

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: July 5, 2022

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	3
2 Mathematical Programming	7
2.1 Linear Programming (LP)	8
2.2 Mixed-Integer Linear Programming (MILP)	9
2.3 Non-Linear Programming (NLP)	11
2.3.1 Convex Programming	11
2.3.2 Non-Convex Non-linear Programming	12
2.4 Mixed-Integer Non-Linear Programming (MINLP)	12
2.4.1 Convex Mixed-Integer Non-Linear Programming	12
2.4.2 Non-Convex Mixed-Integer Non-Linear Programming	12
I Linear Programming	13
3 Modeling: Linear Programming	17
3.1 Modeling and Assumptions in Linear Programming	18
3.1.1 General models	19
3.1.2 Assumptions	20
3.2 Examples	21
3.2.1 Knapsack Problem	27
3.2.2 Capital Investment	27
3.2.3 Work Scheduling	27
3.2.4 Assignment Problem	28
3.2.5 Multi period Models	29
3.2.5.1 Production Planning	29
3.2.5.2 Crop Planning	29
3.2.6 Mixing Problems	29
3.2.7 Financial Planning	29
3.2.8 Network Flow	29
3.2.8.1 Graphs	30
3.2.8.2 Maximum Flow Problem	31
3.2.8.3 Minimum Cost Network Flow	33
3.2.9 Multi-Commodity Network Flow	34
3.3 Modeling Tricks	34
3.3.1 Maximizing a minimum	34

6 ■ CONTENTS

3.4 Other examples	35
4 Graphically Solving Linear Programs	37
4.1 Nonempty and Bounded Problem	37
4.2 Infinitely Many Optimal Solutions	41
4.3 Problems with No Solution	44
4.4 Problems with Unbounded Feasible Regions	46
4.5 Formal Mathematical Statements	51
5 Software - Excel	59
5.0.1 Excel Solver	59
5.0.2 Videos	59
5.0.3 Links	59
6 Software - Python	61
6.1 Installing and Managing Python	61
6.2 NumPy Visual Guide	65
6.3 Plot Customization and Matplotlib Syntax Guide	69
6.4 Networkx - A Python Graph Algorithms Package	75
6.5 PuLP - An Optimization Modeling Tool for Python	76
6.5.1 Installation	76
6.5.2 Example Problem	77
6.5.2.1 Product Mix Problem	77
6.5.3 Things we can do	78
6.5.3.1 Exploring the variables	79
6.5.3.2 Other things you can do	79
6.5.4 Common issue	80
6.5.4.1 Transportation Problem	80
6.5.4.2 Optimization with PuLP	81
6.5.4.3 Optimization with PuLP: Round 2!	82
6.5.5 Changing details of the problem	84
6.5.6 Changing Constraint Coefficients	86
6.6 Multi Objective Optimization with PuLP	86
6.6.0.1 Transportation Problem	86
6.6.0.2 Initial Optimization with PuLP	87
6.6.1 Creating the Pareto Efficient Frontier	89
6.7 Comments	91
6.8 Jupyter Notebooks	91
6.9 Reading and Writing	92
6.10 Python Crash Course	92
6.11 Gurobi	92
6.12 Plots, Pandas, and Geopandas	92
6.12.1 Geopandas	92
6.13 The Simplex Method	92

6.13.1 Pivoting	96
6.13.2 Termination and Reading the Dictionary	98
6.13.3 Exercises	99
6.13.4 2.5 Exercises	101
7 Simplex Method	103
7.1 Simplex Method	106
7.2 Finding Feasible Basis	106
8 Duality	111
8.1 The Dual of Linear Program	112
8.2 Linear programming duality	117
8.2.1 The dual problem	119
Exercises	121
Solutions	122
9 Sensitivity Analysis	125
10 Multi-Objective Optimization	127
10.1 Multi Objective Optimization and The Pareto Frontier	127
10.2 What points will the Scalarization method find if we vary λ ?	133
10.3 Political Redistricting [3]	133
10.4 Portfolio Optimization [5]	133
10.5 Simulated Portfolio Optimization based on Efficient Frontier	134
10.6 Aircraft Design [1]	134
10.7 Vehicle Dynamics [4]	135
10.8 Sustainable Constriction [2]	135
10.9 References	135
II Discrete Algorithms	137
11 Graph Algorithms	139
11.1 Graph Theory and Network Flows	139
11.2 Graphs	139
11.2.1 Drawing Graphs	139
11.3 Definitions	142
11.4 Shortest Path	145
11.5 Spanning Trees	154
11.6 Exercise Answers	158
11.7 Additional Exercises	160
11.7.1 Notes	168

III Integer Programming	169
12 Integer Programming Formulations	173
12.1 Knapsack Problem	173
12.2 Capital Budgeting	176
12.3 Set Covering	178
12.3.1 Covering (Generalizing Set Cover)	182
12.4 Assignment Problem	182
12.5 Facility Location	183
12.5.1 Capacitated Facility Location	184
12.5.2 Uncapacitated Facility Location	185
12.6 Basic Modeling Tricks - Using Binary Variables	185
12.6.1 Connecting to continuous variables	187
12.6.2 Exact absolute value	188
12.6.2.1 Exact 1-norm	188
12.6.2.2 Maximum	189
12.7 Network Flow	189
12.7.1 Example - Multicommodity Flow	189
12.7.2 Corresponding optimization problems	190
12.7.3 Relation to other problems	190
12.7.4 Usage	191
12.8 Transportation Problem	191
12.9 Other examples	191
12.10 Notes from AIMMS modeling book.	191
12.10.1 Further Topics	192
12.11 MIP Solvers and Modeling Tools	192
13 Algorithms and Complexity	193
13.1 Big-O Notation	194
13.2 Algorithms - Example with Bubble Sort	197
13.2.1 Sorting	197
13.3 Problem, instance, size	202
13.3.1 Problem, instance	202
13.3.2 Format and examples of problems/instances	202
13.3.3 Size of an instance	202
13.4 Complexity Classes	203
13.4.1 P	204
13.4.2 NP	205
13.4.3 Problem Reductions	206
13.4.4 NP-Hard	206
13.4.5 NP-Complete	207
13.5 Problems and Algorithms	208
13.5.1 Matching Problem	209
13.5.1.1 Greedy Algorithm for Maximal Matching	210

13.5.1.2	Other algorithms to look at	210
13.5.2	Minimum Spanning Tree	210
13.5.3	Kruskal's algorithm	211
13.5.3.1	Prim's Algorithm	212
13.5.4	Traveling Salesman Problem	212
13.5.4.1	Nearest Neighbor - Construction Heuristic	212
13.5.4.2	Double Spanning Tree - 2-Apx	213
13.5.4.3	Christofides - Approximation Algorithm - (3/2)-Apx	214
14	Introduction to computational complexity	215
14.1	Introduction	215
14.2	Problem, instance, size	215
14.2.1	Problem, instance	215
14.2.2	Format and examples of problems/instances	216
14.2.3	Size of an instance	216
14.3	Algorithms, running time, Big-O notation	216
14.3.1	Basics	216
14.3.2	Worst-time complexity	217
14.3.3	Big-O notation	217
14.3.4	Examples	217
14.4	Basics	217
14.5	Complexity classes	218
14.5.1	Polynomial time problems	218
14.5.2	Non-deterministic polynomial time problems	218
14.5.3	Complements of problems in NP	219
14.6	Relationship between the classes	219
14.6.1	A basic result	219
14.6.2	An \$1,000,000 open question	219
14.7	Comparing problems, Polynomial time reductions	219
14.8	Comparing problems, Polynomial time reductions	220
14.8.1	Definition	220
14.8.2	Basic properties	221
14.9	NP-Completeness	221
14.9.1	The basics	221
14.9.2	Do NP-complete problems exist?	222
14.10	NP-Hardness	222
14.11	Exercises	222
15	Exponential Size Formulations	229
15.1	Cutting Stock	229
15.1.1	Pattern formulation	231
15.1.2	Column Generation	233
15.1.3	Cutting Stock - Multiple widths	234
15.2	Spanning Trees	235

15.3 Traveling Salesman Problem	235
15.3.1 Miller Tucker Zemlin (MTZ) Model	237
15.3.2 Dantzig-Fulkerson-Johnson (DFJ) Model	242
15.3.3 Traveling Salesman Problem - Branching Solution	245
15.3.4 Traveling Salesman Problem Variants	245
15.3.4.1 Many salespersons (m-TSP)	245
15.3.4.2 TSP with order variants	248
15.4 Vehicle Routing Problem (VRP)	248
15.4.1 Case Study: Bus Routing in Boston	248
15.5 Steiner Tree Problem	249
15.6 Literature and other notes	249
15.6.1 Google maps data	250
15.6.2 TSP In Excel	250
16 Algorithms to Solve Integer Programs	251
16.1 LP to solve IP	251
16.1.1 Rounding LP Solution can be bad!	252
16.1.2 Rounding LP solution can be infeasible!	252
16.1.3 Fractional Knapsack	252
16.2 Branch and Bound	252
16.2.1 Algorithm	253
16.2.2 Knapsack Problem and 0/1 branching	255
16.2.3 Traveling Salesman Problem solution via Branching	257
16.3 Cutting Planes	257
16.3.1 Chvátal Cuts	259
16.3.2 Gomory Cuts	260
16.3.3 Cover Inequalities	262
16.4 Branching Rules	263
16.5 Lagrangian Relaxation for Branch and Bound	263
16.6 Benders Decomposition	263
16.7 Literature	264
16.8 Other material for Integer Linear Programming	264
Exercises	268
Solutions	268
16.8.1 Other discrete problems	269
16.8.2 Assignment Problem and the Hungarian Algorithm	269
16.8.3 History of Computation in Combinatorial Optimization	269
17 Heuristics for TSP	271
17.1 Construction Heuristics	272
17.1.1 Random Solution	272
17.1.2 Nearest Neighbor	272
17.1.3 Insertion Method	272
17.2 Improvement Heuristics	273

17.2.1 2-Opt (Subtour Reversal)	273
17.2.2 3-Opt	274
17.2.3 k -Opt	274
17.3 Meta-Heuristics	274
17.3.1 Hill Climbing (2-Opt for TSP)	274
17.3.2 Simulated Annealing	276
17.3.3 Tabu Search	278
17.3.4 Genetic Algorithms	278
17.3.5 Greedy randomized adaptive search procedure (GRASP)	278
17.3.6 Ant Colony Optimization	278
17.4 Computational Comparisons	279
IV Nonlinear Programming	281
18 Non-linear Programming (NLP)	285
18.1 Convex Sets	286
18.2 Convex Functions	288
18.2.1 Proving Convexity - Characterizations	291
18.2.2 Proving Convexity - Composition Tricks	291
18.3 Convex Optimization Examples	292
18.3.1 Unconstrained Optimization: Linear Regression	292
18.4 Machine Learning - SVM	294
18.4.0.1 Feasible separation	295
18.4.0.2 Distance between hyperplanes	295
18.4.0.3 SVM	297
18.4.0.4 Approximate SVM	298
18.4.1 SVM with non-linear separators	298
18.4.2 Support Vector Machines	300
18.5 Classification	301
18.5.1 Machine Learning	301
18.5.2 Neural Networks	301
18.6 Box Volume Optimization in Scipy.Minimize	301
18.7 Modeling	301
18.7.1 Minimum distance to circles	302
18.8 Machine Learning	305
18.9 Machine Learning - Supervised Learning - Regression	305
18.10 Machine learning - Supervised Learning - Classification	306
18.10.1 Python SGD implementation and video	306
19 NLP Algorithms	307
19.1 Algorithms Introduction	307
19.2 1-Dimensional Algorithms	307
19.2.1 Golden Search Method - Derivative Free Algorithm	308

19.2.1.1 Example:	309
19.2.2 Bisection Method - 1st Order Method (using Derivative)	309
19.2.2.1 Minimization Interpretation	309
19.2.2.2 Root finding Interpretation	309
19.2.3 Gradient Descent - 1st Order Method (using Derivative)	310
19.2.4 Newton's Method - 2nd Order Method (using Derivative and Hessian)	310
19.3 Multi-Variate Unconstrained Optimizaiton	311
19.3.1 Descent Methods - Unconstrained Optimization - Gradient, Newton	311
19.3.1.1 Choice of α_t	311
19.3.1.2 Choice of d_t using $\nabla f(x)$	311
19.3.2 Stochastic Gradient Descent - The mother of all algorithms.	312
19.3.3 Neural Networks	313
19.3.4 Choice of Δ_k using the hessian $\nabla^2 f(x)$	313
19.4 Constrained Convex Nonlinear Programming	314
19.4.1 Barrier Method	314
20 Computational Issues with NLP	317
20.1 Irrational Solutions	317
20.2 Discrete Solutions	317
20.3 Convex NLP Harder than LP	317
20.4 NLP is harder than IP	318
20.5 Karush-Huhn-Tucker (KKT) Conditions	319
20.6 Gradient Free Algorithms	321
20.6.1 Needler-Mead	321
21 Material to add...	323
21.0.1 Bisection Method and Newton's Method	323
21.1 Gradient Descent	323
22 Fairness in Algorithms	325
23 One-dimensional Optimization	327
24 Gradient Descent Methods	337
A Linear Algebra	349

Todo list

■ General notes about the book.....	1
■ Chapter 1. Resources and Notation	
90% complete. Goal 80% completion date: Done	
Notes:	3
■ Chapter 2. Mathematical Programming	
50% complete. Goal 80% completion date: July 20	
Notes:	7
■ Add discussion of Optimization, Operations Research, and Mathematical Programming including background and applications. Also, give an introduction to the content in this book, what you will learn by working though the book, and why this book is interesting and different from other sources.	7
■ Describe applications and andd images	8
■ Fill in this section with formulas and discuss applications.	12
■ Part I: Linear Programming	
Notes: This Part applies to DORI. We hope for 80% completion by January 2023, and 100% completion for January 2024	15
■ Chapter 3. Modeling: Linear Programming	
50% complete. Goal 80% completion date: July 20	
Notes:	17
■ Section 3.1. Modeling and Assumptions in Linear Programming	
20% complete. Goal 80% completion date: July 20	
Notes: Clean up this section. Describe process of modeling a problem.	18
■ Section 3.2. Examples	
40% complete. Goal 80% completion date: July 20	
Notes: Clean up this section. Finish describing several of the problems, give examples for all problem classes and attach code to all examples.	21
■ Add mathematical model	28
■ Fill in this subsection	29
■ Fill in this subsection	29
■ Section 3.3. Modeling Tricks	
40% complete. Goal 80% completion date: July 20	
Notes: Only one modeling trick listed here. Discuss absolute value application and also making a free variable non-negative.	34
■ Chapter 4. Graphically Solving Linear Programs	
50% complete. Goal 80% completion date: July 20	
Notes:	37

■ Section 4.1. Nonempty and Bounded Problem	20% complete. Goal 80% completion date: July 20	
	Notes: Need to work on this section.	37
■ Section 4.2. Infinitely Many Optimal Solutions	20% complete. Goal 80% completion date: July 20	
	Notes: Need to work on this section.	41
■ Section 4.3. Problems with No Solution	20% complete. Goal 80% completion date: July 20	
	Notes: Need to work on this section.	44
■ Section 4.4. Problems with Unbounded Feasible Regions	20% complete. Goal 80% completion date: July 20	
	Notes: Need to work on this section.	46
■ To do: add contours to plot to show extreme point is the optimal solution.	47
■ Section 4.5. Formal Mathematical Statements	20% complete. Goal 80% completion date: July 20	
	Notes: Need to work on this section.	51
■ Chapter 5. Software - Excel	10% complete. Goal 80% completion date: July 20	
	Notes:	59
■ Chapter 6. Software - Python	10% complete. Goal 80% completion date: July 20	
	Notes:	61
■ Chapter 7. Simplex Method	10% complete. Goal 80% completion date: July 20	
	Notes: This section hasn't been cleaned at all. This needs to be looked at and cleaned up.	103
■ Chapter 8. Duality	0% complete. Goal 80% completion date: July 20	
	Notes: This is a borrowed section. Likely we should update this to match our CC-BY-SA 4.0 license. Also, update all content to match notation in the book.	111
■ Chapter 9. Sensitivity Analysis	0% complete. Goal 80% completion date: July 20	
	Notes: Need to write this section. Add examples from lecture notes. Create code to help generate examples.	125
■ Chapter 10. Multi-Objective Optimization	10% complete. Goal 80% completion date: July 20	
	Notes: Clean up this section. Add more information.	127
■ Chapter 11. Graph Algorithms	10% complete. Goal 80% completion date: July 20	
	Notes:	139
■ Write this section.	139
■ Part II: Integer Programming	Notes: This Part applies to DORII. Ideally it will be ready for September 2022.	171

Chapter 12. Integer Programming Formulations	
70% complete. Goal 80% completion date: August 20	
Notes:	173
Add flight crew scheduling example and images.	179
Include picture and example data	183
Fix up this section	189
Add discussion of transportation problem and picture.	191
Chapter 13. Algorithms and Complexity	
60% complete. Goal 80% completion date: August 20	
Notes:	193
INCLUDE PICTURES OF MATCHINGS	209
Chapter 14. Introduction to computational complexity	
Move this section to mode advanced version of the book.	215
Chapter 15. Exponential Size Formulations	
60% complete. Goal 80% completion date: August 20	
Notes:	229
Chapter 16. Algorithms to Solve Integer Programs	
50% complete. Goal 80% completion date: September 20	
Notes:	251
D	257
Chapter 17. Heuristics for TSP	
50% complete. Goal 80% completion date: October 20	
Notes:	271
Part III: Nonlinear Programming	
Notes: This Part applies to DORII. Ideally, it will be ready for November 2022.	283
Chapter 18. Non-linear Programming (NLP)	
50% complete. Goal 80% completion date: November 20	
Notes:	285
Chapter 19. NLP Algorithms	
50% complete. Goal 80% completion date: November 20	
Notes:	307
Chapter 20. Computational Issues with NLP	
50% complete. Goal 80% completion date: November 20	
Notes:	317
Chapter 21. Material to add...	
Decide if we want this material.	323
Chapter 22. Fairness in Algorithms	
Decide if we want to include this chapter or not. No material currently written for it.	325
Chapter 23. One-dimensional Optimization	
Todo: Adapt and incorporate this material.	327
Chapter 24. Gradient Descent Methods	
Todo: Adapt and incorporate this material.	337
Chapter 1. Linear Algebra	
Decide which material to add here.	349

General notes about the book....

1. Resources and Notation

Chapter 1. Resources and Notation

90% complete. Goal 80% completion date: Done

Notes:

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

FREE NOTES AND TEXTBOOKS

- 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 O'Reilly.com.
- Lesson files for the Winston Book
- Video instructions for solver and an example workbook



GUROBI LINKS

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

HOW TO PROVE THINGS

- Hammack - Book of Proof

STATISTICS

- Open Stax - Introductory Statistics

LINEAR ALGEBRA

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

REAL ANALYSIS

- Mathematical Analysis I by Elias Zakon

DISCRETE MATHEMATICS, GRAPHS, ALGORITHMS, AND COMBINATORICS

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

PROGRAMMING WITH PYTHON

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

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

Notation

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

6 ■ Resources and Notation

- 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

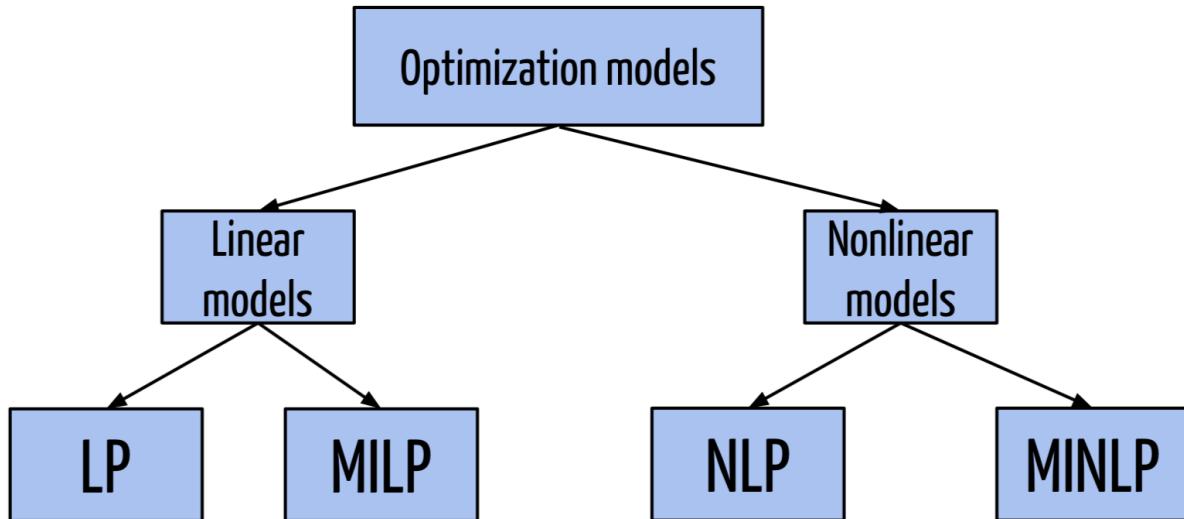
Chapter 2. Mathematical Programming

50% complete. Goal 80% completion date: July 20

Notes:

Add discussion of Optimization, Operations Research, and Mathematical Programming including background and applications. Also, give an introduction to the content in this book, what you will learn by working though the book, and why this book is interesting and different from other sources.

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



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

Along with each problem class, we will associate a complexity class for the general version of the problem. See section 13 for a discussion of complexity classes. Although we will often state that input data for a problem comes from \mathbb{R} , when we discuss complexity of such a problem, we actually mean that the data is rational, i.e., from \mathbb{Q} , and is given in binary encoding.

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

2.1 Linear Programming (LP)

Describe applications and add images

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

Linear Programming (LP):

Polynomial time (P)

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

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

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

Linear Programming (LP) Standard Form:

Polynomial time (P)

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

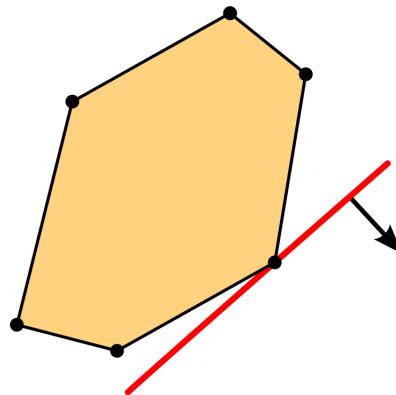
$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{2.2}$$

Figure 2.2

Exercise 2.1:

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

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



© wiki/File/linear-programming.png²

Figure 2.2: Linear programming constraints and objective.

2.2 Mixed-Integer Linear Programming (MILP)

Mixed-integer linear programming will be the focus of Sections 12, 15, 16, and ???. Recall that the notation \mathbb{Z} means the set of integers and the set \mathbb{R} means the set of real numbers. The first problem of interest here is a *binary integer program* (BIP) where all n variables are binary (either 0 or 1).

