have:

$$f(x) \approx f(x^*) + \frac{1}{2} (x - x^*)^T \nabla^2 f(x^*) (x - x^*)$$

3. Using Newton's iteration, the update becomes:

$$x_{k+1} - x^* = (x_k - x^*) - (\nabla^2 f(x_k))^{-1} \nabla f(x_k)$$

4. As f is twice continuously differentiable and $\nabla^2 f(x^*)$ is positive definite, there exists a neighborhood N of x^* where the Hessian is invertible and Lipschitz continuous. Thus, for x_k close enough to x^* , the Hessian at x_k will be close to the Hessian at x^* .

5. Consequently, for x_k close enough to x^* , $\nabla^2 f(x_k)$ will also be positive definite, ensuring that Newton's direction is a descent direction.

6. From the Taylor expansion, the gradient $\nabla f(x_k)$ is approximately proportional to $x_k - x^*$, and the Hessian is roughly constant. Hence, the update step in the Newton iteration will be proportional to the square of the error:

$$x_{k+1} - x^* \propto (x_k - x^*)^2$$

From the above, we see that the error at iteration $k+1$ is bounded by the square of the error at iteration k , proving the local quadratic convergence. ♠

The Secant Method

The second derivative of an objective function is not always known or may be prohibitively expensive to evaluate. The *secant method* solves this problem by numerically approximating the second derivative with a difference quotient.

$$f''(x) \approx \frac{f'(x+h) - f'(x)}{h}$$

Selecting $x = x_k$ and $h = x_{k-1} - x_k$ gives the following approximation.

$$f''(x_k) \approx \frac{f'(x_k + x_{k-1} - x_k) - f'(x_k)}{x_{k-1} - x_k} = \frac{f(x_k) - f'(x_{k-1})}{x_k - x_{k-1}} \quad (22.2)$$

Inserting (22.2) into (22.1) results in the complete secant method formula.

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} f'(x_k) = \frac{x_{k-1}f'(x_k) - x_kf'(x_{k-1})}{f'(x_k) - f'(x_{k-1})} \quad (22.3)$$

Notice that this recurrence relation requires two previous points (both x_k and x_{k-1}) to calculate the next estimate. This method converges superlinearly—slower than Newton’s method, but faster than the golden section search—with convergence criteria similar to Newton’s method.

Problem 22.4: Implement secant method

Write a function that accepts a first derivative f' , starting points x_0 and x_1 , a stopping tolerance tol , and a maximum of iterations $maxiter$. Use (22.3) to implement the Secant method. Try to make as few computations as possible by only computing $f'(x_k)$ once for each k . Return the minimizer approximation, whether or not the algorithm converged, and the number of iterations computed. Test your code with the function $f(x) = x^2 + \sin(x) + \sin(10x)$ and with initial guesses of $x_0 = 0$ and $x_1 = -1$. Plot your answer with the graph of the function. Also compare your results to `scipy.optimize.newton()`; without providing the `fprime` argument, this function uses the secant method. However, it still only takes in one initial condition, so it may converge to a different local minimum than your function.

```
>>> df = lambda x: 2*x + np.cos(x) + 10*np.cos(10*x)
>>> opt.newton(df, x0=0, tol=1e-10, maxiter=500)
-3.2149595174761636
```

Descent Methods

Consider now a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. *Descent methods*, also called *line search methods*, are optimization algorithms that create a convergent sequence $(x_k)_{k=1}^\infty$ by the following rule.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (22.4)$$

Here $\alpha_k \in \mathbb{R}$ is called the *step size* and $\mathbf{p}_k \in \mathbb{R}^n$ is called the *search direction*. The choice of \mathbf{p}_k is usually what distinguishes an algorithm; in the one-dimensional case ($n = 1$), $p_k = f'(x_k)/f''(x_k)$ results in Newton's method, and using the approximation in (22.2) results in the secant method.

To be effective, a descent method must also use a good step size α_k . If α_k is too large, the method may repeatedly overstep the minimum; if α_k is too small, the method may converge extremely slowly. See Figure 22.3.

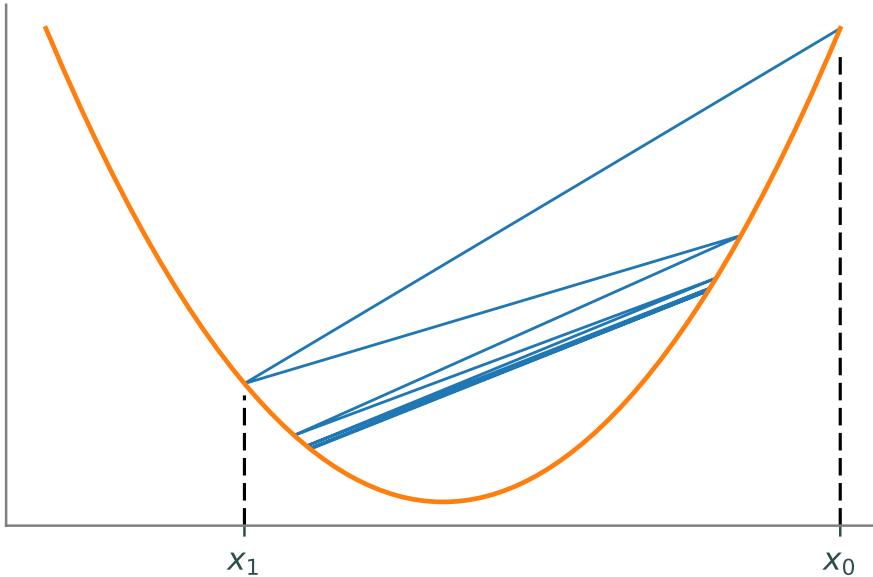


Figure 22.3: If the step size α_k is too large, a descent method may repeatedly overstep the minimizer.

Given a search direction \mathbf{p}_k , the best step size α_k minimizes the function $\phi_k(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{p}_k)$. Since f is scalar-valued, $\phi_k : \mathbb{R} \rightarrow \mathbb{R}$, so any of the optimization methods discussed previously can be used to minimize ϕ_k . However, computing the best α_k at every iteration is not always practical. Instead, some methods use a cheap routine to compute a step size that may not be optimal, but which is good enough. The most common approach is to find an α_k that satisfies the *Wolfe conditions*:

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k Df(\mathbf{x}_k)^\top \mathbf{p}_k \quad (22.5)$$

$$-Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^\top \mathbf{p}_k \leq -c_2 Df(\mathbf{x}_k)^\top \mathbf{p}_k \quad (22.6)$$

where $0 < c_1 < c_2 < 1$ (for the best results, choose $c_1 \ll c_2$). The condition (22.5) is also called the *Armijo rule* and ensures that the step decreases f . However, this condition is not enough on its own. By

Taylor's theorem,

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) = f(\mathbf{x}_k) + \alpha_k Df(\mathbf{x}_k)^\top \mathbf{p}_k + \mathcal{O}(\alpha_k^2).$$

Thus, a very small α_k will always satisfy (22.5) since $Df(\mathbf{x}_k)^\top \mathbf{p}_k < 0$ (as \mathbf{p}_k is a descent direction). The condition (22.6), called the *curvature condition*, ensures that the α_k is large enough for the algorithm to make significant progress.

It is possible to find an α_k that satisfies the Wolfe conditions, but that is far from the minimizer of $\phi_k(\alpha)$. The *strong Wolfe conditions* modify (22.6) to ensure that α_k is near the minimizer.

$$|Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^\top \mathbf{p}_k| \leq c_2 |Df(\mathbf{x}_k)^\top \mathbf{p}_k|$$

The *Armijo–Goldstein conditions* provide another alternative to (22.6):

$$f(\mathbf{x}_k) + (1 - c)\alpha_k Df(\mathbf{x}_k)^\top \mathbf{p}_k \leq f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c\alpha_k Df(\mathbf{x}_k)^\top \mathbf{p}_k,$$

where $0 < c < 1$. These conditions are very similar to the Wolfe conditions (the right inequality is (22.5)), but they do not require the calculation of the directional derivative $Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^\top \mathbf{p}_k$.

Backtracking

A *backtracking line search* is a simple strategy for choosing an acceptable step size α_k : start with an fairly large initial step size α , then repeatedly scale it down by a factor ρ until the desired conditions are satisfied. The following algorithm only requires α to satisfy (22.5). This is usually sufficient, but if it finds α 's that are too small, the algorithm can be modified to satisfy (22.6) or one of its variants.