Binary Integer programming (BIP):

NP-Complete

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

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

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

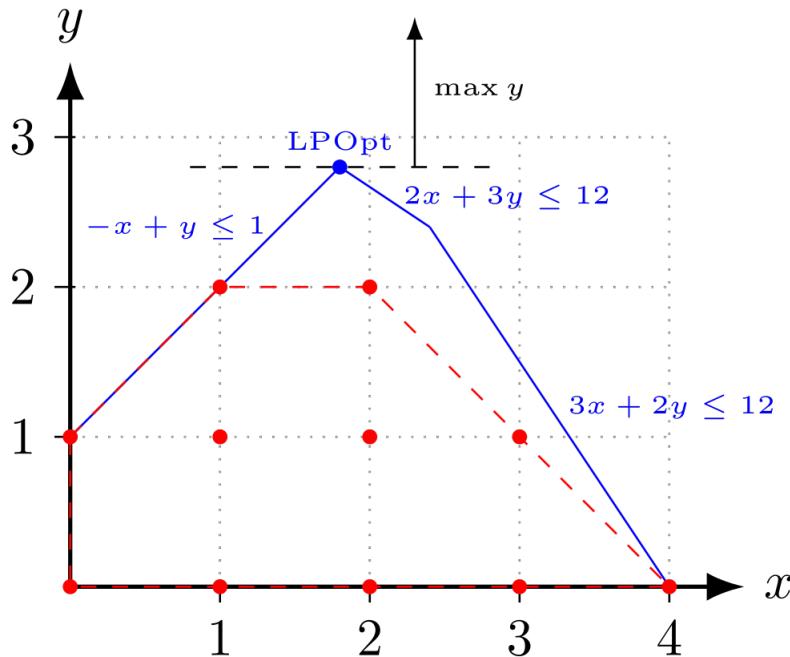
Figure 2.3

Integer Linear Programming (ILP):

NP-Complete

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

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



© wiki/File/integer-programming.png³

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

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

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

2.3 Non-Linear Programming (NLP)

NLP:

NP-Hard

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

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

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

2.3.1. Convex Programming

Here the functions are all **convex**!

Convex Programming:

Polynomial time (P) (typically)

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

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

Example 2.2

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

2.3.2. Non-Convex Non-linear Programming

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

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

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

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

Binary Integer programming (BIP) as a NLP:

NP-Hard

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

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

2.4 Mixed-Integer Non-Linear Programming (MINLP)

Fill in this section with formulas and discuss applications.

2.4.1. Convex Mixed-Integer Non-Linear Programming

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

Part I

Linear Programming

Part I: Linear Programming

Notes: This Part applies to DORI. We hope for 80% completion by January 2023, and 100% completion for January 2024

3. Modeling: Linear Programming

Chapter 3. Modeling: Linear Programming

50% complete. Goal 80% completion date: July 20

Notes:

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 chemical 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 units 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

Section 3.1. Modeling and Assumptions in Linear Programming

20% complete. Goal 80% completion date: July 20

Notes: Clean up this section. Describe process of modeling a problem.

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:

$$z(x_1, \dots, x_n) = c_1x_1 + \dots + c_nx_n \tag{3.8}$$

Lemma 3.3: Linear Function

If $z : \mathbb{R}^n \rightarrow \mathbb{R}$ is linear then for all $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ and for all scalar constants $\alpha \in \mathbb{R}$ we have:

$$z(\mathbf{x}_1 + \mathbf{x}_2) = z(\mathbf{x}_1) + z(\mathbf{x}_2) \quad (3.9)$$

$$z(\alpha \mathbf{x}_1) = \alpha z(\mathbf{x}_1) \quad (3.10)$$

Exercise 3.4

Prove Lemma 3.1.1.

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*. These problems are outside the scope of this course, but there is a vast literature dealing with them [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 3.5

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?]

Exercise 3.6: Stochastic Objective

Suppose the costs are not known with certainty but instead a probability distribution for each value of c_i ($i = 1, \dots, n$) is known. Suggest a way of constructing a linear program from the probability distributions.

[Hint: Suppose I tell you that I'll give you a uniformly random amount of money between \$1 and \$2. How much money do you expect to receive? Use the same reasoning to answer the question.]

3.2 Examples

Section 3.2. Examples

40% complete. Goal 80% completion date: July 20

Notes: Clean up this section. Finish describing several of the problems, give examples for all problem classes and attach code to all 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:

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

**Example 3.7: The Diet Problem**

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 3.8: The Next Diet Problem

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.

Solution. Sets:

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

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

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:

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



Example 3.9: Work Scheduling Problem

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 3.10: LP Burger - extended

LP Burger has changed it's 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 our total cost.

Add mathematical model

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

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

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

Cost	Task 1	Task 2	Task 3
Person 1	40	47	80
Person 2	72	36	58
Person 3	24	61	71

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

Fill in this subsection

3.2.5.1. Production Planning

3.2.5.2. Crop Planning

3.2.6. Mixing Problems

3.2.7. Financial Planning

Fill in this subsection

3.2.8. Network Flow

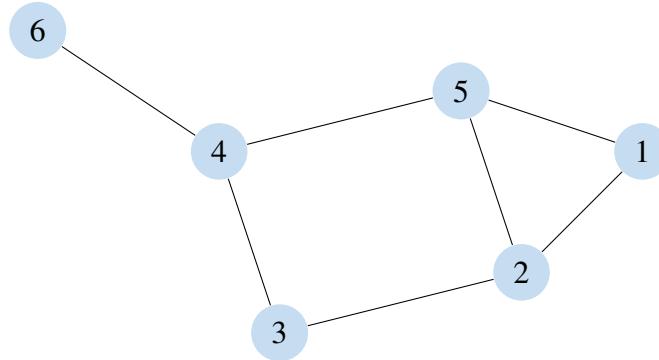
Resources

- 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

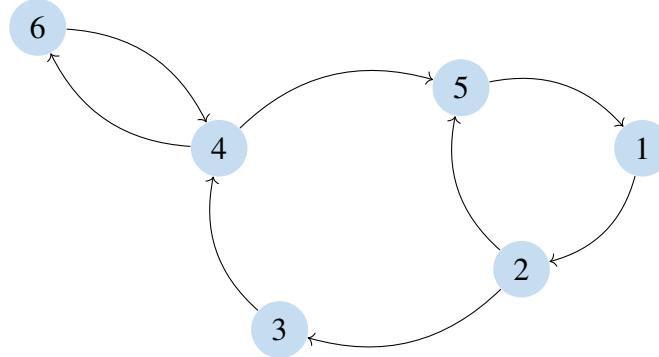
A graph $G = (V, E)$ is defined by a set of vertices V and a set of edges E that contains pairs of vertices. For example, the following graph G can be described by the vertex set $V = \{1, 2, 3, 4, 5, 6\}$ and the edge set $E = \{(4, 6), (4, 5), (5, 1), (1, 2), (2, 5), (2, 3), (3, 4)\}$.



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

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

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

3.2.8.2. Maximum Flow Problem

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

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

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

$$\text{minimize} \quad \sum_{u \rightarrow v} \ell_{u \rightarrow v} \cdot x_{u \rightarrow v}$$

$$\text{subject to } \sum_u x_{u \rightarrow s} - \sum_w x_{s \rightarrow w} = 1$$

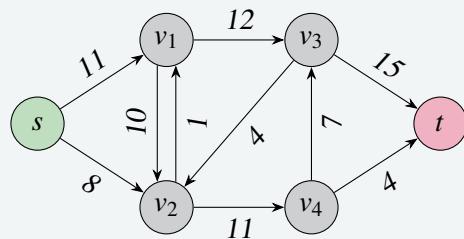
$$\sum_u x_{u \rightarrow t} - \sum_w x_{t \rightarrow w} = -1$$

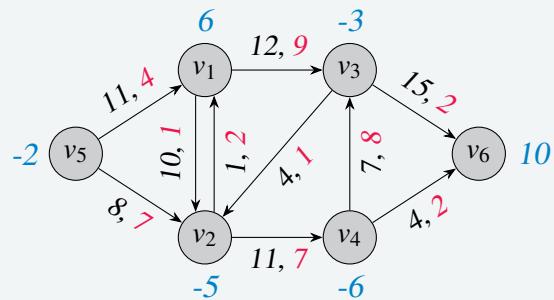
$$\sum_u x_{u \rightarrow v} - \sum_w x_{v \rightarrow w} = 0 \quad \text{for every vertex } v \neq s, t$$

$$x_{u \rightarrow v} \geq 0 \quad \text{for every edge } u \rightarrow v$$

SHORTEST PATH PROBLEM

Example 3.11: Max flow example



Example 3.12: Min Cost Network Flow

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

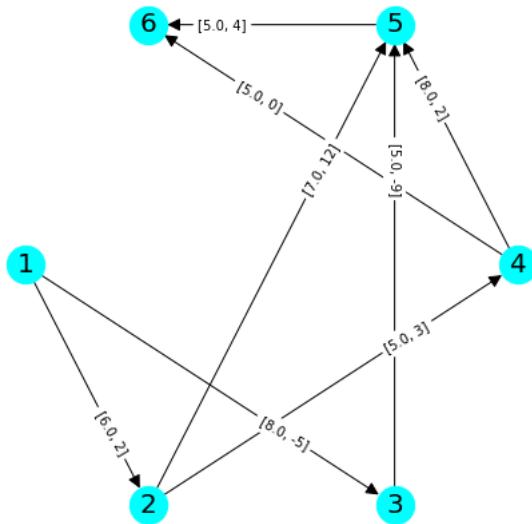


Figure 3.1: network-flow²

²network-flow, from **network-flow**. **network-flow**, **network-flow**.

³network-flow-solution, from **network-flow-solution**. **network-flow-solution**, **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:

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

Theorem 3.13: 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 3.14

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

Section 3.3. Modeling Tricks

40% complete. Goal 80% completion date: July 20

Notes: Only one modeling trick listed here. Discuss absolute value application and also making a free variable non-negative.

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

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

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

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

Example 3.15: 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 - Corneujoles, Tütüncü

4. Graphically Solving Linear Programs

Chapter 4. Graphically Solving Linear Programs

50% complete. Goal 80% completion date: July 20

Notes:

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

Section 4.1. Nonempty and Bounded Problem

20% complete. Goal 80% completion date: July 20

Notes: Need to work on this section.

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

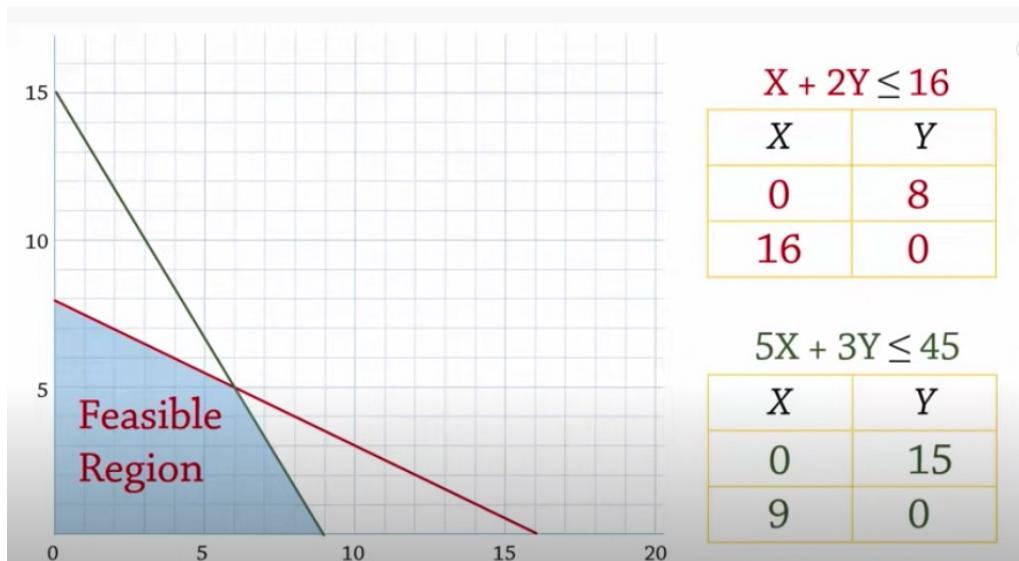
¹Special thanks to Joshua Emmanuel and Christopher Griffin for sharing their content to help put this section together. Proper citations and references are forthcoming.

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

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.

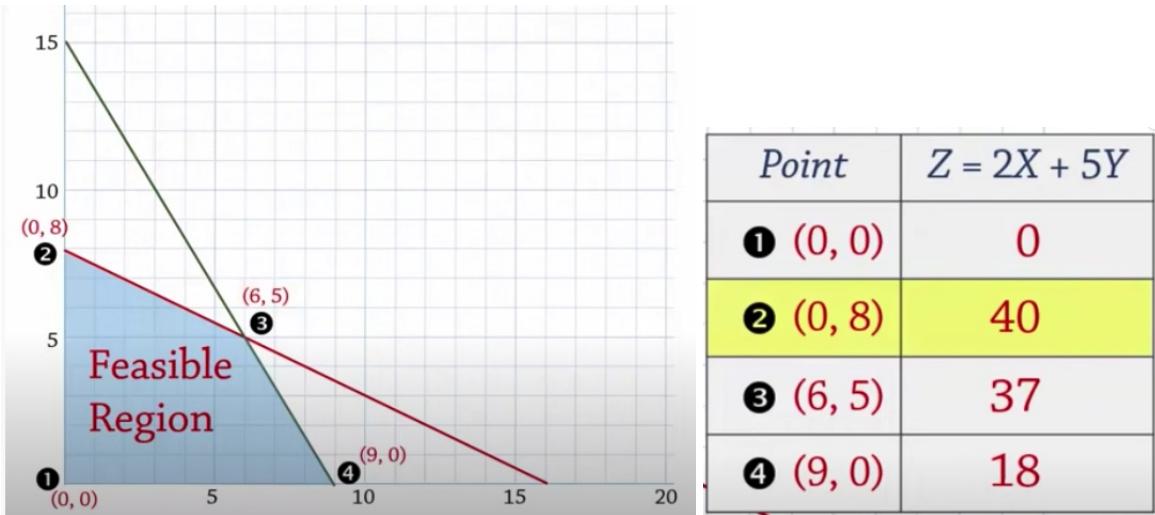


Notice that the feasible region is nonempty (it has points that satisfy all the inequalities) and also that it is bounded (the feasible points don't continue infinitely in any direction).

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.

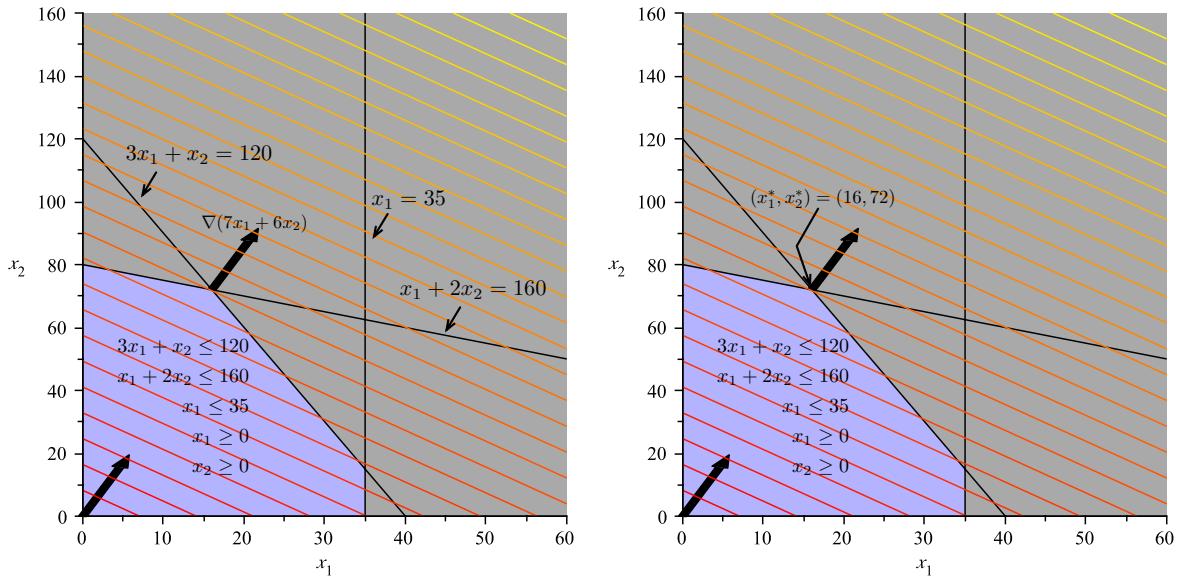


Figure 4.1: Feasible Region and Level Curves of the Objective Function: The shaded region in the plot is the feasible region and represents the intersection of the five inequalities constraining the values of x_1 and x_2 . On the right, we see the optimal solution is the “last” point in the feasible 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

Section 4.2. Infinitely Many Optimal Solutions

20% complete. Goal 80% completion date: July 20

Notes: Need to work on this section.

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.

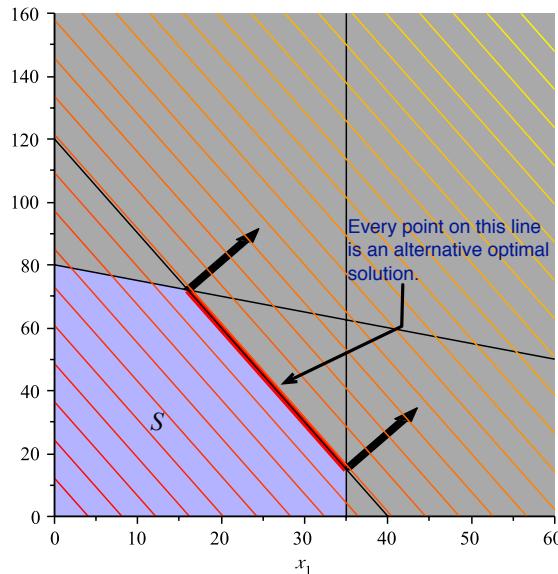


Figure 4.2: An example of infinitely many alternative optimal solutions in a linear programming problem. The level curves for $z(x_1, x_2) = 18x_1 + 6x_2$ are parallel to one face of the polygon boundary of the feasible region. Moreover, this side contains the points of greatest value for $z(x_1, x_2)$ 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

Section 4.3. Problems with No Solution

20% complete. Goal 80% completion date: July 20

Notes: Need to work on this section.

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.

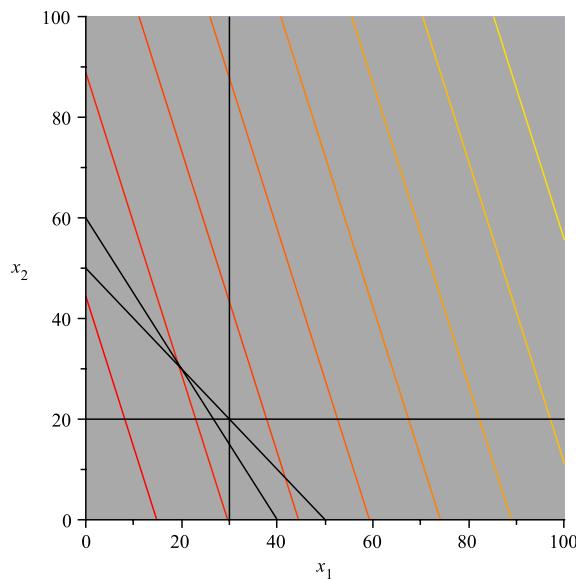


Figure 4.3: A Linear Programming Problem with no solution. The feasible region of the linear programming problem is empty; that is, there are no values for x_1 and x_2 that can simultaneously satisfy all the constraints. Thus, no solution exists.

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

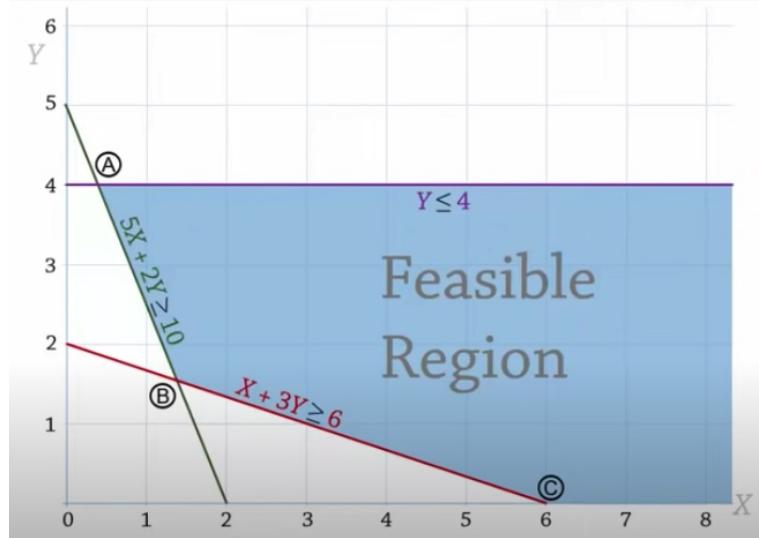
Section 4.4. Problems with Unbounded Feasible Regions

20% complete. Goal 80% completion date: July 20

Notes: Need to work on this section.

Consider the problem

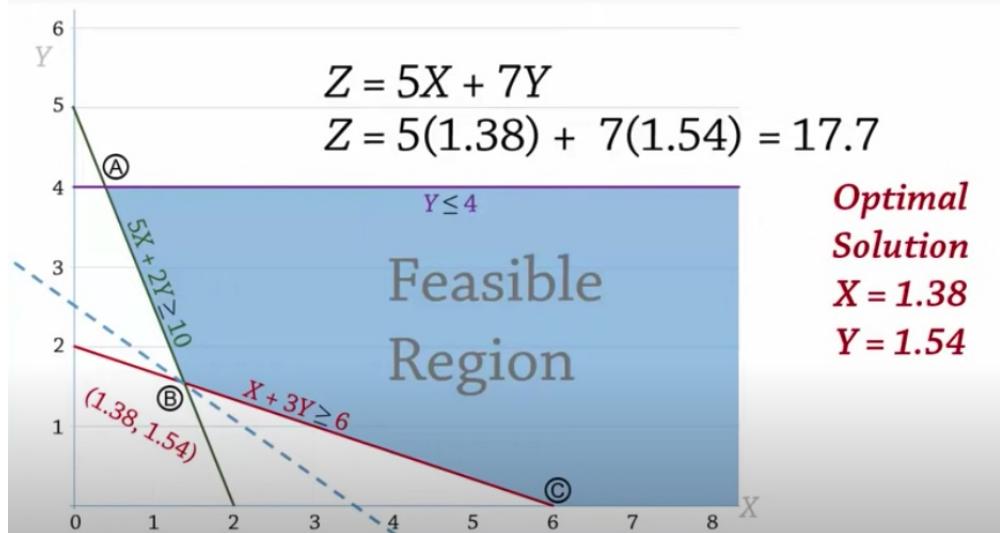
$$\begin{aligned}
 \min \quad & Z = 5X + 7Y \\
 \text{s.t.} \quad & X + 3Y \geq 6 \\
 & 5X + 2Y \geq 10 \\
 & Y \leq 4 \\
 & X, Y \geq 0
 \end{aligned}$$



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.

To do: add contours to plot to show extreme point is the optimal solution.



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.