Algorithm 14 Backtracking using the Armijo Rule

```

1: procedure BACKTRACKING( $f, Df, \mathbf{x}_k, \mathbf{p}_k, \alpha, \rho, c$ )
2:    $Dfp \leftarrow Df(\mathbf{x}_k)^\top \mathbf{p}_k$                                  $\triangleright$  Compute these values only once.
3:    $fx \leftarrow f(\mathbf{x}_k)$ 
4:   while  $(f(\mathbf{x}_k + \alpha \mathbf{p}_k) > fx + c\alpha Dfp)$  do
5:      $\alpha \leftarrow \rho \alpha$ 
return  $\alpha$ 

```

Problem 22.5: Implement the backtracking method

Write a function that accepts a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, its derivative $Df : \mathbb{R}^n \rightarrow \mathbb{R}^n$, an approximate minimizer \mathbf{x}_k , a search direction \mathbf{p}_k , an initial step length α , and parameters ρ and c . Implement the backtracking method of Algorithm 14. Return the computed step size.

The functions f and Df should both accept 1-D NumPy arrays of length n . For example, if $f(x,y,z) = x^2 + y^2 + z^2$, then f and Df could be defined as follows.

```
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> Df = lambda x: np.array([2*x[0], 2*x[1], 2*x[2]])
```

SciPy's `scipy.optimize.linesearch.scalar_search_armijo()` finds an acceptable step size using the Armijo rule. It may not give the exact answer as your implementation since it decreases α differently, but the answers should be similar.

```
>>> from scipy.optimize import linesearch
>>> from autograd import numpy as np
>>> from autograd import grad

# Get a step size for f(x,y,z) = x^2 + y^2 + z^2.
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> x = np.array([150., .03, 40.])           # Current minimizer guesss.
>>> p = np.array([-5., -100., -4.5])        # Current search direction.
>>> phi = lambda alpha: f(x + alpha*p)       # Define phi(alpha).
>>> dphi = grad(phi)
>>> alpha, _ = linesearch.scalar_search_armijo(phi, phi(0.), dphi(0.))
```

23. Gradient Descent Methods

Lab Objective: Iterative optimization methods choose a search direction and a step size at each iteration. One simple choice for the search direction is the negative gradient, resulting in the method of steepest descent. While theoretically foundational, in practice this method is often slow to converge. An alternative method, the conjugate gradient algorithm, uses a similar idea that results in much faster convergence in some situations. In this lab we implement a method of steepest descent and two conjugate gradient methods, then apply them to regression problems.

The Method of Steepest Descent

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ with first derivative $Df : \mathbb{R}^n \rightarrow \mathbb{R}^n$. The following iterative technique is a common template for methods that aim to compute a local minimizer \mathbf{x}^* of f .

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (23.1)$$

Here \mathbf{x}_k is the k th approximation to \mathbf{x}^* , α_k is the *step size*, and \mathbf{p}_k is the *search direction*. Newton's method and its relatives follow this pattern, but they require the calculation (or approximation) of the inverse Hessian matrix $Df^2(\mathbf{x}_k)^{-1}$ at each step. The following idea is a simpler and less computationally intensive approach than Newton and quasi-Newton methods.

The derivative $Df(\mathbf{x})^\top$ (often called the *gradient* of f at \mathbf{x} , sometimes notated $\nabla f(\mathbf{x})$) is a vector that points in the direction of greatest **increase** of f at \mathbf{x} . It follows that the negative derivative $-Df(\mathbf{x})^\top$ points in the direction of steepest **decrease** at \mathbf{x} . The *method of steepest descent* chooses the search direction $\mathbf{p}_k = -Df(\mathbf{x}_k)^\top$ at each step of (23.1), resulting in the following algorithm.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\top \quad (23.2)$$

Setting $\alpha_k = 1$ for each k is often sufficient for Newton and quasi-Newton methods. However, a constant choice for the step size in (23.2) can result in oscillating approximations or even cause the sequence $(\mathbf{x}_k)_{k=1}^\infty$ to travel away from the minimizer \mathbf{x}^* . To avoid this problem, the step size α_k can be chosen in a few ways.