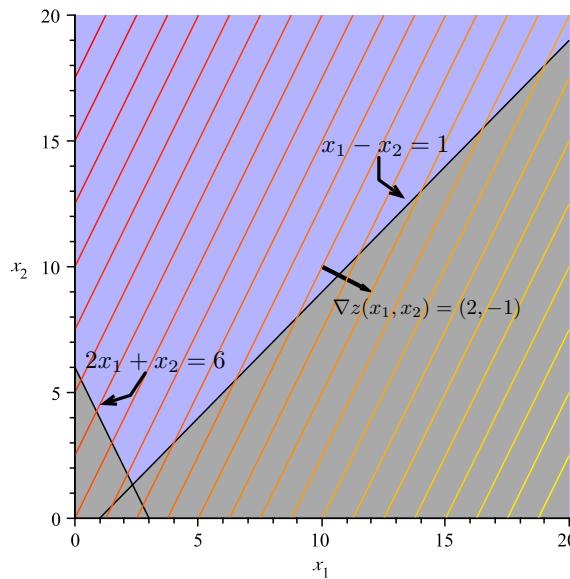


Figure 4.4: A Linear Programming Problem with Unbounded Feasible Region: Note that we can continue to make level curves of $z(x_1, x_2)$ corresponding to larger and larger values as we move down and to the right. These curves will continue to intersect the feasible region for any value of $v = z(x_1, x_2)$ 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:

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

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.

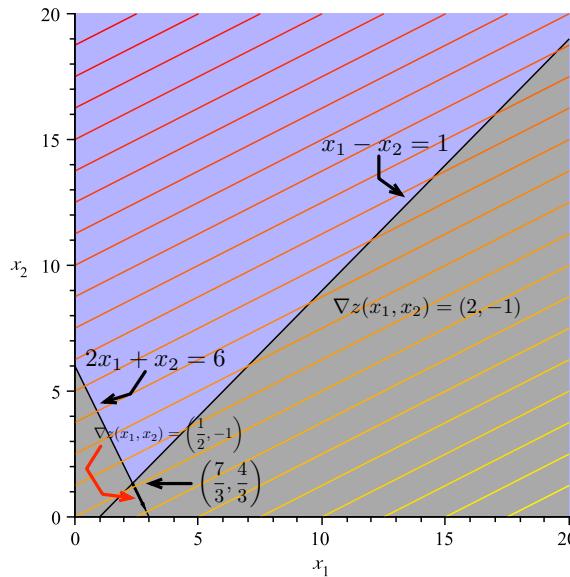


Figure 4.5: A Linear Programming Problem with Unbounded Feasible Region and Finite Solution:
In this problem, the level curves of $z(x_1, x_2)$ increase in a more “southerly” direction than in Example 4.4—that is, *away* from the direction in which the feasible region increases without bound. The point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

Section 4.5. Formal Mathematical Statements

20% complete. Goal 80% completion date: July 20

Notes: Need to work on this section.

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

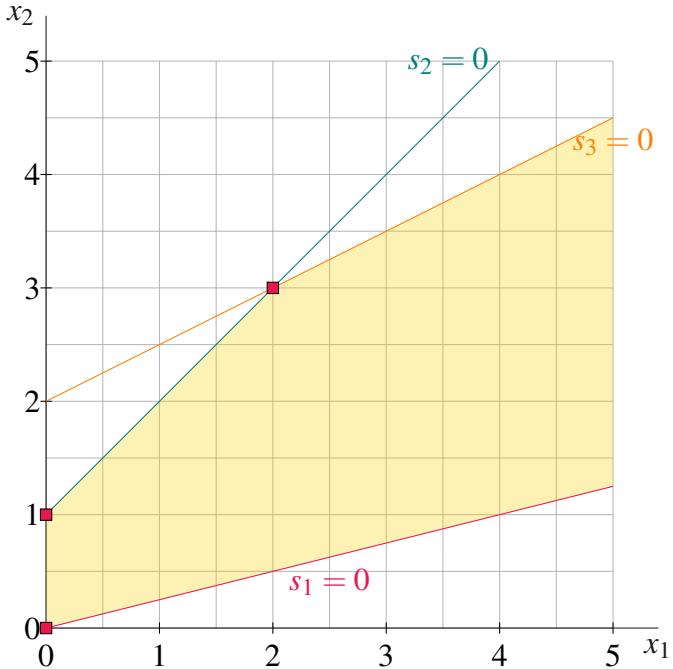
Unbounded Polyhedral Sets: If \mathcal{S} is unbounded, it will have *directions*. \mathbf{d} is a direction of \mathcal{S} only if $\mathbf{Ax} + \lambda\mathbf{d} \leq \mathbf{b}, \mathbf{x} + \lambda\mathbf{d} \geq 0$ for all $\lambda \geq 0$ and all $\mathbf{x} \in \mathcal{S}$. In other words, consider the ray $\{\mathbf{x} : \mathbf{x}_0 + \lambda\mathbf{d}, \lambda \geq 0\}$

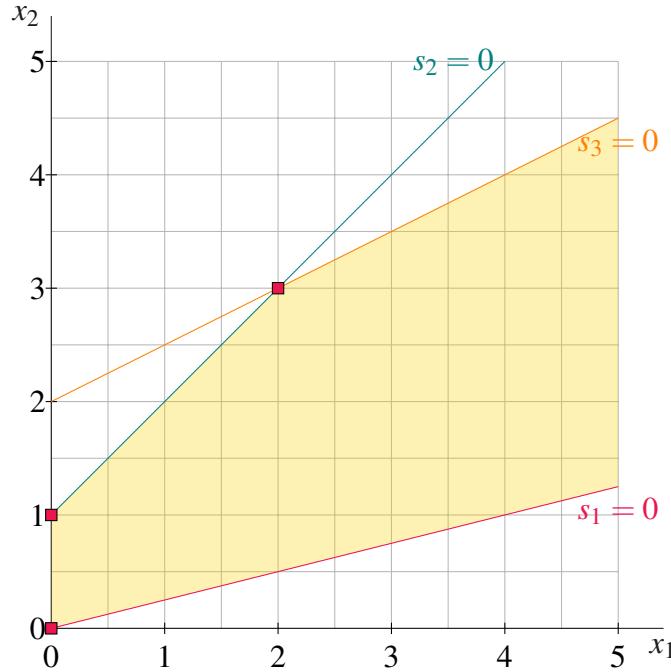
in \mathbb{R}^n , where \mathbf{x}_0 is the vertex and \mathbf{d} is the direction of the ray. $\mathbf{d} \neq 0$ is a **direction** of set \mathcal{S} if for each \mathbf{x}_0 in \mathcal{S} the ray $\{\mathbf{x}_0 + \lambda \mathbf{d}, \lambda \geq 0\}$ also belongs to \mathcal{S} .

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.

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



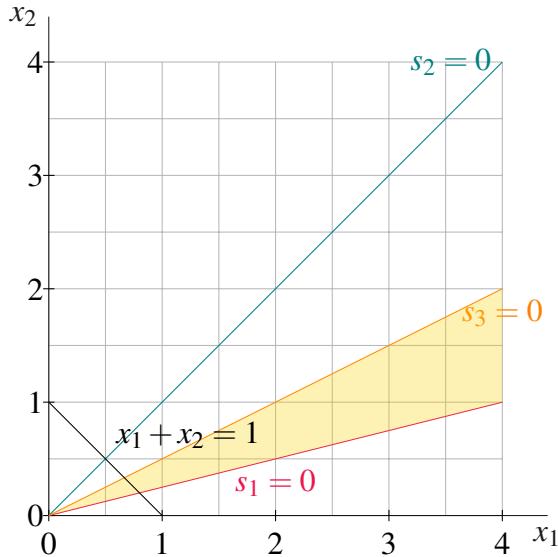


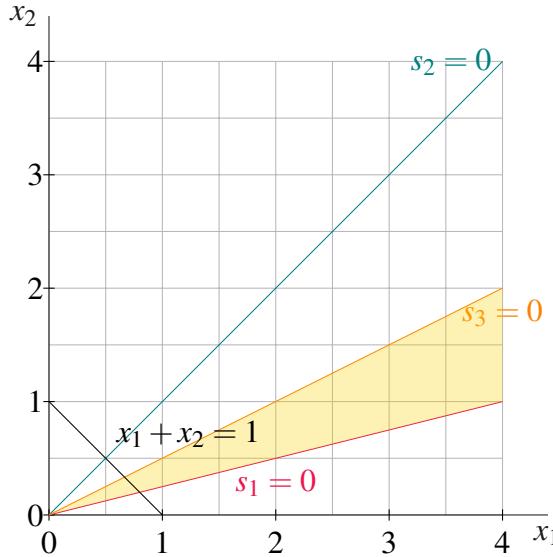
E.g., consider the $s_3 = 0$ (orange) line, to find the extreme direction start at extreme point $(2,3)$ and find another feasible point on the orange line, say $(4,4)$ and subtract $(2,3)$ from $(4,4)$, which yields $(2,1)$.

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.

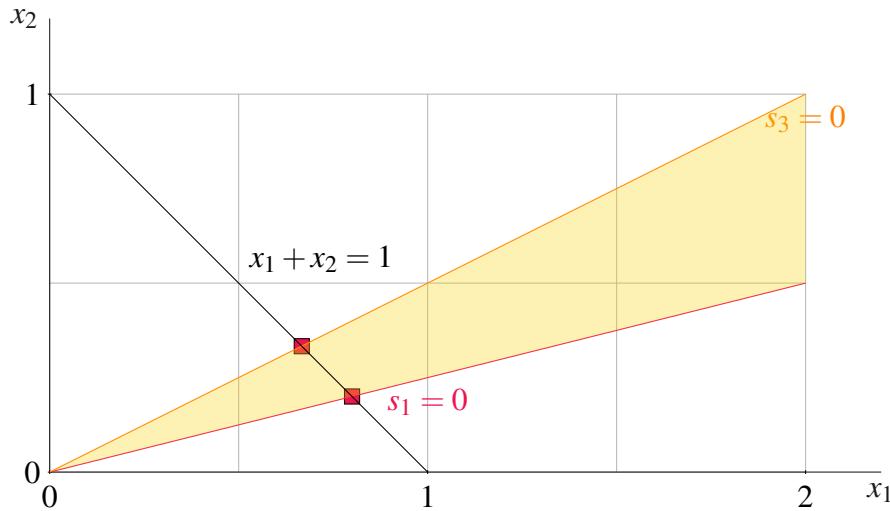
To find the extreme directions we can change the right-hand-side to $\mathbf{b} = 0$, which forms a polyhedral cone (in yellow), and then add the constraint $x_1 + x_2 = 1$. The intersection of the cone and $x_1 + x_2 = 1$ form a line segment.

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





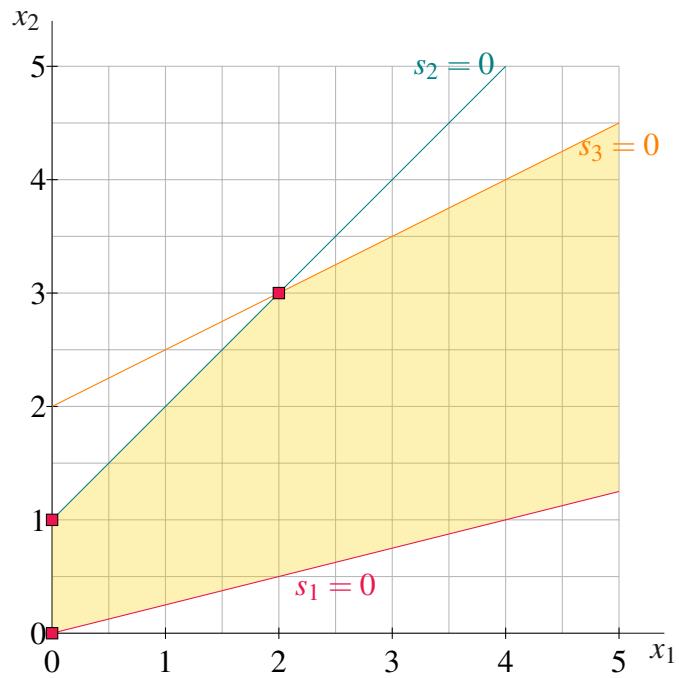
Magnifying for clarity, and removing the $s_2 = 0$ (teal) line, as it is redundant, and marking the extreme points of the new feasible region, $(4/5, 1/5)$ and $(2/3, 1/3)$, with red boxes, we have:



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:

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

5. Software - Excel

Chapter 5. Software - Excel

10% complete. Goal 80% completion date: July 20

Notes:

Resources

- *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

Chapter 6. Software - Python

10% complete. Goal 80% completion date: July 20

Notes:

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 embedded 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 Lab Objective: *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[2,1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax $[a:b]$ can be read as “the a th entry up to (but not including) the b th entry.” Similarly, $[a:]$ means “the a th entry to the end” and $[:b]$ means “everything up to (but not including) the b th entry.”

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

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

Stacking

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

$$A = \begin{bmatrix} \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{blue}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{blue}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{blue}{x} \end{bmatrix} \quad B = \begin{bmatrix} \textcolor{red}{*} & \textcolor{red}{*} & \textcolor{red}{*} \\ \textcolor{red}{*} & \textcolor{red}{*} & \textcolor{red}{*} \\ \textcolor{red}{*} & \textcolor{red}{*} & \textcolor{red}{*} \end{bmatrix}$$

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

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

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()`.

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

$$y = [* * * *]$$

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

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

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

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.

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

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

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

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

4 colors = np.array(["k", "g", "b", "r", "c", "m", "y", "w"])
x = np.linspace(0, 5, 1000)
6 y = np.ones(1000)

8 for i in xrange(8):
    plt.plot(x, i*y, colors[i], linewidth=18)
10
11 plt.ylim([-1, 8])
12 plt.savefig("colors.pdf", format='pdf')
plt.clf()

```

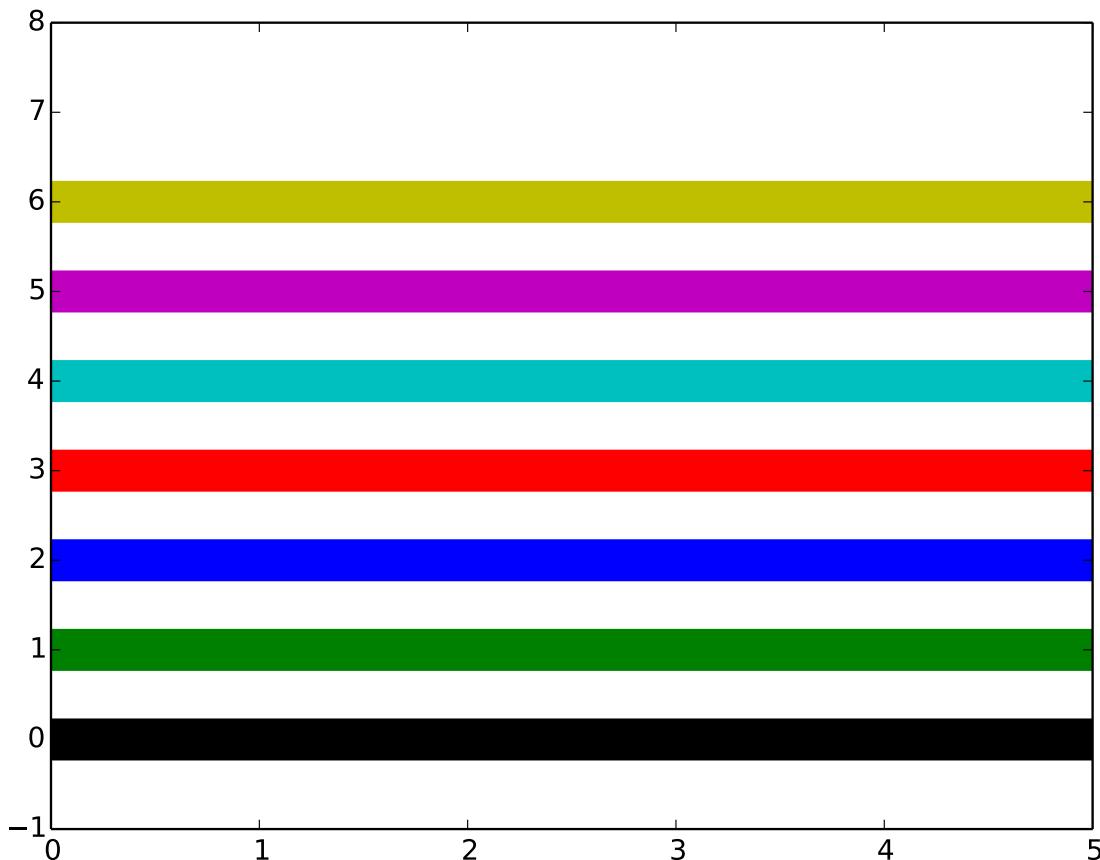
colors.py

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

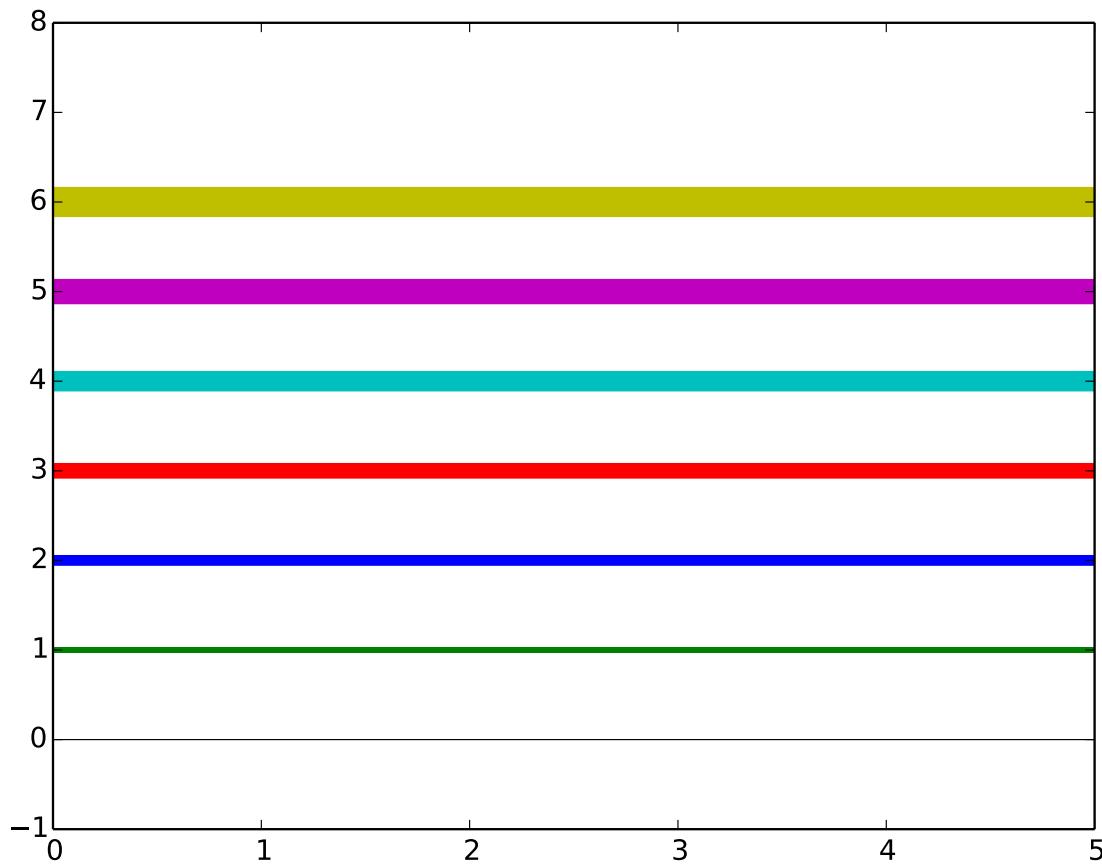


Figure 6.2: plot of varying linewidths.

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

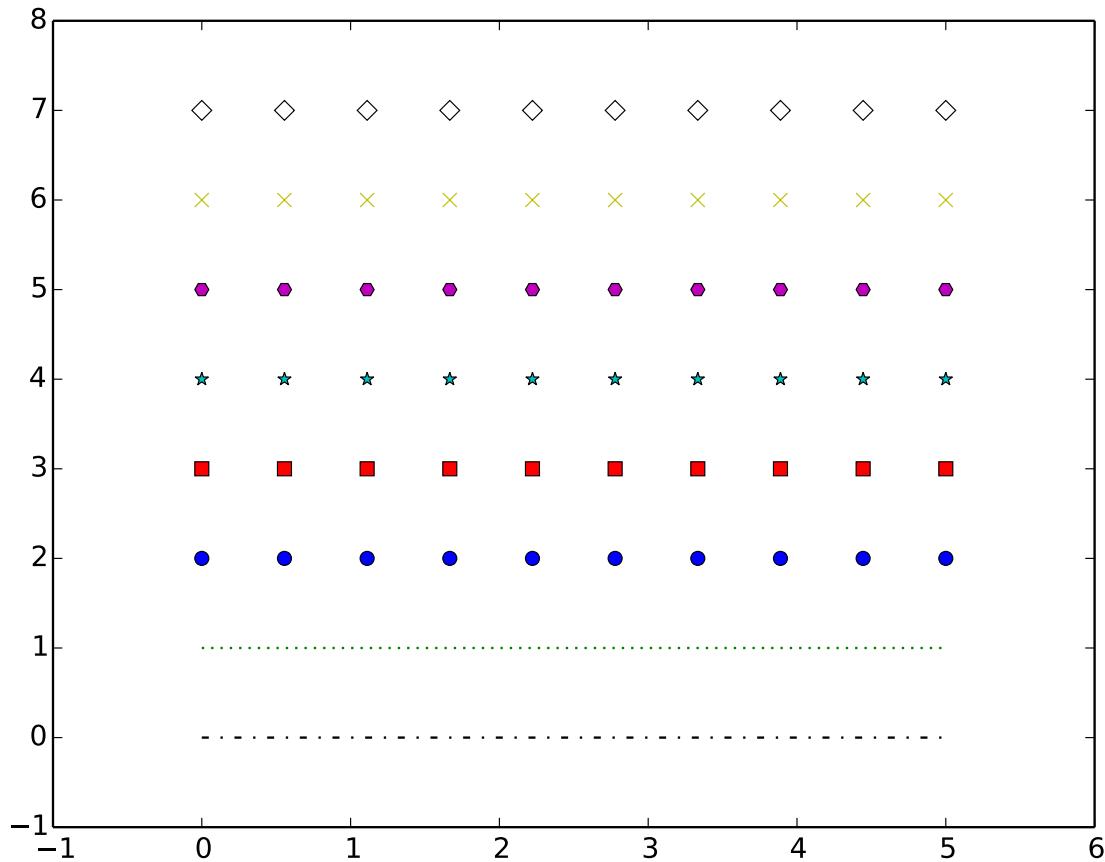


Figure 6.3: plot of varying linestyles.

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])
3
4 plt.title("My Plot of Varying Linestyles", fontsize = 20, color = "gold")
5 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

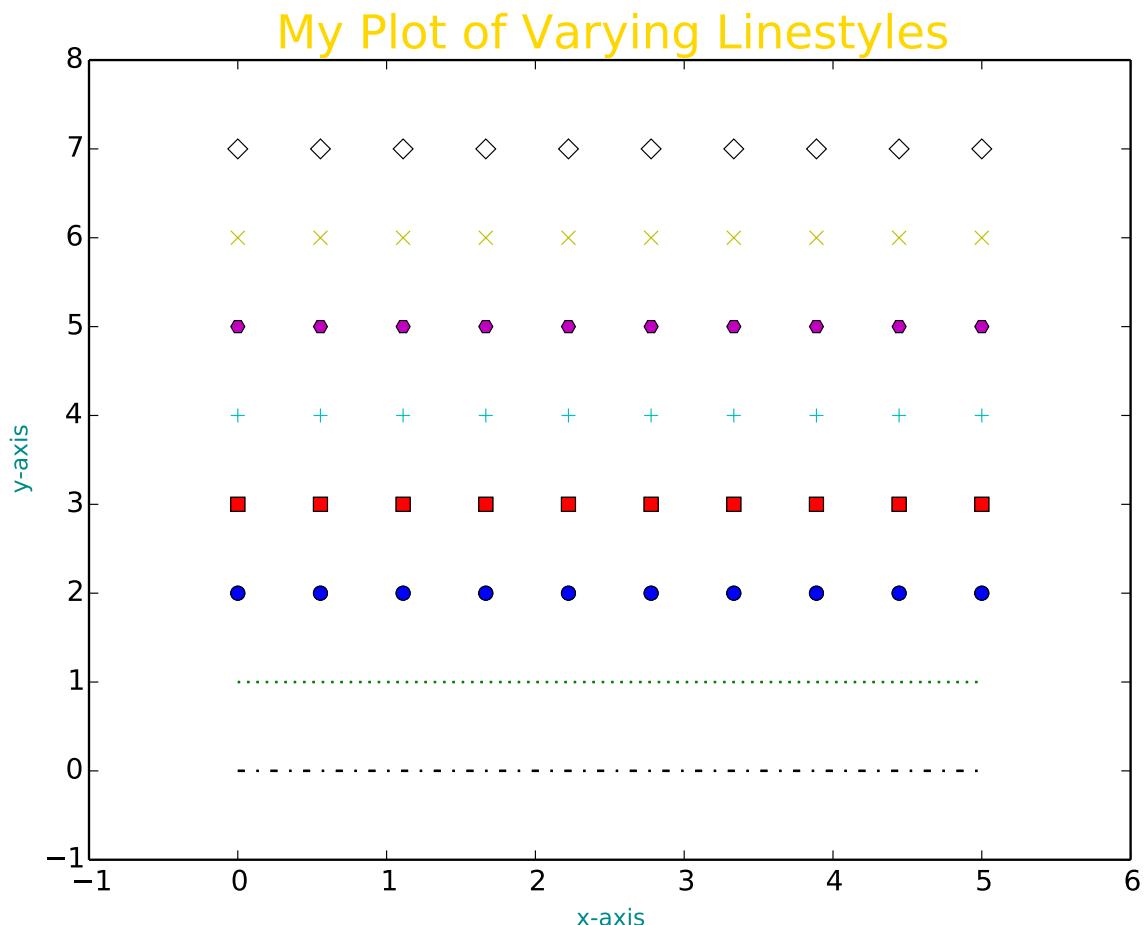


Figure 6.4: plot of varying linestyles using text labels.

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
 $_C1: 10 X1 + 5 X2 \leq 300$

$_C2: 4 X1 + 4 X2 \leq 160$

$_C3: 2 X1 + 6 X2 \leq 180$

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 \leq , $=$, or \geq 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

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

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

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

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

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

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

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

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

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()

```

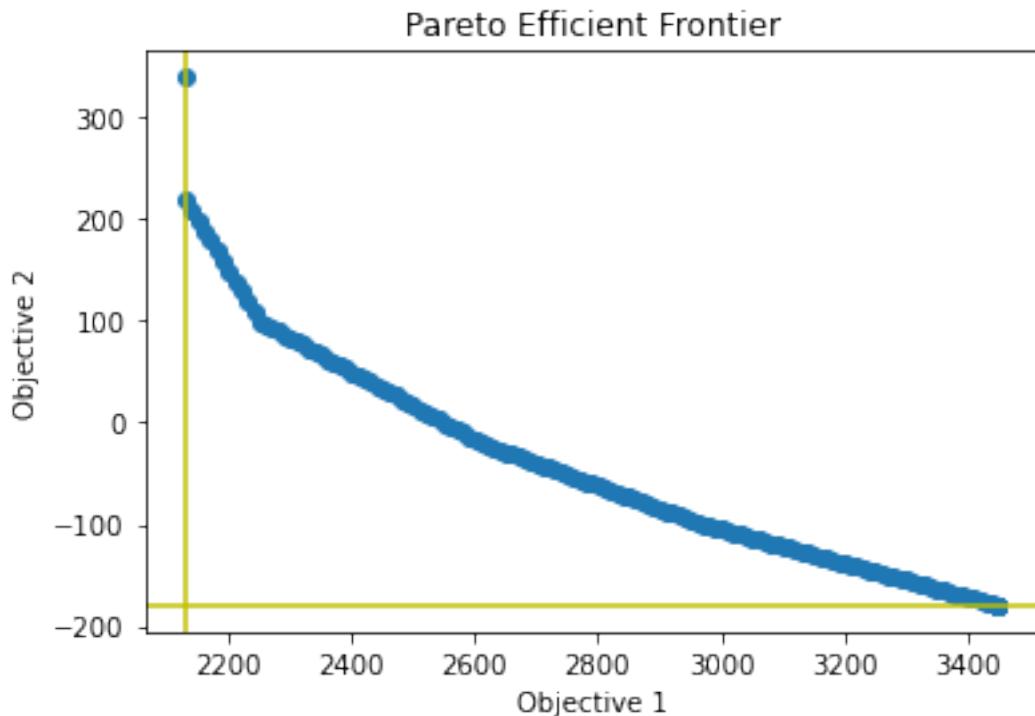
[12]: import matplotlib.pyplot as plt

```

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

Gurobi Log Tools

6.12 Plots, Pandas, and Geopandas

6.12.1. Geopandas

<https://jcutrer.com/python/learn-geopandas-plotting-usmaps>

<https://github.com/joncutter/geopandas-tutorial>

6.13 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.*

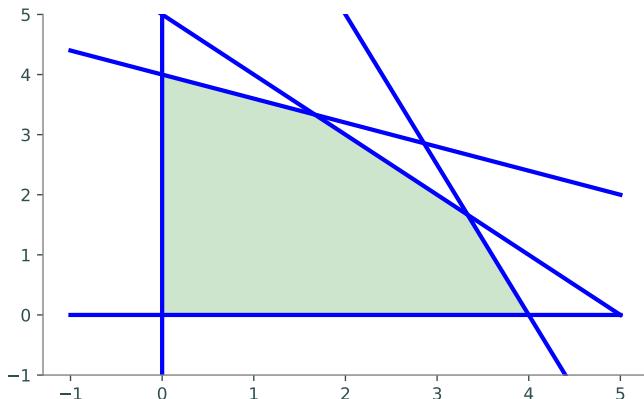
Standard Form

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

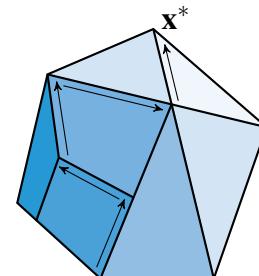
$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{array}$$

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 6.5). 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 6.5: 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.

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

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 6.1: 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

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

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

$$D = \begin{bmatrix} 0 & \bar{\mathbf{c}}^T \\ \mathbf{b} & -\bar{A} \end{bmatrix} \quad (6.1)$$

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 6.2: 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.

6.13.1. 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.

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

Definition 6.3: 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:

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

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

Problem 6.4: 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`.

6.13.2. 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

$$\begin{aligned} x_0 &= 1.6 \\ x_1 &= .2. \end{aligned}$$

Problem 6.5: 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.

6.13.3. 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.

6.13.4. 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.

7. Simplex Method

Chapter 7. Simplex Method

10% complete. Goal 80% completion date: July 20

Notes: This section hasn't been cleaned at all. This needs to be looked at and cleaned up.

Definition 7.1: Standard Form

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

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

Definition 7.2: 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 7.3: 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]



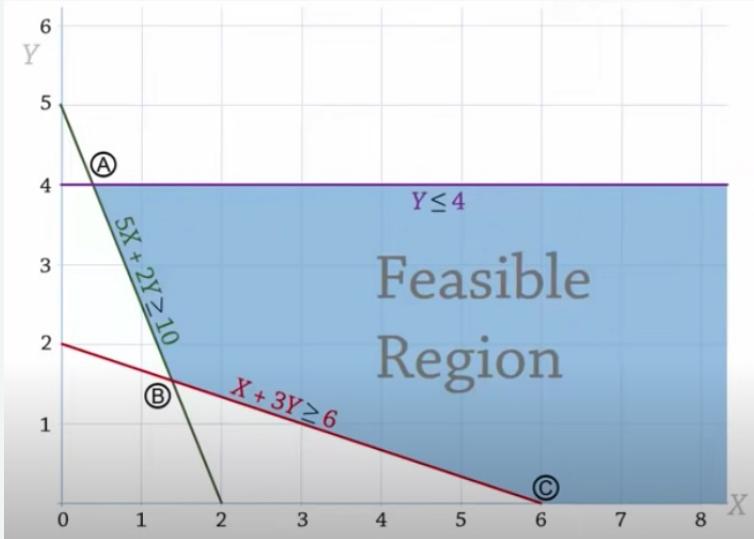
Definition 7.4: Basic solution

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 7.5

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

$$\max \begin{bmatrix} -5 \\ -7 \\ 0 \\ 0 \\ 0 \end{bmatrix}^\top \begin{bmatrix} X \\ Y \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \quad (7.1)$$

$$\begin{bmatrix} 1 & 3 & -1 & 0 & 0 \\ 5 & 2 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 10 \\ 4 \end{bmatrix} \quad (7.2)$$

$$(X, Y, s_1, s_2, s_3) \geq 0 \quad (7.3)$$

Definition 7.6: Basic feasible solution

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

Theorem 7.7: 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 7.8: 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 7.9: 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 unboundedness.

- 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. ♠

7.1 Simplex Method

7.2 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:

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

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).

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

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.

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

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.

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

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 $BV=\{s_1, s_2\}$, 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	-40	-30	0	0	0
0	6	4	1	0	40
0	4	2	0	1	20

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.

8. Duality

Chapter 8. Duality

0% complete. Goal 80% completion date: July 20

Notes: This is a borrowed section. Likely we should update this to match out CC-BY-SA 4.0 license. Also, update all content to match notation in the book.

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.

8.1 The Dual of Linear Program

Suppose that we have the following linear program in maximization standard form:

$$\begin{array}{ll} \text{maxim} & x_1 + 2x_2 + x_3 + x_4 \\ \text{maximize} & \\ \text{subject to} & x_1 + 2x_2 + x_3 \leq 2 \\ & x_2 + x_4 \leq 1 \\ & x_1 + 2x_3 \leq 1 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \\ & x_3 \geq 0 \end{array}$$

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

$$\begin{aligned} & (a_{1,1}y_1 + \dots + a_{m,1}y_m) \cdot x_1 \\ & + \dots \\ & + (a_{1,n}y_1 + \dots + a_{m,n}y_m) \cdot x_n \\ & \leq y_1b_1 + \dots + y_mb_m \end{aligned}$$

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

$$\begin{aligned} &\text{minimize} && b_1y_1 + \cdots + b_my_m \\ &\text{subject to} && \\ &&& a_{1,1}y_1 + \cdots + a_{m,1}y_m \geq c_1 \\ &&& \vdots \\ &&& a_{n,1}y_1 + \cdots + a_{m,n}y_m \geq c_n \\ &&& y_1 \geq 0 \\ &&& \vdots \\ &&& y_m \geq 0 \end{aligned}$$

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

$$\begin{aligned} &\mathbf{c}^T \mathbf{x} \\ &\text{maximize} \\ &\text{subject to} \\ &\quad A\mathbf{x} \leq \mathbf{b} \\ &\quad \mathbf{x} \geq 0 \end{aligned}$$

is a linear program in maximization standard form, then its dual is the minimization linear program

$$\begin{aligned} & \text{minimize}_{\mathbf{y}} && \mathbf{b}^T \mathbf{y} \\ & \text{subject to} && \\ & && A^T \mathbf{y} \geq \mathbf{c} \\ & && \mathbf{y} \geq 0 \end{aligned}$$

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

$$\begin{aligned} & \text{minimize}_{\mathbf{y}} && \mathbf{c}^T \mathbf{y} \\ & \text{subject to} && \\ & && A \mathbf{y} \geq \mathbf{b} \\ & && \mathbf{y} \geq 0 \end{aligned}$$

in an equivalent way as

$$\begin{aligned} & \text{mubject to}_{\mathbf{y}} && -\mathbf{c}^T \mathbf{y} \\ & \text{maximize} && \\ & && -A \mathbf{y} \leq -\mathbf{b} \\ & && \mathbf{y} \geq 0 \end{aligned}$$

If we compute the dual of the above program we get

$$\begin{aligned} & \text{mubject to}_{\mathbf{z}} && -\mathbf{b}^T \mathbf{z} \\ & \text{minimize} && \\ & && -A^T \mathbf{z} \geq -\mathbf{c} \\ & && \mathbf{z} \geq 0 \end{aligned}$$

that is,

$$\begin{aligned} & \text{maximize}_{\mathbf{z}} && \mathbf{b}^T \mathbf{z} \\ & \text{subject to} && \\ & && A^T \mathbf{z} \leq \mathbf{c} \\ & && \mathbf{z} \geq 0 \end{aligned}$$

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.

8.2 Linear programming duality

Consider the following problem:

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b}. \end{aligned} \tag{8.1}$$

In the remark at the end of Chapter ??, we saw that if (8.1) 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 (8.1).

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 (8.1). 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

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{A} = \mathbf{c}^T \\ & \mathbf{y} \geq 0. \end{aligned} \tag{8.2}$$

Note that (8.2) is a linear programming problem! We call it the **dual problem** of the **primal problem** (8.1). 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 (8.1) to be the linear programming problem (8.2). 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 8.1: strong-duality-special

Suppose that (8.1) has an optimal solution. Then (8.2) 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 (8.1) 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:

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{A}'\mathbf{x} \leq \mathbf{b}' \\ & \mathbf{A}''\mathbf{x} = \mathbf{b}'' \end{aligned} \tag{8.3}$$

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 (8.1) 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 (8.3) 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:

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} + \mathbf{y}'^T \mathbf{b}' + \mathbf{y}''^T \mathbf{b}'' \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{A} + \mathbf{y}'^T \mathbf{A}' + \mathbf{y}''^T \mathbf{A}'' = \mathbf{c}^T \\ & \mathbf{y} \geq 0 \\ & \mathbf{y}' \leq 0. \end{aligned} \tag{8.4}$$

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 (8.3). The details are left as an exercise.

Observation 2

Consider the primal problem of the following form:

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{Ax} \geq \mathbf{b} \\ & x_i \geq 0 \quad i \in P \\ & x_i \leq 0 \quad i \in N \end{aligned} \tag{8.5}$$

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 (8.5) as defined under Observation 1, we obtain the dual problem

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{a}^{(i)} = c_i \quad i \in \{1, \dots, n\} \setminus (P \cup N) \\ & \mathbf{y}^T \mathbf{a}^{(i)} + p_i = c_i \quad i \in P \\ & \mathbf{y}^T \mathbf{a}^{(i)} + q_i = c_i \quad i \in N \\ & p_i \geq 0 \quad i \in P \\ & q_i \leq 0 \quad i \in N \end{aligned} \tag{8.6}$$

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 $q_i, i \in N$:

$$\begin{aligned} \max \quad & \mathbf{y}^\top \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^\top \mathbf{a}^{(i)} = c_i \quad i \in \{1, \dots, n\} \setminus (P \cup N) \\ & \mathbf{y}^\top \mathbf{a}^{(i)} \leq c_i \quad i \in P \\ & \mathbf{y}^\top \mathbf{a}^{(i)} \geq c_i \quad i \in N, \end{aligned} \tag{8.7}$$

which can be taken as the dual problem of (8.5) instead of (8.6). The advantage here is that it has fewer variables than (8.6).

Hence, the dual problem of

$$\begin{aligned} \min \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

is simply

$$\begin{aligned} \max \quad & \mathbf{y}^\top \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^\top \mathbf{A} \leq \mathbf{c}^\top \\ & \mathbf{y} \geq 0. \end{aligned}$$

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.

8.2.1. 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:

$$\begin{array}{llllll} \min & x_1 & - & 2x_2 & + & 3x_3 \\ \text{s.t.} & -x_1 & & & + & 4x_3 = 5 \\ & 2x_1 & + & 3x_2 & - & 5x_3 \geq 6 \\ & & & & 7x_2 & \leq 8 \\ & x_1 & & & & \geq 0 \\ & & x_2 & & & \text{free} \\ & & & x_3 & & \leq 0. \end{array}$$

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:

$$\begin{array}{lll} \max & 5y_1 + 6y_2 + 8y_3 \\ \text{s.t.} & -y_1 + 2y_2 & \leq 1 \\ & 3y_2 + 7y_3 & = -2 \\ & 4y_1 - 5y_2 & \geq 3 \\ & y_1 & \text{free} \\ & y_2 & \geq 0 \\ & y_3 & \leq 0. \end{array}$$

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 8.2 as stated below. The details are left as an exercise.

Theorem 8.2: 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 8.2.1 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

$$\begin{aligned} \min \quad & c_1x_1 + c_2x_2 \\ \text{s.t.} \quad & 2x_1 + x_2 \geq 2 \\ & x_1 + 3x_2 \geq 1. \end{aligned}$$

has an optimal solution. Justify your answer

Solutions

1. The dual is

$$\begin{aligned} \max \quad & 3y_1 \\ \text{s.t.} \quad & y_1 + 3y_2 = 4 \\ & 2y_1 - 4y_2 \leq -2 \\ & y_1 \geq 0. \end{aligned}$$