- Start with $\alpha_k = 1$, then set $\alpha_k = \frac{\alpha_k}{2}$ until $f(\mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\top) < f(\mathbf{x}_k)$, terminating the iteration if α_k gets too small. This guarantees that the method actually descends at each step and that α_k satisfies the Armijo rule, without endangering convergence.

- At each step, solve the following one-dimensional optimization problem.

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_k - \alpha Df(\mathbf{x}_k)^\top)$$

Using this choice is called *exact steepest descent*. This option is more expensive per iteration than the above strategy, but it results in fewer iterations before convergence.

Problem 23.1: Implement exact steepest descent

Write a function that accepts an objective function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, its derivative $Df : \mathbb{R}^n \rightarrow \mathbb{R}^n$, an initial guess $\mathbf{x}_0 \in \mathbb{R}^n$, a convergence tolerance tol defaulting to $1e^{-5}$, and a maximum number of iterations maxiter defaulting to 100. Implement the exact method of steepest descent, using a one-dimensional optimization method to choose the step size (use `opt.minimize_scalar()` or your own 1-D minimizer). Iterate until $\|Df(\mathbf{x}_k)\|_\infty < \text{tol}$ or $k > \text{maxiter}$. Return the approximate minimizer \mathbf{x}^* , whether or not the algorithm converged (`True` or `False`), and the number of iterations computed.

Test your function on $f(x, y, z) = x^4 + y^4 + z^4$ (easy) and the Rosenbrock function (hard). It should take many iterations to minimize the Rosenbrock function, but it should converge eventually with a large enough choice of `maxiter`.

The Conjugate Gradient Method

Unfortunately, the method of steepest descent can be very inefficient for certain problems. Depending on the nature of the objective function, the sequence of points can zig-zag back and forth or get stuck on flat areas without making significant progress toward the true minimizer.

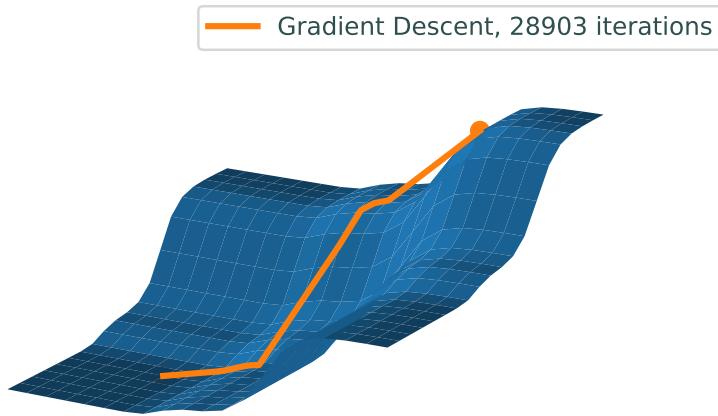


Figure 23.1: On this surface, gradient descent takes an extreme number of iterations to converge to the minimum because it gets stuck in the flat basins of the surface.

Unlike the method of steepest descent, the *conjugate gradient algorithm* chooses a search direction that is guaranteed to be a descent direction, though not the direction of greatest descent. These directions are using a generalized form of orthogonality called *conjugacy*.

Let Q be a square, positive definite matrix. A set of vectors $\{\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_m\}$ is called Q -*conjugate* if each distinct pair of vectors $\mathbf{x}_i, \mathbf{x}_j$ satisfy $\mathbf{x}_i^T Q \mathbf{x}_j = 0$. A Q -conjugate set of vectors is linearly independent and can form a basis that diagonalizes the matrix Q . This guarantees that an iterative method to solve $Q\mathbf{x} = \mathbf{b}$ only require as many steps as there are basis vectors.

Solve a positive definite system $Q\mathbf{x} = \mathbf{b}$ is valuable in and of itself for certain problems, but it is also equivalent to minimizing certain functions. Specifically, consider the quadratic function

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T Q \mathbf{x} - \mathbf{b}^T \mathbf{x} + c.$$

Because $Df(\mathbf{x})^T = Q\mathbf{x} - \mathbf{b}$, minimizing f is the same as solving the equation

$$0 = Df(\mathbf{x})^T = Q\mathbf{x} - \mathbf{b} \quad \Rightarrow \quad Q\mathbf{x} = \mathbf{b},$$

which is the original linear system. Note that the constant c does not affect the minimizer, since if \mathbf{x}^* minimizes $f(\mathbf{x})$ it also minimizes $f(\mathbf{x}) + c$.