2. The dual is

$$\begin{aligned} \max \quad & y_1 \\ \text{s.t.} \quad & y_1 + y_2 \leq 0 \\ & y_1 \leq 3 \\ & 2y_1 - 3y_2 \leq 1 \\ & y_1 \quad \text{free} \\ & y_2 \leq 0. \end{aligned}$$

3. The dual is

$$\begin{aligned} \max \quad & y_1 \\ \text{s.t.} \quad & y_1 \geq 1 \\ & -3y_1 = 0 \\ & 2y_1 \leq -9 \\ & y_1 \quad \text{free.} \end{aligned}$$

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

$$\begin{aligned} \max \quad & 2y_1 + y_2 \\ & 2y_1 + y_2 = c_1 \\ & y_1 + 3y_2 = c_2 \\ & y_1, y_2 \geq 0. \end{aligned}$$

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

$$\begin{aligned}\frac{3}{5}c_1 - \frac{1}{5}c_2 &\geq 0 \\ -\frac{1}{5}c_1 + \frac{2}{5}c_2 &\geq 0,\end{aligned}$$

or more simply,

$$\frac{1}{3}c_2 \leq c_1 \leq 2c_2.$$

9. Sensitivity Analysis

Chapter 9. Sensitivity Analysis

0% complete. Goal 80% completion date: July 20

Notes: Need to write this section. Add examples from lecture notes. Create code to help generate examples.

10. Multi-Objective Optimization

Chapter 10. Multi-Objective Optimization

10% complete. Goal 80% completion date: July 20

Notes: Clean up this section. Add more information.

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 \(Pymoo\)](#)

10.1 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.

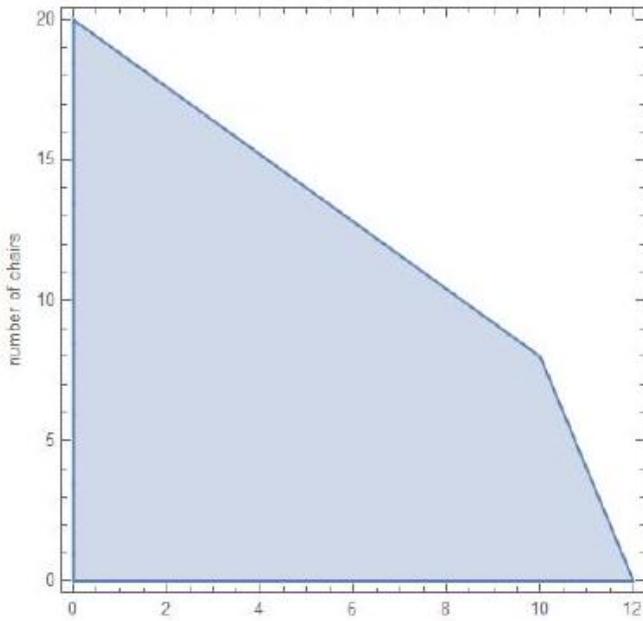


The manufacturer has an ongoing deal with a foreign sawmill which supplies them with 960 and 200 board-feet (bdft) of bocote and rosewood respectively each month.

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:

$$\text{Maximize } 8000x + 2000y$$

$$\text{s.t. } 80x + 20y \leq 960$$

$$12x + 10y \leq 200$$

$$x, y \geq 0$$

-4	-1	0	0	0
4	1	1	0	48
6	5	0	1	100

$$\text{Maximize } 4x + y$$

$$\text{s.t. } 4x + y + s_1 = 48$$

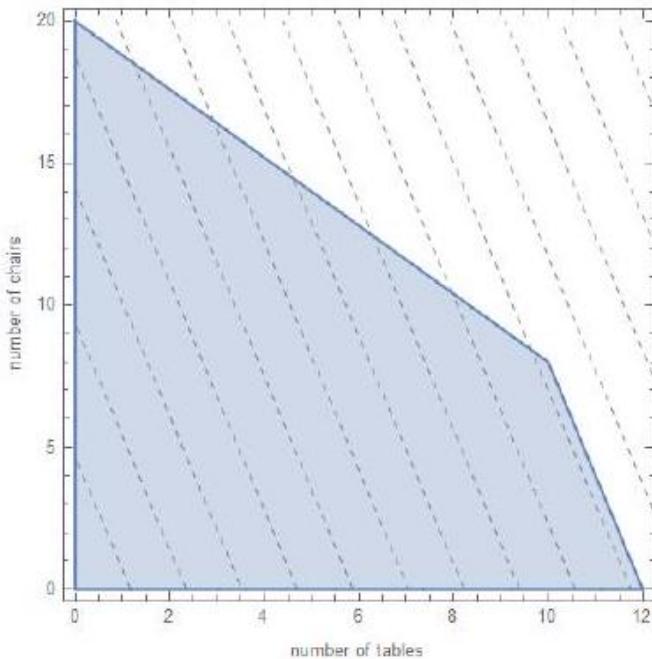
$$\begin{array}{ll} f & 6x + 5y + s_2 = 100 \\ \hline 2000 & x, y \geq 0 \end{array}$$

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

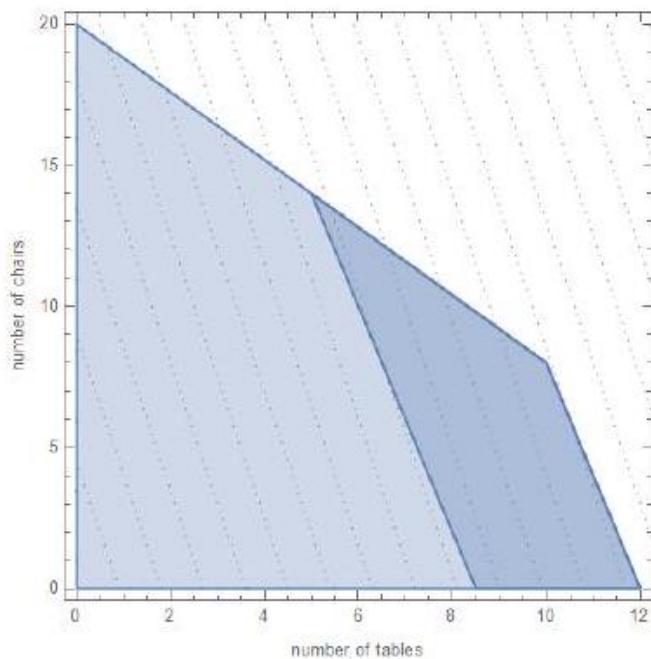
$$\arg \min \{10x + 2y : (x, y) \in \{(12, 0), (10, 8)\}\}$$

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.

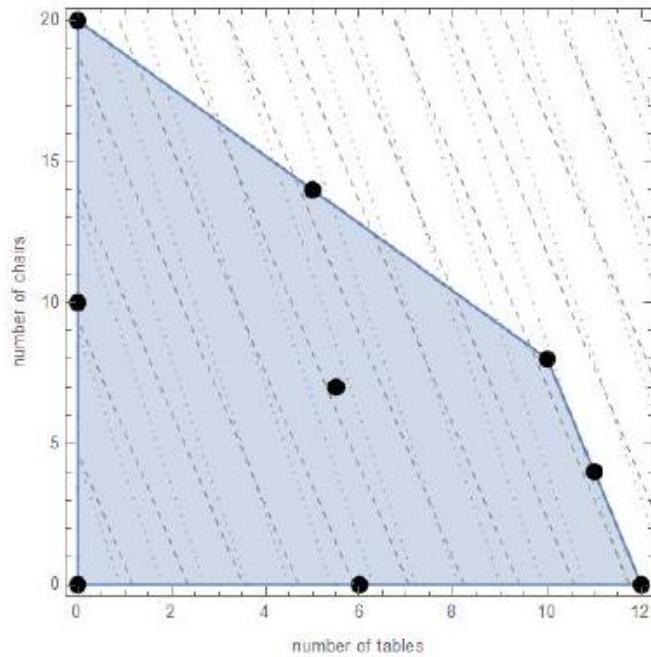
Interestingly, our rollover solution is NOT an extreme point to the ORIGINAL feasible region. Given a set P and some number of functions $f_i : P \rightarrow \mathbb{R}$ that we seek to maximize, we call a point $\mathbf{x} \in P$ Pareto Optimal or Efficient if there does not exist another point $\bar{\mathbf{x}} \in P$ 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?

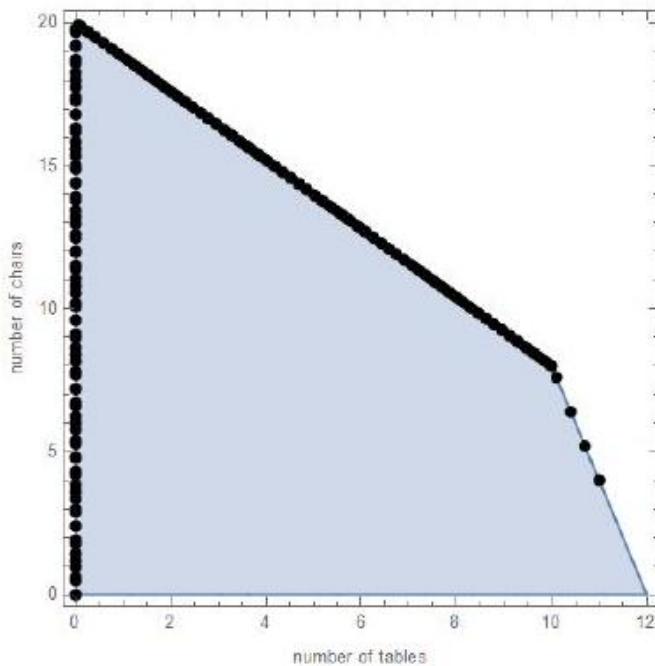
What is the frontier of this problem?



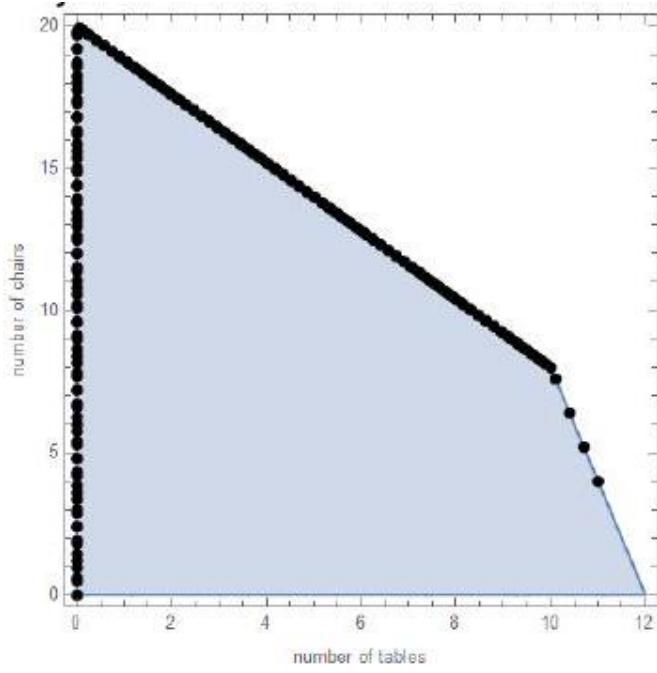
The rollover method is generalized in Goal Programming

By varying α , it is possible to generate many distinct efficient solutions.

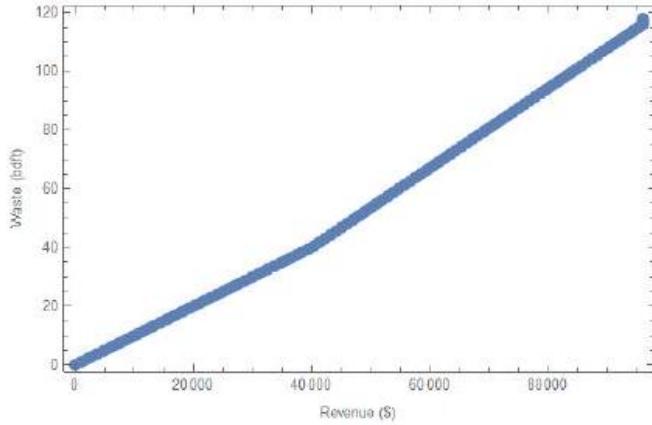
However, this method can generate inefficient solutions if the underlying model is poorly constructed.



It is more common to see a Pareto frontier plotted with respect to its objectives.



number of table



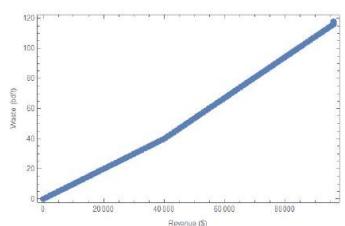
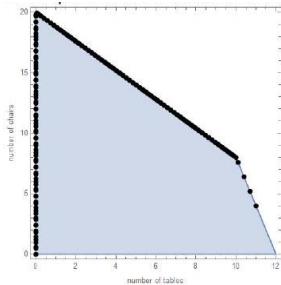
One of the owners of our manufactory decides to explore possible planning himself; he implements the multi-objective method that he remembers, Scalarization by picking some arbitrary constant $\lambda \in [0, 1]$ and combining his two objectives like so:

$$\begin{aligned} \text{Minimize} \quad & \lambda(8000x + 2000y) + (1 - \lambda)(10x + 2y) \\ \text{s.t.} \quad & 80x + 20y \leq 960 \\ & 12x + 10y \leq 200 \\ & x, y \geq 0 \end{aligned}$$

What is the benefit of this method?

Where does it fall short?

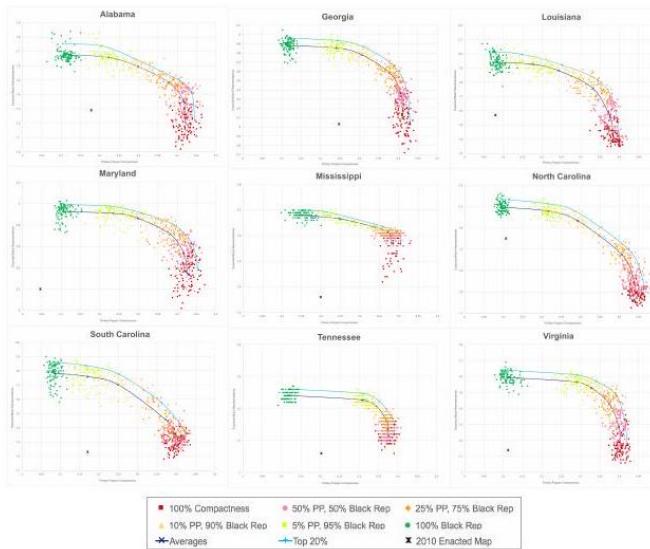
10.2 What points will the Scalarization method find if we vary λ ?



These are all nice ideas, but the problem presented above is neither difficult nor practical.

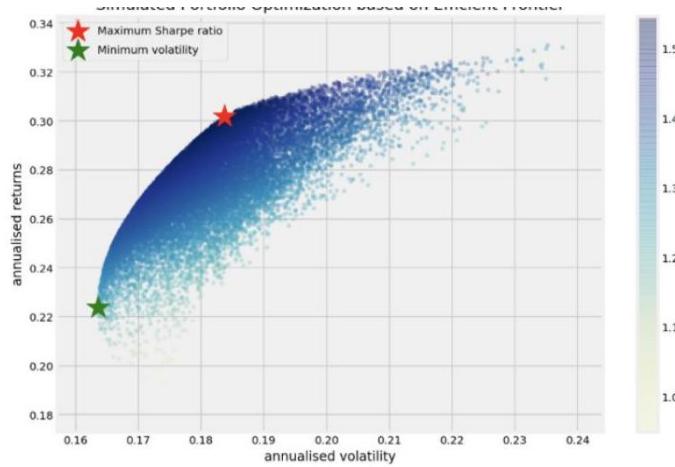
What are some areas that a Pareto frontier would be actually useful?

10.3 Political Redistricting [3]

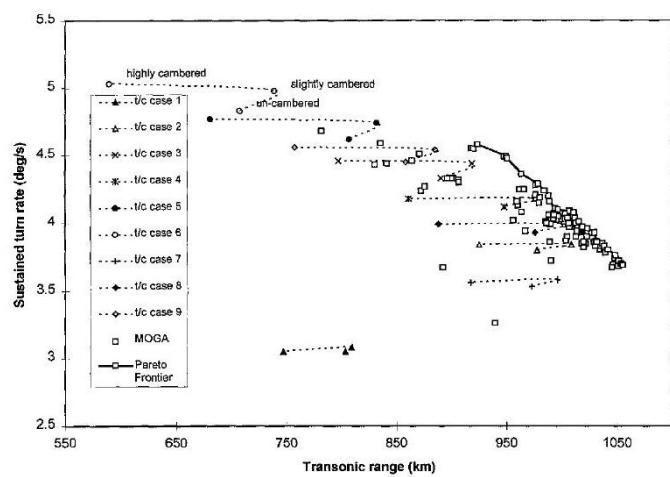


10.4 Portfolio Optimization [5]

10.5 Simulated Portfolio Optimization based on Efficient Frontier



10.6 Aircraft Design [1]



10.7 Vehicle Dynamics [4]

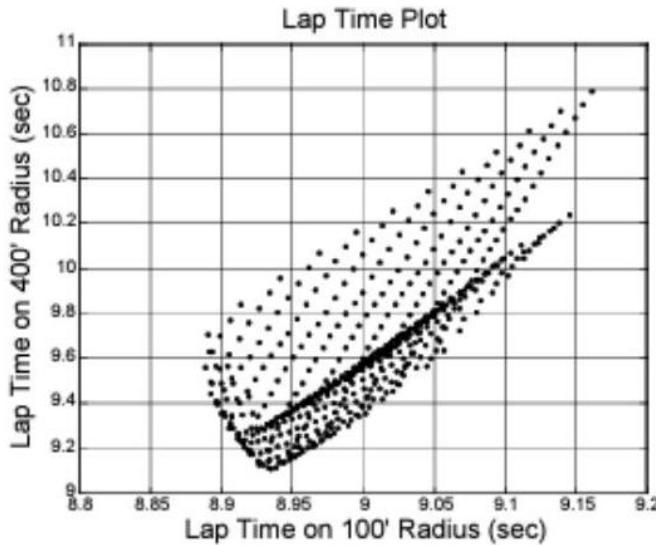
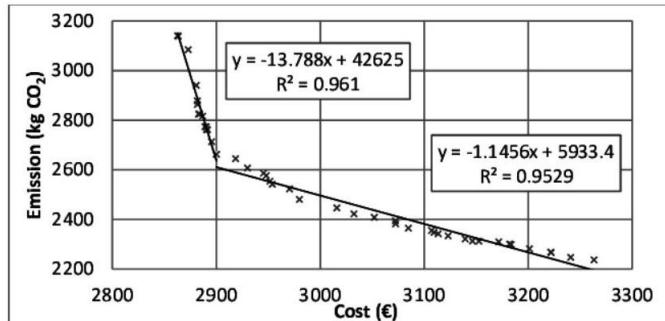


Figure 7: Grid Search Results in the Performance Space

10.8 Sustainable Constriction [2]



10.9 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

11. Graph Algorithms

Chapter 11. Graph Algorithms

10% complete. Goal 80% completion date: July 20

Notes: .

Resources

Youtube! Video of many graph algorithms by Google engineer (6+ hours)

Write this section.

11.1 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.

11.2 Graphs

11.2.1. Drawing Graphs

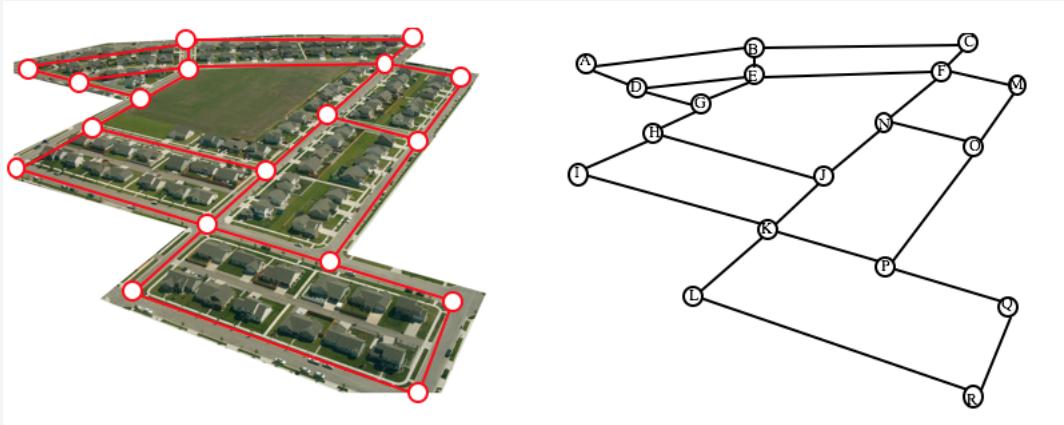
Example 11.1

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.



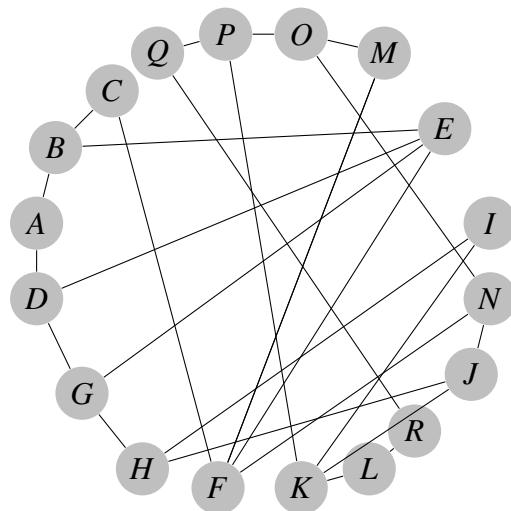
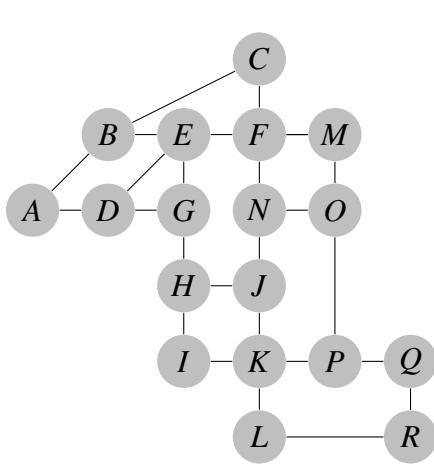
^aSame Beebe. <http://www.flickr.com/photos/sbeebe/2850476641/>

This type of simplified picture is called a **graph**.

Definition 11.2: 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.

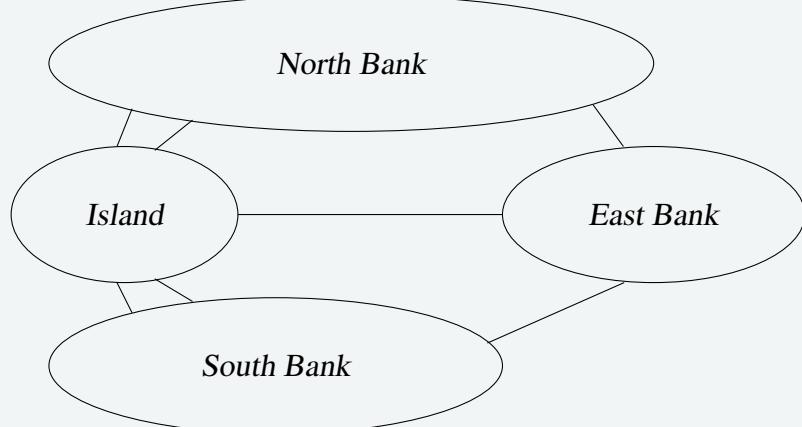
Example 11.3

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

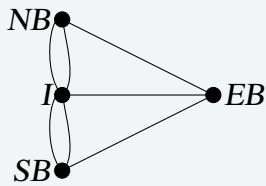
Picture

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.

11.3 Definitions

While we loosely defined some terminology earlier, we now will try to be more specific.

Definition 11.4: Vertex

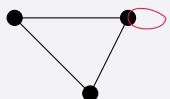
A vertex is a dot in the graph that could represent an intersection of streets, a land mass, or a general location, like "work?" or "school". Vertices are often connected by edges. Note that vertices only occur when a dot is explicitly placed, not whenever two edges cross. Imagine a freeway overpass – 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 11.5: 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.

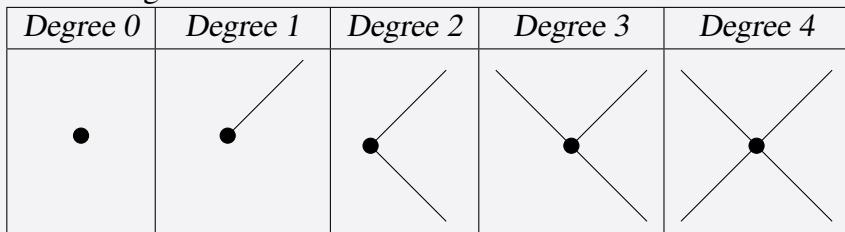
Definition 11.6: Loop

A loop is a special type of edge that connects a vertex to itself. Loops are not used much in street network graphs.

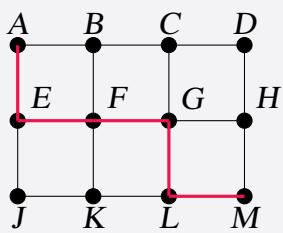


Definition 11.7: Degree of a vertex

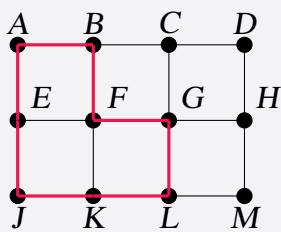
The degree of a vertex is the number of edges meeting at that vertex. It is possible for a vertex to have a degree of zero or larger.

**Definition 11.8: 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.

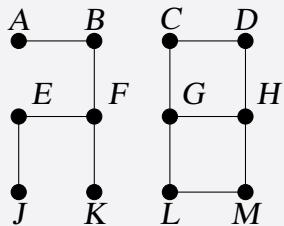
**Definition 11.9: Circuit (a.k.a. cycle)**

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 11.10: 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.

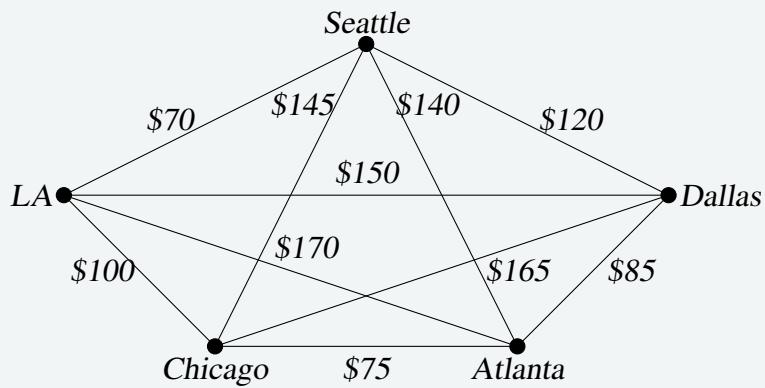
**Definition 11.11: Weights**

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 11.12

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?



11.4 Shortest Path

Outcomes

- *What is the problem statement?*
- *How to use Dijkstra's algorithm*
- *Software solutions*

Resources

- *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.

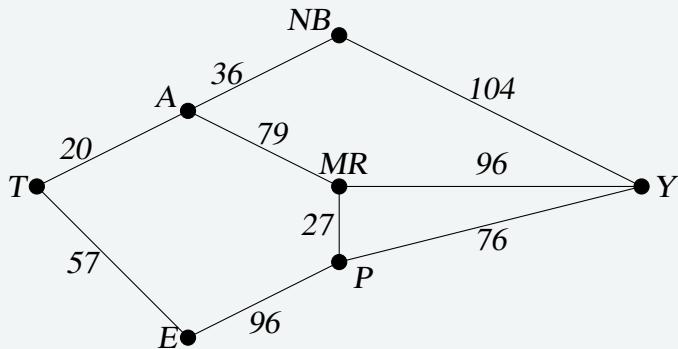
While often it is possible to find a shortest path on a small graph by guess-and-check, our goal in this chapter is to develop methods to solve complex problems in a systematic way by following **algorithms**. An algorithm is a step-by-step procedure for solving a problem. Dijkstra's (pronounced dike-strə) algorithm will find the shortest path between two vertices.

Dijkstra's Algorithm

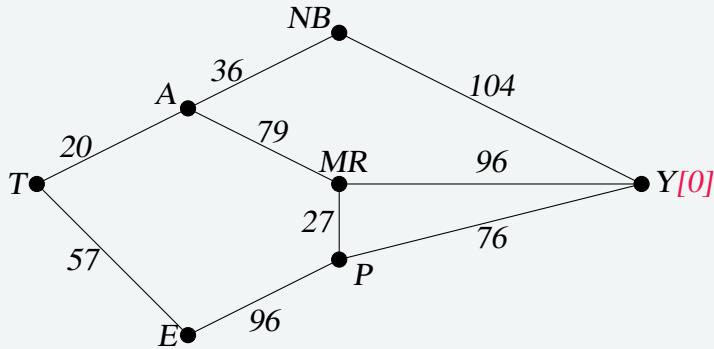
1. Mark the ending vertex with a distance of zero. Designate this vertex as current.
2. Find all vertices leading to the current vertex. Calculate their distances to the end. Since we already know the distance the current vertex is from the end, this will just require adding the most recent edge. Don't 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 11.13

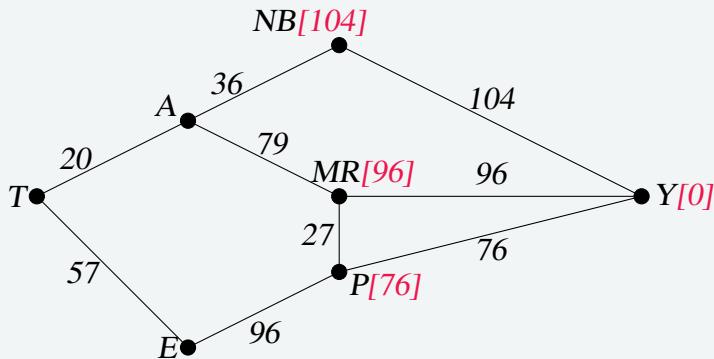
Suppose you need to travel from Yakima, WA (vertex Y) to Tacoma, WA (vertex T). Looking at a map, it looks like driving through Auburn (A) then Mount Rainier (MR) might be shortest, but it's not totally clear since that road is probably slower than taking the major highway through North Bend (NB). A 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.



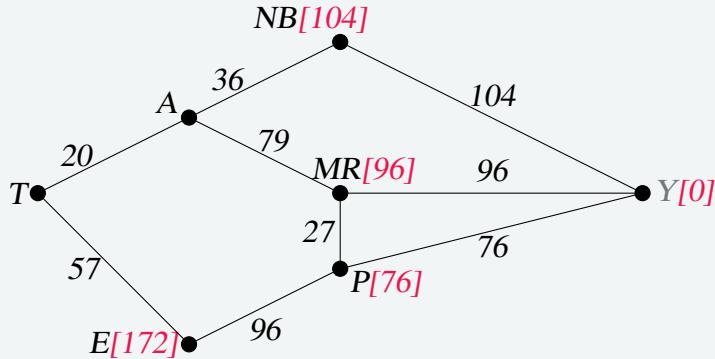
Step 2: For each vertex leading to Y, we calculate the distance to the end. For example, NB is a distance of 104 from the end, and MR is 96 from the end. Remember that distances in this case refer to the travel time in minutes.



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.

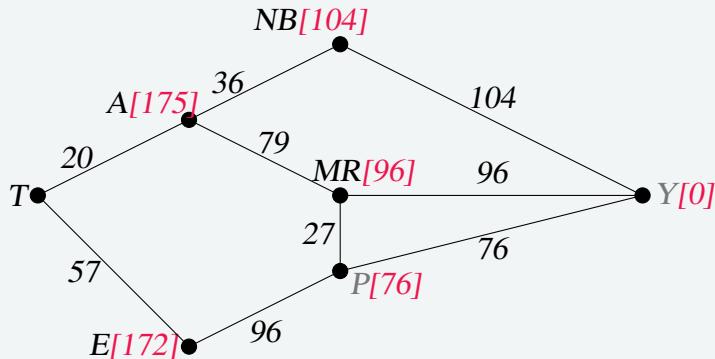
Step 2 (#2): For each vertex leading to P (and not leading to a visited vertex) we find the distance from the end. Since E is 96 minutes from P, and we've already calculated P is 76 minutes from Y, we can compute that E is $96 + 76 = 172$ minutes from Y.

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.



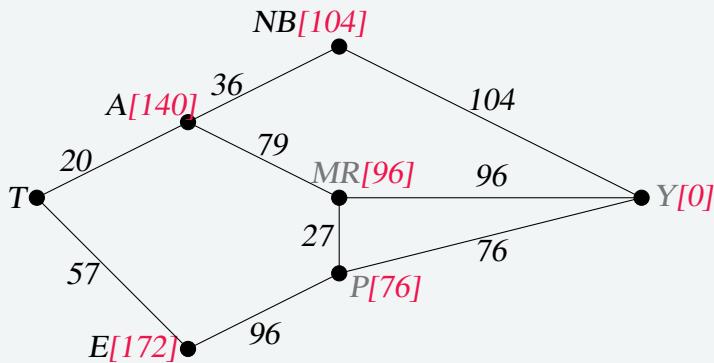
Step 3 & 4 (#2): We mark P as visited, and designate the vertex with the smallest recorded distance as current: MR. Back to step 2.

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.



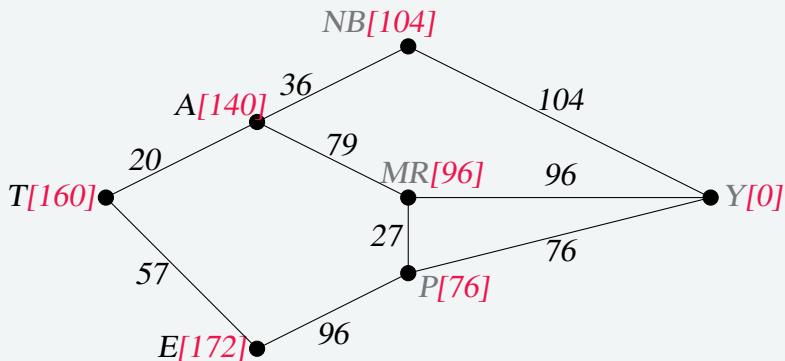
Step 3 & 4 (#3): We mark MR as visited, and designate the vertex with smallest recorded distance as current: NB. Back to step 2.

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.

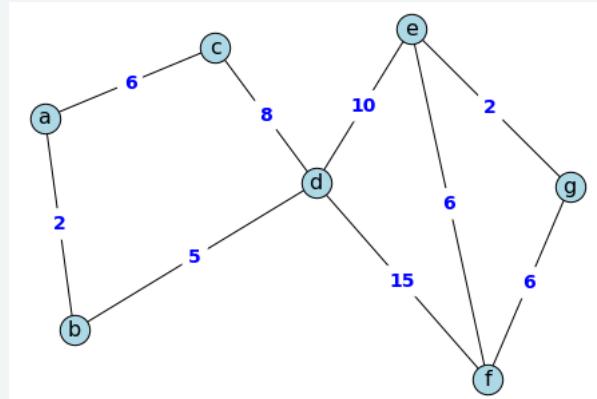
Dijkstra's algorithm is an **optimal algorithm**, meaning that it always produces the actual shortest path, not just a path that is pretty short, provided one exists. This algorithm is also **efficient**, meaning that it can be implemented in a reasonable amount of time. Dijkstra's algorithm takes around 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.

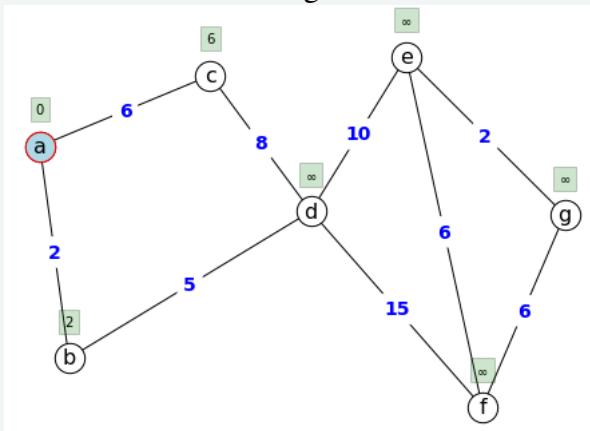
In contrast, an **inefficient** algorithm might try to list all possible paths then compute the length of each path. Trying to list all possible paths could easily take 10^{25} calculations to compute the shortest path with only 25 vertices; that's a 1 with 25 zeros after it! To put that in perspective, the fastest computer in the world would still spend over 1000 years analyzing all those paths.

Example 11.14: Dijkstra's algorithm example

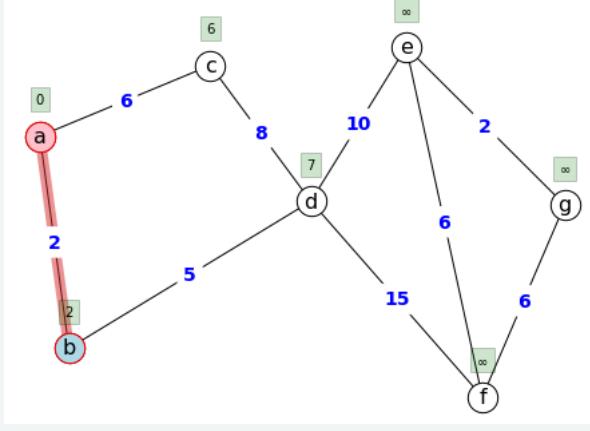
We would like to find a shortest path in the graph from node a to node g. See *Code for python code to solve this problem and create these graphics.*



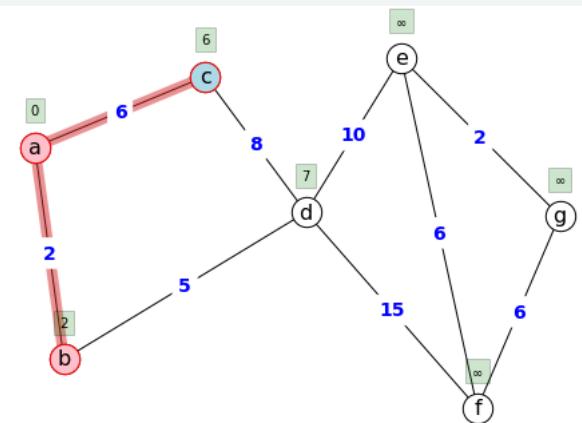
We will initialize our algorithm at node 'a'.



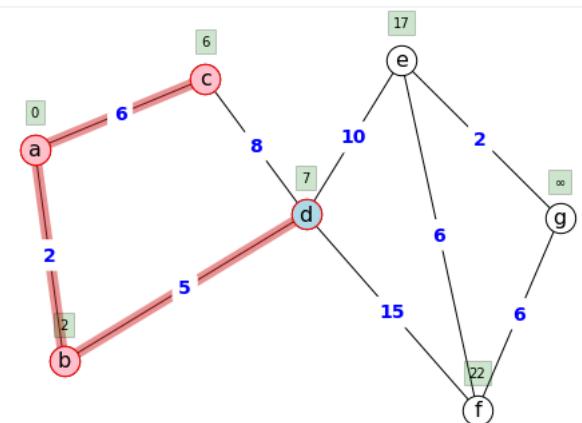
current	a	b	c	d	e	f	g
a	0	2	6	∞	∞	∞	∞



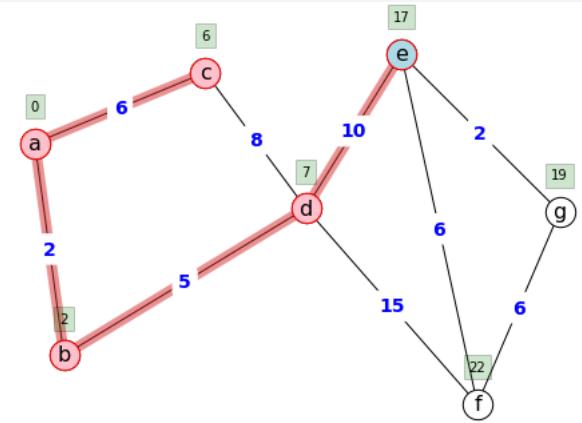
current	a	b	c	d	e	f	g
b	0	2	6	7	∞	∞	∞



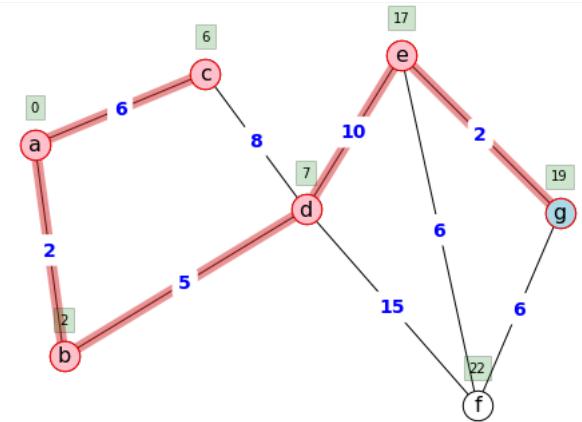
current	a	b	c	d	e	f	g
c	0	2	6	7	∞	∞	∞



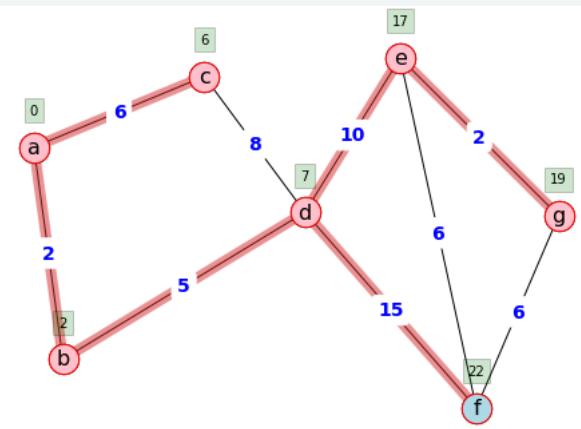
current	a	b	c	d	e	f	g
d	0	2	6	7	17	22	∞



current	a	b	c	d	e	f	g
e	0	2	6	7	17	22	19

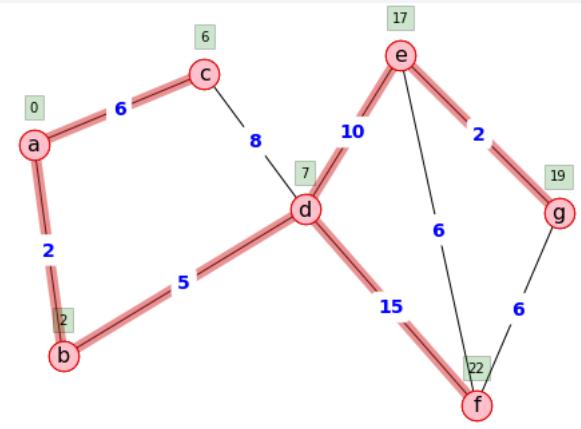


current	a	b	c	d	e	f	g
g	0	2	6	7	17	22	19



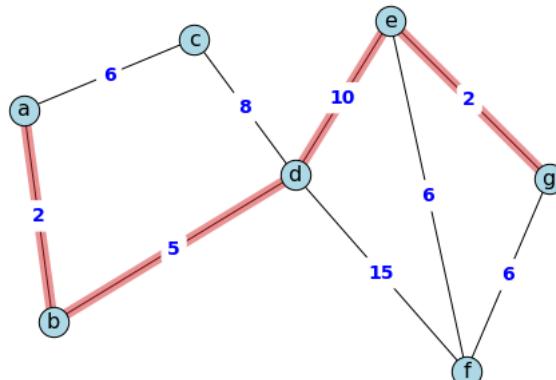
current	a	b	c	d	e	f	g
f	0	2	6	7	17	22	19

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 11.15

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			*

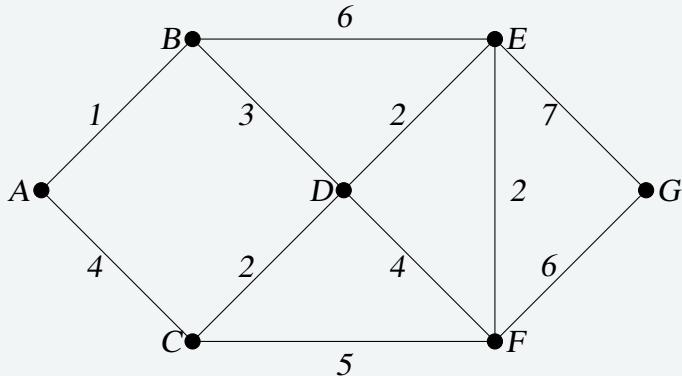
Step 2 (#5): The distance from Atlanta to Baltimore is 14. Adding that to the distance already calculated for Atlanta gives a total distance of $14 + 40 = 54$ hours from Baltimore to Bakersfield through Atlanta. Since this is larger than the currently calculated distance, we do not replace the distance 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 11.16

Find the shortest path between vertices A and G in the graph below.



11.5 Spanning Trees

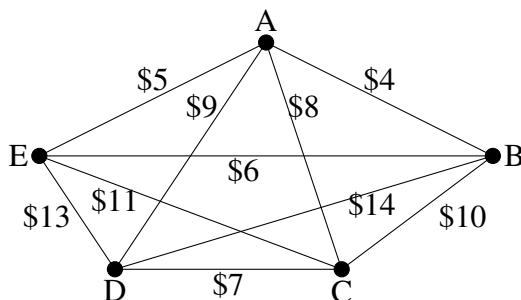
Outcomes

- 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.

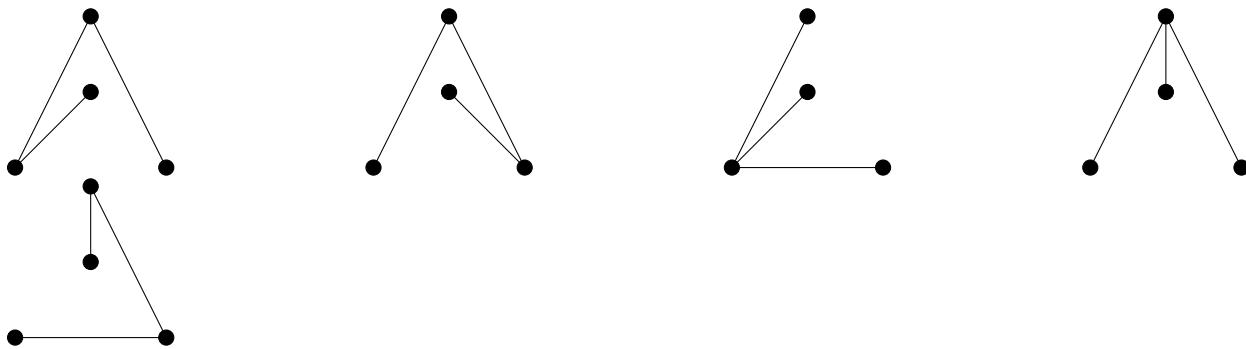


In this case, we don't need to find a circuit, or even a specific path; all we need to do is make sure we can make a call from any office to any other. In other words, we need to be sure there is a path from any vertex to any other vertex.

Definition 11.17: 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 11.18: 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.

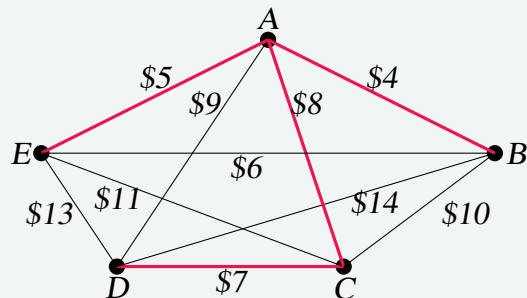
Kruskal's Algorithm

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 11.19

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 11.20

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	-

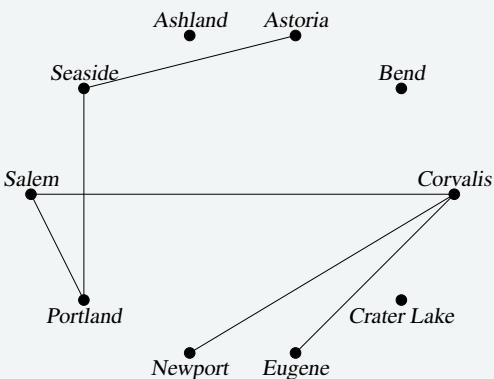
Using Kruskal's algorithm, we add edges from cheapest to most expensive, rejecting any that close a circuit. We stop when the graph is connected.

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

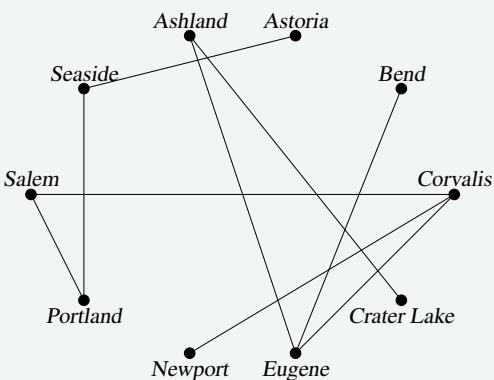
The graph up to this point is shown to the right.

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

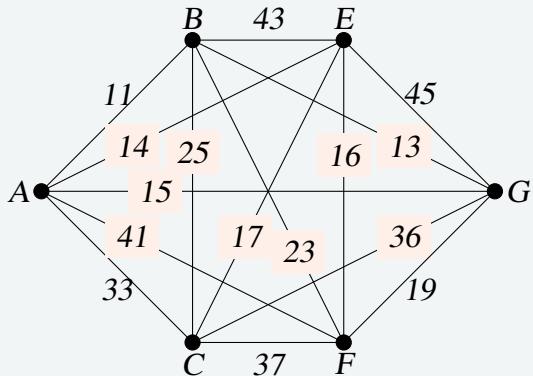


This connects the graph. The total length of cable to lay would be 695 miles.



Exercise 11.21: Min Cost Spanning Tree

Find a minimum cost spanning tree on the graph below using Kruskal's algorithm.

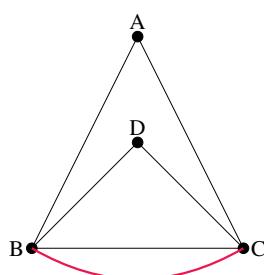


11.6 Exercise Answers

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: ABEGFCD(F)EDBCA

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)

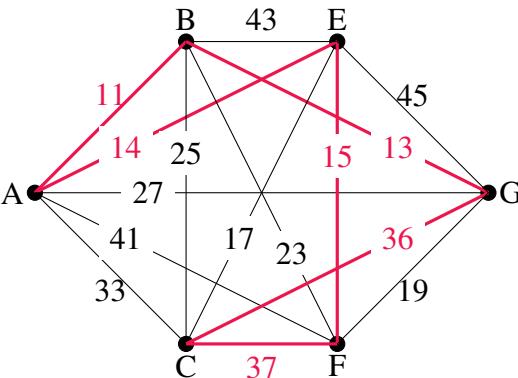
FG: Skip (would create a circuit not including C)

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

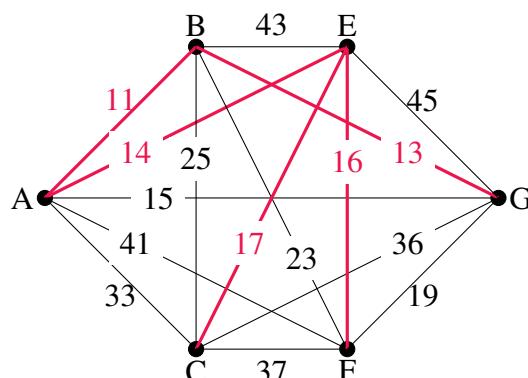
AE: Add, cost 14

AG: Skip, would create circuit ABGA

EF: Add, cost 16

EC: Add, cost 17

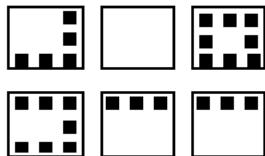
This completes the spanning tree.



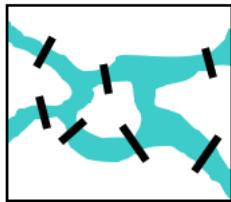
11.7 Additional Exercises

Skills

1. 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.

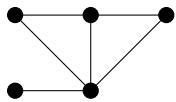
	Plano	Mesquite	Arlington	Denton
Fort Worth	54	52	19	42
Plano		38	53	41
Mesquite			43	56
Arlington				50

4. Shown in the table below are the one-way airfares between 5 cities². Create a graph showing this data.

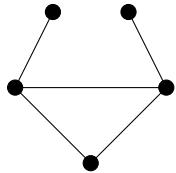
	Honolulu	London	Moscow	Cairo
Seattle	\$159	\$370	\$654	\$684
Honolulu		\$830	\$854	\$801
London			\$245	\$323
Moscow				\$329

5. Find the degree of each vertex in the graph below.

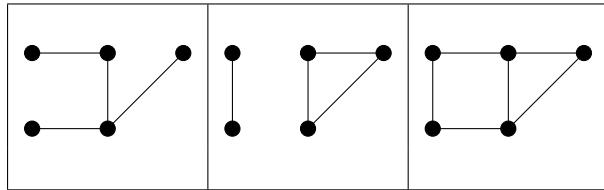
²Cheapest fares found when retrieved Sept. 1, 2009 for travel Sept. 22, 2009



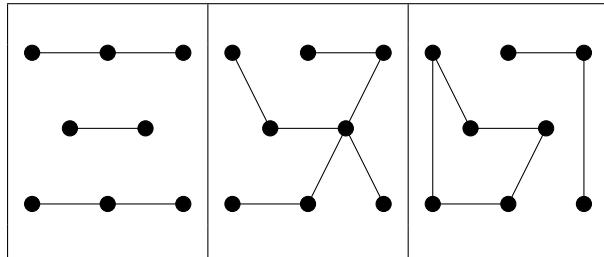
6. Find the degree of each vertex in the graph below.



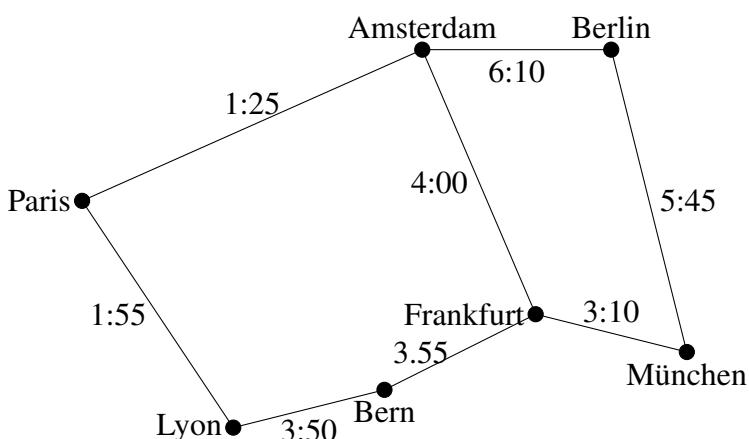
7. Which of these graphs are connected?



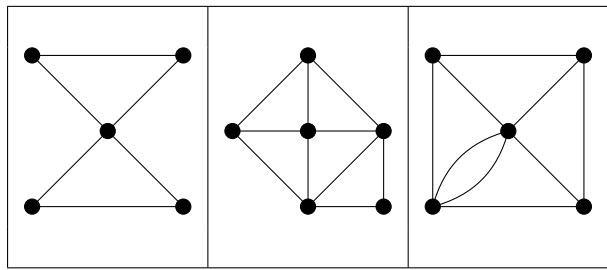
8. Which of these graphs are connected?



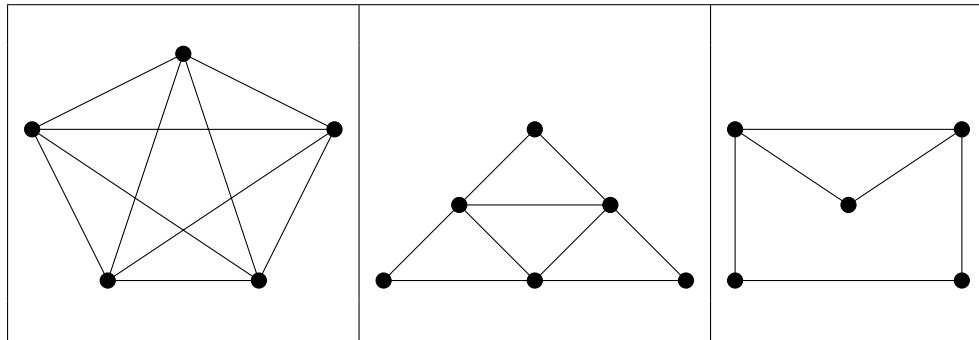
9. Travel times by rail for a segment of the Eurail system is shown below with travel times in hours and minutes. Find path with shortest travel time from Bern to Berlin by applying Dijkstra's algorithm.



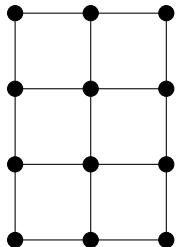
10. Using the graph from the previous problem, find the path with shortest travel time from Paris to Munchen.
11. Does each of these graphs have an Euler circuit? If so, find it.



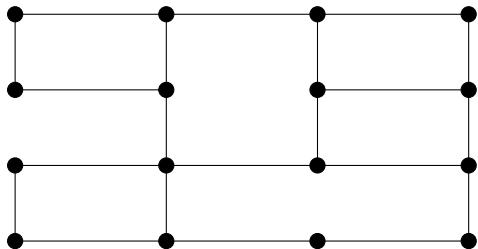
12. Does each of these graphs have an Euler circuit? If so, find it.



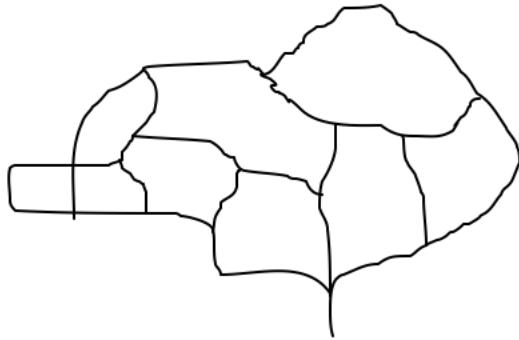
13. Eulerize this graph using as few edge duplications as possible. Then, find an Euler circuit.



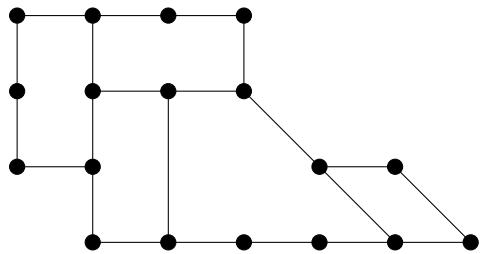
14. Eulerize this graph using as few edge duplications as possible. Then, find an Euler circuit.



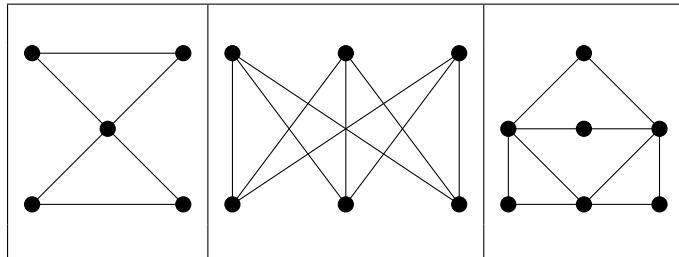
15. The maintenance staff at an amusement park need to patrol the major walkways, shown in the graph below, collecting litter. Find an efficient patrol route by finding an Euler circuit. If necessary, eulerize the graph in an efficient way.



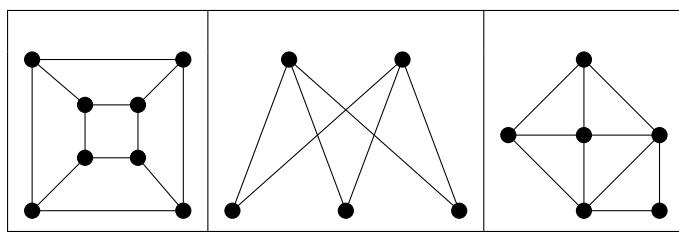
16. After a storm, the city crew inspects for trees or brush blocking the road. Find an efficient route for the neighborhood below by finding an Euler circuit. If necessary, eulerize the graph in an efficient way.



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.



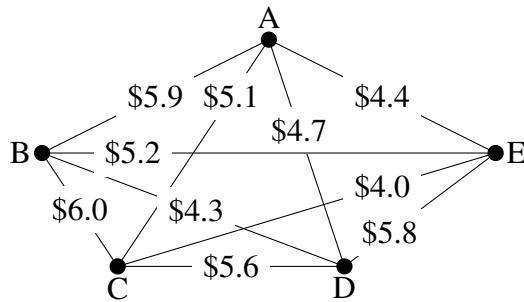
19. A company needs to deliver product to each of their 5 stores around the Dallas, TX area. Driving distances between the stores are shown below. Find a route for the driver to follow, returning to the distribution center in Fort Worth:

- (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:
- Using Nearest Neighbor starting in Seattle
 - Using Repeated Nearest Neighbor
 - 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:
- Using Nearest Neighbor starting at building A
 - Using Repeated Nearest Neighbor
 - 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:
- Using Nearest Neighbor starting in Jerusalem
 - Using Repeated Nearest Neighbor
 - 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

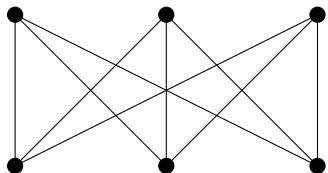
23. Find a minimum cost spanning tree for the graph you created in problem #3.

24. Find a minimum cost spanning tree for the graph you created in problem #22.

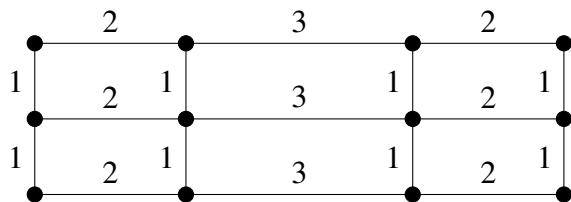
25. Find a minimum cost spanning tree for the graph from problem #21.

Concepts

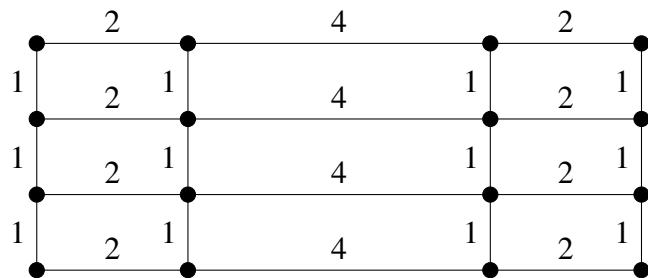
26. Can a graph have one vertex with odd degree? If not, are there other values that are not possible? Why?
27. A complete graph is one in which there is an edge connecting every vertex to every other vertex. For what values of n does complete graph with n vertices have an Euler circuit? A Hamiltonian 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?



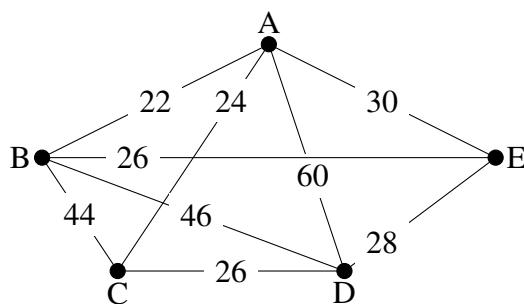
29. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



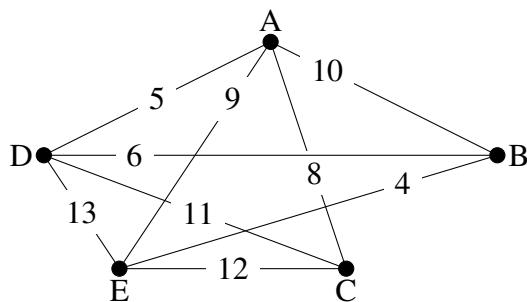
30. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



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) 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?

11.7.1. 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>

Part III

Integer Programming

Part II: Integer Programming

Notes: This Part applies to DORII. Ideally it will be ready for September 2022.

12. Integer Programming Formulations

Chapter 12. Integer Programming Formulations

70% complete. Goal 80% completion date: August 20

Notes:

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.

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](#)

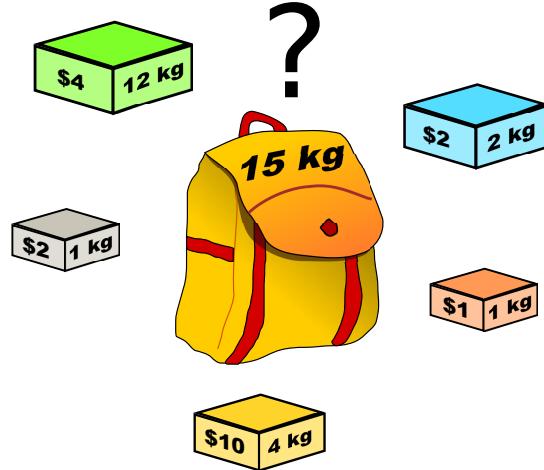
In this section, we will describe classical integer programming formulations. These formulations may reflect a real world problem exactly, or may be part of the setup of a real world problem.

12.1 Knapsack Problem

The *knapsack problem*¹ 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 12.1), but has applications in many contexts.

¹Video! - Michel Belaire (EPFL) teaching knapsack problem

²[wiki/File/knapsack](#), from [wiki/File/knapsack](#). [wiki/File/knapsack](#), [wiki/File/knapsack](#).



© wiki/File/knapsack²

Figure 12.1: Knapsack Problem: which items should we choose take in the knapsack that maximizes the value while respecting the 15kg weight limit?

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 $c \in \mathbb{Q}^n$,

$$\begin{aligned} & \max c^\top x \\ & \text{s.t. } a^\top x \leq b \\ & \quad x \in \{0, 1\}^n \end{aligned} \tag{12.1}$$

Example: Knapsack

Gurobipy

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 12.1.

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:

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

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 $c \in \mathbb{Q}^n$,

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & a^\top x \leq b \\ & x \in \mathbb{Z}_+^n \end{aligned} \tag{12.2}$$

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 $c \in \mathbb{Q}^n$,

$$\max \quad c^\top x \tag{12.3}$$

$$\text{s.t.} \quad a^\top x = b \tag{12.4}$$

$$x \in \mathbb{Z}_+^n \tag{12.5}$$

Example 12.1:

Using pennies, nickels, dimes, and quarters, how can you minimize the number of coins you need 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

$$\begin{array}{ll} \min & p + n + d + q \\ \text{s.t.} & p + 5n + 10d + 25q = 83 \\ & p, d, n, q \in \mathbb{Z}_+ \end{array}$$

total number of coins used
sums to 83¢
each is a non-negative integer

12.2 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:

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

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:

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

12.3 Set Covering

Resources

Video! - Michel Belaire (EPFL) explaining set covering problem

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

$$\begin{aligned} & \min 1^\top x \\ \text{s.t. } & \sum_{v \in V_i} x_v \geq 1 \text{ for all } i = 1, \dots, l \\ & x_v \in \{0, 1\} \text{ for all } v \in V \end{aligned} \tag{12.1}$$

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 .

Resources

See AIMMS - Media Selection for an example of set covering applied to media selection.

Add flight crew scheduling example and images.

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:

$$\begin{aligned} & \min 1^\top x \\ \text{s.t. } & x_u + x_v \geq 1 \text{ for all } (u, v) \in E \\ & x_v \in \{0, 1\} \text{ for all } v \in V. \end{aligned} \tag{12.2}$$

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 12.2 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:

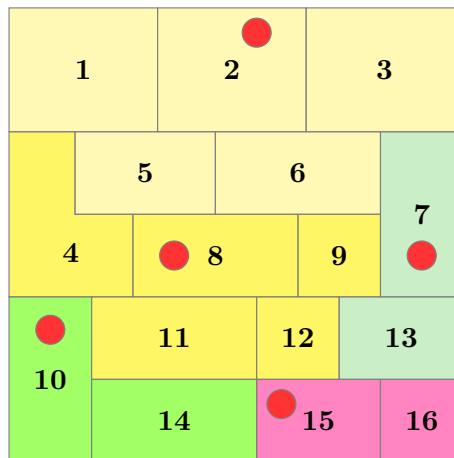
- Let V be the set of districts ($V = \{1, \dots, 16\}$)
- Let V_i be the set of districts that neighbor district i (e.g. $V_1 = \{2, 4, 5\}$).

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.pdf³

Figure 12.2: 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

³tikz/Illustration1.pdf, from tikz/Illustration1.pdf. tikz/Illustration1.pdf, tikz/Illustration1.pdf.

⁴tikz/Illustration2.pdf, from tikz/Illustration2.pdf. tikz/Illustration2.pdf, tikz/Illustration2.pdf.

⁵tikz/Illustration3.pdf, from tikz/Illustration3.pdf. tikz/Illustration3.pdf, tikz/Illustration3.pdf.

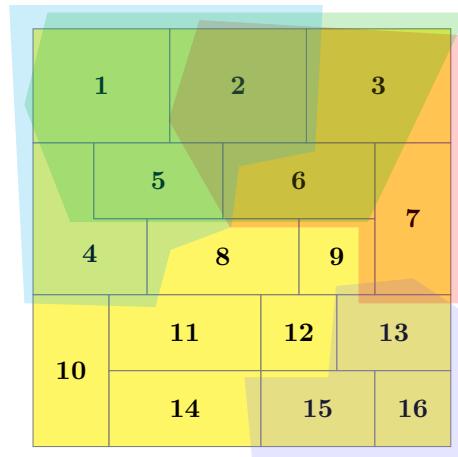
© tikz/Illustration2.pdf⁴