Using the conjugate directions guarantees an iterative method to converge on the minimizer because each iteration minimizes the objective function over a subspace of dimension equal to the iteration number. Thus, after n steps, where n is the number of conjugate basis vectors, the algorithm has found a minimizer over the entire space. In certain situations, this has a great advantage over gradient descent, which can bounce back and forth. This comparison is illustrated in Figure 23.2. Additionally, because the method utilizes a basis of conjugate vectors, the previous search direction can be used to find a conjugate projection onto the next subspace, saving computational time.

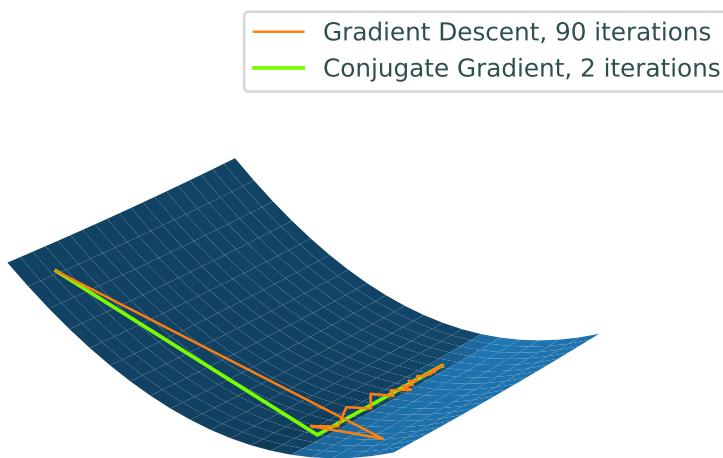


Figure 23.2: Paths traced by Gradient Descent (orange) and Conjugate Gradient (red) on a quadratic surface. Notice the zig-zagging nature of the Gradient Descent path, as opposed to the Conjugate Gradient path, which finds the minimizer in 2 steps.

Algorithm 15

```

1: procedure CONJUGATE GRADIENT( $\mathbf{x}_0, Q, \mathbf{b}, \text{tol}$ )
2:    $\mathbf{r}_0 \leftarrow Q\mathbf{x}_0 - \mathbf{b}$ 
3:    $\mathbf{d}_0 \leftarrow -\mathbf{r}_0$ 
4:    $k \leftarrow 0$ 
5:   while  $\|\mathbf{r}_k\| \geq \text{tol}, k < n$  do
6:      $\alpha_k \leftarrow \mathbf{r}_k^\top \mathbf{r}_k / \mathbf{d}_k^\top Q \mathbf{d}_k$ 
7:      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$ 
8:      $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha_k Q \mathbf{d}_k$ 
9:      $\beta_{k+1} \leftarrow \mathbf{r}_{k+1}^\top \mathbf{r}_{k+1} / \mathbf{r}_k^\top \mathbf{r}_k$ 
10:     $\mathbf{d}_{k+1} \leftarrow -\mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k$ 
11:     $k \leftarrow k + 1.$ 
return  $\mathbf{x}_{k+1}$ 

```

The points \mathbf{x}_k are the successive approximations to the minimizer, the vectors \mathbf{d}_k are the conjugate descent directions, and the vectors \mathbf{r}_k (which actually correspond to the steepest descent directions) are used in determining the conjugate directions. The constants α_k and β_k are used, respectively, in the line search, and in ensuring the Q -conjugacy of the descent directions.

Problem 23.2: Conjugate gradient for linear systems

Write a function that accepts an $n \times n$ positive definite matrix Q , a vector $\mathbf{b} \in \mathbb{R}^n$, an initial guess $\mathbf{x}_0 \in \mathbb{R}^n$, and a stopping tolerance. Use Algorithm 15 to solve the system $Q\mathbf{x} = \mathbf{b}$. Continue the algorithm until $\|\mathbf{r}_k\|$ is less than the tolerance, iterating no more than n times. Return the solution \mathbf{x} , whether or not the algorithm converged in n iterations or less, and the number of iterations computed.