Figure 12.3: Set cover representation of fire station problem. For example, choosing district 16 to have a fire station covers districts 13, 15, and 16.

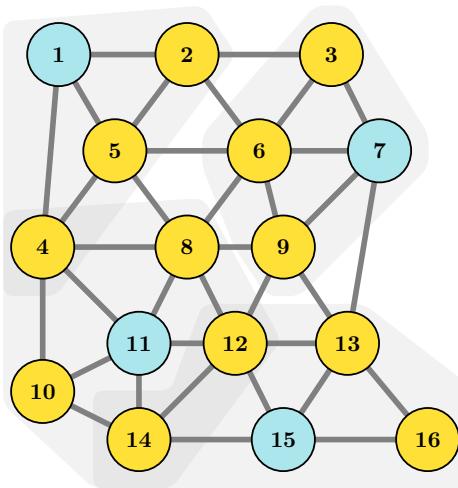
© tikz/Illustration3.pdf⁵

Figure 12.4: Graph representation of fire station problem. Every node is connected to a chosen node by an edge

set cover problem is

$$\begin{aligned}
 & \max c^\top x \\
 & \text{s.t. } Ax \geq 1 \\
 & \quad x \in \{0, 1\}^n.
 \end{aligned} \tag{12.3}$$

Example: Vertex Cover with matrix

An alternate way to solve ?? 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.

12.3.1. 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

$$\begin{aligned} & \max \quad c^\top x \\ & \text{s.t..} \quad Ax \geq b \\ & \quad x \in \mathbb{Z}_+^n. \end{aligned} \tag{12.4}$$

12.4 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} .

Include picture and example data

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:

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

12.5 Facility Location

Resources

- Wikipedia - Facility Location Problem
- See GUROBI Modeling Examples - 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*.

12.5.1. 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

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

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

Model:

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

12.5.2. 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

$$\begin{aligned} \min \quad & \sum_{i=1}^n \sum_{j=1}^m c_{ij} z_{ij} + \sum_{i=1}^n f_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n z_{ij} = 1 \text{ for all } j = 1, \dots, m \\ & \sum_{j=1}^m z_{ij} \leq M x_i \text{ for all } i = 1, \dots, n \\ & z_{ij} \in \{0, 1\} \text{ for all } i = 1, \dots, n \text{ and } j = 1, \dots, m \\ & x_i \in \{0, 1\} \text{ for all } i = 1, \dots, n \end{aligned} \tag{12.1}$$

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

12.6 Basic Modeling Tricks - Using Binary Variables

Resources

- JuMP tips and tricks
- Mosek Modeling Cookbook

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.

$$x_i \leq x_j \tag{12.1}$$

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

$$x_1 + x_2 + \dots + x_5 \leq 5 \cdot x_6 \tag{12.2}$$

Alternatively!

$$x_i \leq x_6 \quad \text{for all } i = 1, \dots, 5 \tag{12.3}$$

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

$$x_i \leq x_j \quad (12.4)$$

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

$$x_1 + x_2 + \dots + x_i \leq i \cdot x_j \quad (12.5)$$

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

$$x_i \leq x_j \quad (12.6)$$

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

$$x_i + x_j = 1 \quad (12.7)$$

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

$$x_i + x_j \geq 1 \quad (12.8)$$

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

$$x_j \leq (1 - x_i) \quad (12.9)$$

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

$$x_i + x_j + x_k \leq 1 \quad (12.10)$$

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

$$x_i + x_j + x_k \leq 2 \quad (12.11)$$

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

$$x_i + x_j + x_k = 1 \quad (12.12)$$

These tricks can be connected to create different function values.

Example 12.2: 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 12.3: 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 $i < j$.

12.6.1. Connecting to continuous variables

Let $x_i \geq 0$ and $y_i \in \{0, 1\}$ for all $i = 1, \dots, n$.

1. If $x_i > 0$, then $y_i = 1$.

$$x_i \leq M y_i \tag{12.13}$$

where M is a sufficiently large upper bound on the variable x_i .

2. If $x_i = 0$, then $y_i = 0$.

This is harder to model! Alternatively, we try modeling "if x_i is sufficiently small, then $y_i = 0$. For instance, if $x_i \leq 0.0000001$, then $y_i = 0$. This can be modeled as

$$x_i - 0.0000001 \geq y_i - 1. \tag{12.14}$$

3. If $y_i = 1$, then $x_i \geq 5$

$$5y_i \leq x_i. \tag{12.15}$$

12.6.2. 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.

12.6.2.1. 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:

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

12.6.2.2. 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:

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

12.7 Network Flow

Fix up this section



12.7.1. Example - Multicommodity Flow

https://en.wikipedia.org/wiki/Multi-commodity_flow_problem The **multi-commodity flow problem** is a network flow problem with multiple commodities (flow demands) between different source and sink nodes.

PROBLEM DEFINITION Given a flow network $G(V, E)$, where edge $(u, v) \in E$ has capacity $c(u, v)$. There are k commodities K_1, K_2, \dots, K_k , defined by $K_i = (s_i, t_i, d_i)$, where s_i and t_i is the **source** and **sink** of commodity i , and d_i is its demand. The variable $f_i(u, v)$ defines the fraction of flow i along edge (u, v) , where $f_i(u, v) \in [0, 1]$ in case the flow can be split among multiple paths, and $f_i(u, v) \in \{0, 1\}$ otherwise (i.e. "single path routing"). Find an assignment of all flow variables which satisfies the following four constraints:

(1) Link capacity: The sum of all flows routed over a link does not exceed its capacity.

$$\forall (u, v) \in E : \sum_{i=1}^k f_i(u, v) \cdot d_i \leq c(u, v)$$

(2) Flow conservation on transit nodes: The amount of a flow entering an intermediate node u is the same that exits the node.

$$\sum_{w \in V} f_i(u, w) - \sum_{w \in V} f_i(w, u) = 0 \quad \text{when } u \neq s_i, t_i$$

(3) Flow conservation at the source: A flow must exit its source node completely.

$$\sum_{w \in V} f_i(s_i, w) - \sum_{w \in V} f_i(w, s_i) = 1$$

(4) Flow conservation at the destination: A flow must enter its sink node completely.

$$\sum_{w \in V} f_i(w, t_i) - \sum_{w \in V} f_i(t_i, w) = 1$$

12.7.2. Corresponding optimization problems

Load balancing is the attempt to route flows such that the utilization $U(u, v)$ of all links $(u, v) \in E$ is even, where

$$U(u, v) = \frac{\sum_{i=1}^k f_i(u, v) \cdot d_i}{c(u, v)}$$

The problem can be solved e.g. by minimizing $\sum_{u, v \in V} (U(u, v))^2$. A common linearization of this problem is the minimization of the maximum utilization U_{max} , where

$$\forall (u, v) \in E : U_{max} \geq U(u, v)$$

In the **minimum cost multi-commodity flow problem**, there is a cost $a(u, v) \cdot f(u, v)$ for sending a flow on (u, v) . You then need to minimize

$$\sum_{(u, v) \in E} \left(a(u, v) \sum_{i=1}^k f_i(u, v) \right)$$

In the **maximum multi-commodity flow problem**, the demand of each commodity is not fixed, and the total throughput is maximized by maximizing the sum of all demands $\sum_{i=1}^k d_i$

12.7.3. Relation to other problems

The minimum cost variant of the multi-commodity flow problem is a generalization of the minimum cost flow problem (in which there is merely one source s and one sink t). Variants of the circulation problem are generalizations of all flow problems. That is, any flow problem can be viewed as a particular circulation problem.⁶

6

12.7.4. Usage

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

12.8 Transportation Problem

Add discussion of transportation problem and picture.

[Youtube! - TRANSPORTATION PROBLEM with PuLP in PYTHON](#)

[Notebook: Solution with Pyomo](#)

12.9 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

12.10 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

12.10.1. Further Topics

- Precedence Constraints

12.11 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. Algorithms and Complexity

Chapter 13. Algorithms and Complexity

60% complete. Goal 80% completion date: August 20

Notes:

Outcomes

- (a) *Describe asymptotic growth of functions using Big-O notation.*
- (b) *Analyze algorithms for the asymptotic runtime.*
- (c) *Classify problem types with respect to notions of how difficult they are to solve.*

Resources

- *MIT Lecture Notes - Big O*
- *Youtube! - P versus NP*

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).

13.1 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 13.1: 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 know what happens for large values of n .

Notice that

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0. \quad (13.1)$$

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 13.2: 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

$$\limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty. \quad (13.2)$$

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 13.3: 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

$$0 \leq f(n) \leq c g(n) \quad \text{for all } n \geq n_0. \quad (13.3)$$

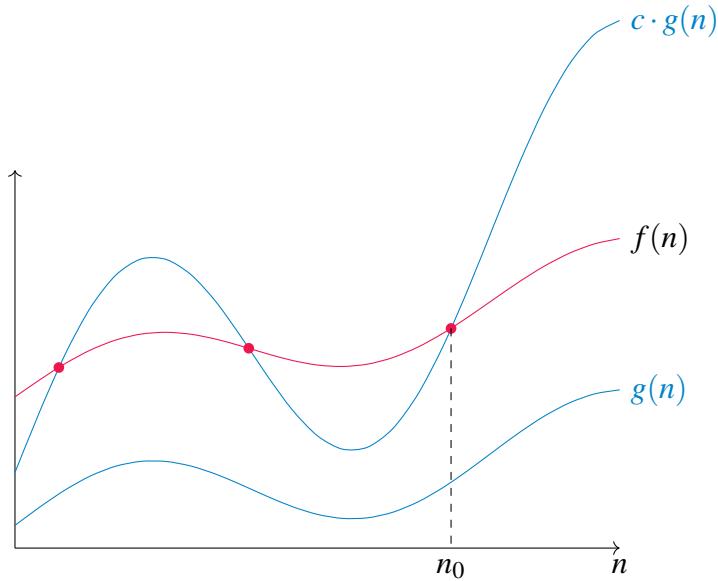


Figure 13.1: Example of Big-O notation: $f(n) = O(g(n))$. We see that for all $n \geq n_0$, we have $c \cdot g(n) \geq f(n)$.

Example 13.4:

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 13.3 is satisfied.

Note first that we always have

$$(a) n^2 \leq n^2 \text{ and therefore } 5n^2 \leq 5n^2$$

Note that if $n \geq 1$, then

$$(a) n \leq n^2 \text{ and therefore } 10n \leq 10n^2$$

$$(b) 1 \leq n^2 \text{ and therefore } 7 \leq 7n^2$$

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

$$5n^2 + 10n + 7 \leq 5n^2 + 10n^2 + 7n^2 \quad \text{for all } n \geq 1, \quad (13.4)$$

$$\Rightarrow 5n^2 + 10n + 7 \leq 22n^2 \quad \text{for all } n \geq 1. \quad (13.5)$$

Hence, we have shown that Equation 13.3 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 $n^2 + n \leq 2n^3$ 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 $f(n) = n^2 + n$, $g(n) = 2n^2$, $h(n) = 2n^3$.

Then we have

$n = 10$.	$n = 100$.	$n = 1000$.	$n = .$ 10000
$f(n)$	110,	10100,	1001000,
$g(n)$	200,	20000,	2000000,
$h(n)$.	2000,	2000000,	200000000000

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 13.2: time-of-algorithms

13.2 Algorithms - Example with Bubble Sort

The following definition comes from Merriam-Webster's dictionary.

Definition 13.5: 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.

13.2.1. Sorting

Resources

- [Wikipedia](#)

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 13.6: Sorting Problem

*Suppose you have the list of number $(10, 35, 9, 4, 15, 22)$.
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:

- (a) Compare numbers in position 1 and 2. If numbers are out of order, then swap them.
 - (b) Next, compare numbers in position 2 and 3. If numbers are out of order, then swap them.
 - (c) 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!
- (d) If no swaps had to be made, then the whole list is sorted!
 - (e) 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:

$$(10, \mathbf{9}, 4, 15, 22 | 35) \rightarrow (\mathbf{9}, 10, 4, 15, 22 | 35)$$

Step 2:

$$(9, \mathbf{10}, 4, 15, 22 | 35) \rightarrow (9, \mathbf{4}, \mathbf{10}, 15, 22 | 35)$$

Step 3:

$$(9, 4, \mathbf{10}, \mathbf{15}, 22 | 35) \rightarrow (9, 4, \mathbf{10}, \mathbf{15}, 22 | 35)$$

Step 4:

$$(9, 4, 10, \mathbf{15}, \mathbf{22} | 35) \rightarrow (9, 4, 10, \mathbf{15}, \mathbf{22} | 35)$$

Now 22 is in the correct spot!

Third pass through the list

Step 1:

$$(\mathbf{9}, 4, 10, 15, | 22, 35) \rightarrow (4, \mathbf{9}, 10, 15, | 22, 35)$$