Test your function on the simple system

$$Q = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 8 \end{bmatrix},$$

which has solution $\mathbf{x}^* = [\frac{1}{2}, 2]^\top$. This is equivalent to minimizing the quadratic function $f(x, y) = x^2 + 2y^2 - x - 8y$; check that your function from Problem 23 gets the same solution.

More generally, you can generate a random positive definite matrix Q for testing by setting setting $Q = A^\top A$ for any A of full rank.

```
>>> import numpy as np
>>> from scipy import linalg as la

# Generate Q, b, and the initial guess x0.
>>> n = 10
>>> A = np.random.random((n,n))
>>> Q = A.T @ A
>>> b, x0 = np.random.random((2,n))

>>> x = la.solve(Q, b)      # Use your function here.
>>> np.allclose(Q @ x, b)
True
```

Non-linear Conjugate Gradient

The algorithm presented above is only valid for certain linear systems and quadratic functions, but the basic strategy may be adapted to minimize more general convex or non-linear functions. Though the non-linear version does not have guaranteed convergence as the linear formulation does, it can still converge in less iterations than the method of steepest descent. Modifying the algorithm for more general functions requires new formulas for α_k , \mathbf{r}_k , and β_k .

- The scalar α_k is simply the result of performing a line-search in the given direction \mathbf{d}_k and is thus defined $\alpha_k = \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_k + \alpha \mathbf{d}_k)$.
- The vector \mathbf{r}_k in the original algorithm was really just the gradient of the objective function, so now

define $\mathbf{r}_k = Df(\mathbf{x}_k)^\top$.

- The constants β_k can be defined in various ways, and the most correct choice depends on the nature of the objective function. A well-known formula, attributed to Fletcher and Reeves, is $\beta_k = Df(\mathbf{x}_k)Df(\mathbf{x}_k)^\top / Df(\mathbf{x}_{k-1})Df(\mathbf{x}_{k-1})^\top$.

Algorithm 16

```

1: procedure NON-LINEAR CONJUGATE GRADIENT( $f, Df, \mathbf{x}_0, \text{tol}, \text{maxiter}$ )
2:    $\mathbf{r}_0 \leftarrow -Df(\mathbf{x}_0)^\top$ 
3:    $\mathbf{d}_0 \leftarrow \mathbf{r}_0$ 
4:    $\alpha_0 \leftarrow \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_0 + \alpha \mathbf{d}_0)$ 
5:    $\mathbf{x}_1 \leftarrow \mathbf{x}_0 + \alpha_0 \mathbf{d}_0$ 
6:    $k \leftarrow 1$ 
7:   while  $\|\mathbf{r}_k\| \geq \text{tol}, k < \text{maxiter}$  do
8:      $\mathbf{r}_k \leftarrow -Df(\mathbf{x}_k)^\top$ 
9:      $\beta_k = \mathbf{r}_k^\top \mathbf{r}_k / \mathbf{r}_{k-1}^\top \mathbf{r}_{k-1}$ 
10:     $\mathbf{d}_k \leftarrow \mathbf{r}_k + \beta_k \mathbf{d}_{k-1}$ .
11:     $\alpha_k \leftarrow \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ .
12:     $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$ .
13:     $k \leftarrow k + 1$ .

```

Problem 23.3

Write a function that accepts a convex objective function f , its derivative Df , an initial guess \mathbf{x}_0 , a convergence tolerance defaultin to $1e^{-5}$, and a maximum number of iterations defaultin to 100. Use Algorithm 16 to compute the minimizer \mathbf{x}^* of f . Return the approximate minimizer, whether or not the algorithm converged, and the number of iterations computed.

Compare your function to SciPy's `opt.fmin_cg()`.

```

>>> opt.fmin_cg(opt.rosen, np.array([10, 10]), fprime=opt.rosen_der)
Optimization terminated successfully.
    Current function value: 0.000000
    Iterations: 44
    Function evaluations: 102 # Much faster than steepest descent!
    Gradient evaluations: 102
    array([ 1.00000007,  1.00000015])

```

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>