Step 2:

$$(4, \mathbf{9}, \mathbf{10}, 15, | 22, 35) \rightarrow (4, \mathbf{9}, \mathbf{10}, 15, | 22, 35)$$

Step 3:

$$(4, 9, \mathbf{10}, \mathbf{15}, | 22, 35) \rightarrow (4, 9, \mathbf{10}, \mathbf{15}, | 22, 35)$$

Fourth pass through the list

Step 1:

$$(4, \mathbf{9}, 10, | 15, 22, 35) \rightarrow (4, \mathbf{9}, 10, | 15, 22, 35)$$

Step 2:

$$(4, \mathbf{9}, \mathbf{10}, | 15, 22, 35) \rightarrow (4, \mathbf{9}, \mathbf{10}, | 15, 22, 35)$$

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 thus 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

$$n\left(\frac{n}{2} - 1\right) + \frac{n}{2} = \frac{n(n - 1)}{2}.$$

Hence, we have proved that

$$\sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2}.$$

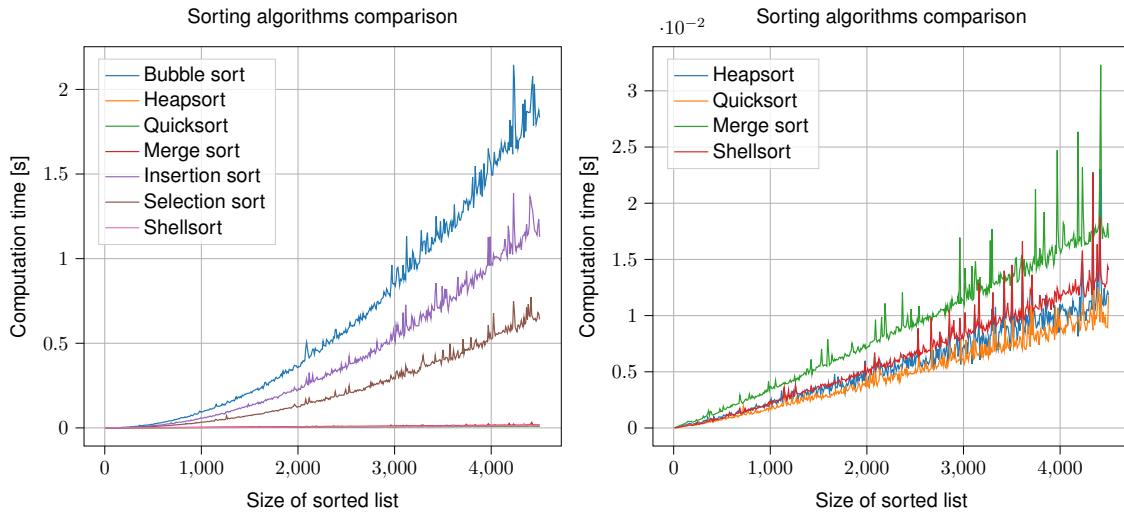
Since we just care about the Big-O expression, we can upper bound this by $O(n^2)$.

Hence, we will say that

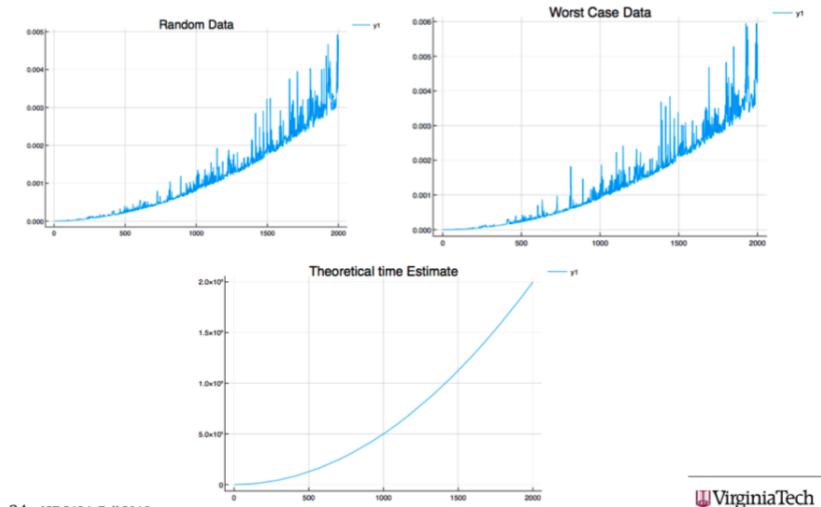
These can be verified experimentally as seen in the following plot. The random case grows quadratically just as the worst case does.

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 13.3: Comparison of runtimes of sorting algorithms.**

Time elapsed in computer for bubble sort

**Figure 13.4: bubble-sort-computational-example**

Theorem 13.7: 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

$$\bullet \sum_{i=1}^n i^k = O(n^{k+1}).$$

13.3 Problem, instance, size

13.3.1. Problem, instance

Definition 13.8: 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 13.9: 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).

13.3.2. 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.

13.3.3. 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 13.10: 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

$\# \text{ of bits to encode the problem}$

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 13.11: 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.

13.4 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 13.5 .

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.

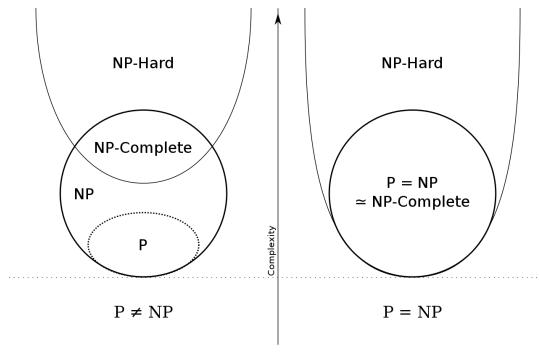


Figure 13.5: Complexity class possibilities. Most academics agree that the case $P \neq NP$ is more likely.

13.4.1. P

Definition 13.12: 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 13.13: Sorting

The sorting problem can be solved in $O(n^2)$ time. Thus, this problem is in *P*

Example 13.14: 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 13.15: 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.

13.4.2. 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 exist a solution with objective value greater than some number?"

Example 13.16: 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 exist a matching with objective value greater than W ?

Output: Either "yes" or "no".

We can now define the class of NP.

Definition 13.17: 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:

- Describe a format for a certificate to the problem.
- Show that given such a certificate, it is easy to verify the solution to the problem.

Example 13.18

Integer Linear Programming is in NP. More explicitly, the feasibility question of
"Does there exist 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

- (a) Check if x is integer.
- (b) Use matrix multiplication to check if $Ax \leq b$ holds.

13.4.3. Problem Reductions

We can compare different types of problems by showing that we can use one to solve the other.

A simple example of this is the problem *Integer Programming* and the *Matching Problem*.

Since we can model the *Matching Problem* as an *Integer Program*, then we know that we can solve the *Matching Problem* provided that we can solve *Integer Programs*.

$$\text{Matching Problem} \leq \text{Integer Programming}.$$

Definition 13.19: 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.

13.4.4. 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.

13.4.5. 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

- (a) showing that it can be used to solve binary integer programming feasibility,
- (b) showing that the problem is in NP.

The first problem proven to be NP-Complete is called *3-SAT* []. 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 there opposite of the value of X_1 .

These can be described in the following table

$$\text{true} \wedge \text{true} = \text{true} \tag{13.1}$$

$$\text{true} \wedge \text{false} = \text{false} \tag{13.2}$$

$$\text{false} \wedge \text{false} = \text{false} \tag{13.3}$$

$$\text{false} \wedge \text{true} = \text{false} \tag{13.4}$$

$$\text{true} \vee \text{true} = \text{true} \tag{13.5}$$

$$\text{true} \vee \text{false} = \text{true} \tag{13.6}$$

$$\text{false} \vee \text{false} = \text{false} \tag{13.7}$$

$$\text{false} \vee \text{true} = \text{true} \tag{13.8}$$

$$\neg \text{true} = \text{false} \tag{13.9}$$

$$\neg \text{false} = \text{true} \tag{13.10}$$

For example, **Missing code here** A *logical expression* is a sequence of logical operations on variables X_1, \dots, X_n , such that

$$(X_1 \wedge \neg X_2 \vee X_3) \wedge (X_1 \vee \neg X_3) \vee (X_1 \wedge X_2 \wedge X_3). \tag{13.11}$$

A *clause* is a logical expression that only contains the operations \vee and \neg and is not nested (with parentheses), such as

$$X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4. \tag{13.12}$$

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

$$(X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4) \wedge (X_1 \vee X_2 \vee X_3) \wedge (X_2 \vee \neg X_3 \vee \neg X_4 \vee X_5). \quad (13.13)$$

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.

13.5 Problems and Algorithms

We will discuss the following concepts:

- Feasible solutions
- Optimal solutions
- Approximate solutions
- Heuristics
- Exact Algorithms
- Approximation Algorithms
- Complexity class relations

13.5.1. Matching Problem

Definition 13.20: 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' .

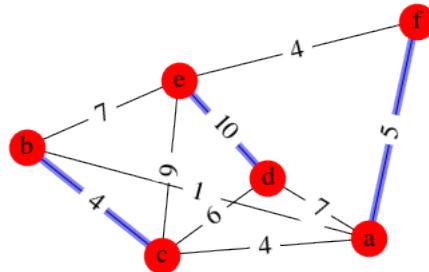
A maximal matching is a matching E' such that there is no matching E'' that strictly contains it.

INCLUDE PICTURES OF MATCHINGS

Figure 13.6: Two possible matchings. On the left, we have a perfect matchings (all nodes are matched). On the right, a feasible matching, but not a perfect matching since not all nodes are matched.

Definition 13.21: 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 13.7: graph-for-matching-maximal

⁴graph-for-matching-maximal, from graph-for-matching-maximal.

graph-for-matching-maximal,

13.5.1.1. 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: $O(|E| \log(|V|))$

- (a) Begin with an empty graph ($M = \emptyset$)
- (b) Label the edges in the graph such that $w(e_1) \geq w(e_2) \geq \dots \geq w(e_m)$
- (c) 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)
- (d) Return M

Theorem 13.22: Greedy algorithm gives a 2-approximation [[Avis83]]

The greedy algorithm finds a 2-approximation of the maximum weighted matching problem. That is, $w(M_{\text{greedy}}) \geq \frac{1}{2}w(M^*)$.

13.5.1.2. Other algorithms to look at

- (a) Improved Algorithm [DRAKE2003211]
- (b) Blossom Algorithm Wikipedia

13.5.2. Minimum Spanning Tree

Definition 13.23: Spanning Tree

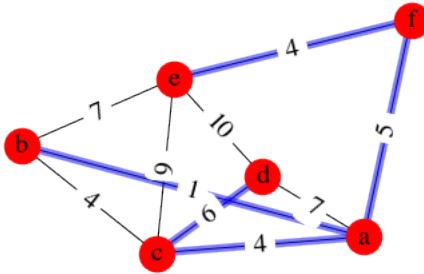
Given a graph $G = (V, E)$, a spanning tree connected, acyclic subgraph T that contains every node in V .

© spanning-tree⁵

Figure 13.8: spanning-tree

Definition 13.24: 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 13.9: spanning-tree-MST

Lemma 13.25: Edges and Spanning Trees

Let G be a connected graph with n vertices.

- (a) T is a spanning tree of G if and only if T has $n - 1$ edges and is connected.
- (b) Any subgraph S of G with more than $n - 1$ edges contains a cycle.

See Section 15.2 for integer programming formulations of this problem.

13.5.3. Kruskal's algorithm

Resources

[Wikipedia](#)

Kruskal - for Minimum Spanning tree:

Complexity: $O(|E| \log(|V|))$

- (a) Begin with an empty tree ($T = \emptyset$)
- (b) Label the edges in the graph such that $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$
- (c) For $i = 1, \dots, m$
If $T \cup \{e_i\}$ is acyclic, then set $T \leftarrow T \cup \{e_i\}$
- (d) Return T

⁵spanning-tree, from [spanning-tree](#). [spanning-tree](#), [spanning-tree](#).

⁶spanning-tree-MST, from [spanning-tree-MST](#). [spanning-tree-MST](#), [spanning-tree-MST](#).

13.5.3.1. Prim's Algorithm

Resources

- [Wikipedia](#)
- [TeXample - Figure for min spanning tree](#)

13.5.4. Traveling Salesman Problem

Resources

-

See Section 15.3 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 ??.

13.5.4.1. Nearest Neighbor - Construction Heuristic

Resources

- [Wikipedia](#)

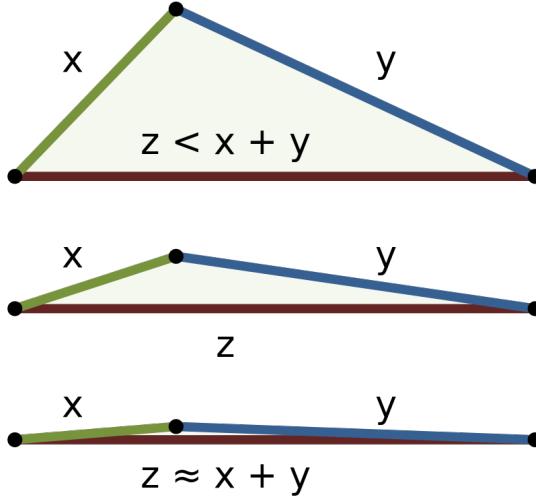
We will discuss heuristics more later in this book. For now, present this construction heuristic as a simple algorithmic example.

Starting from any node, add the edge to the next closest node. Continue this process.

Nearest Neighbor:

Complexity: $O(n^2)$

- (a) Start from any node (lets call this node 1) and label this as your current node.
- (b) Pick the next current node as the one that is closest to the current node that has not yet been visited.
- (c) Repeat step 2 until all nodes are in the tour.



© wiki/File/triangle_inequality.png⁷

Figure 13.10: wiki/File/triangle_inequality.png

13.5.4.2. 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 13.10

Definition 13.26: Triangle Inequality on a Graph

A complete, weighted graph G (i.e., a graph that has all possible edges and a weight assigned to each edge) satisfies the triangle inequality provided that for every triple of vertices a, b, c and edges e_{ab}, e_{bc}, e_{ac} , we have that

$$w(e_{ab}) + w(e_{bc}) \geq w(e_{ac}).$$

Let S be the resulting tour and let S^* be an optimal tour. Since the resulting tour is feasible, it will satisfy

$$w(S^*) \leq w(S).$$

But we also know that the weight of a minimum spanning tree T is less than that of the optimal tour, hence

$$w(T) \leq w(S^*).$$

Lastly, due to the triangle inequality we know that

$$w(S) \leq 2w(T),$$

⁷wiki/File/triangle_inequality.png, from wiki/File/triangle_inequality.png. wiki/File/triangle_inequality.png, wiki/File/triangle_inequality.png.

Algorithm 5 Double Spanning Tree

Require:A graph $G = (V, E)$ that satisfies the triangle inequality

Ensure:A tour that is a 2-Apx of the optimal solution

- 1: Compute a minimum spanning tree T of G .
 - 2: Double each edge in the minimum spanning tree (i.e., if edge e_{ab} is in T , add edge e_{ba} .
 - 3: Compute an Eulerian Tour using these edges.
 - 4: Return tour that visits vertices in the order the Eulerian Tour visits them, but without repeating any vertices.
-

since replacing any edge in the Eulerian tour with a more direct edge only reduces the total weight.

Putting this together, we have

$$w(S) \leq 2w(T) \leq 2w(S^*)$$

and hence, S is a 2-approximation of the optimal solution.

13.5.4.3. 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.

14. Introduction to computational complexity

Chapter 14. Introduction to computational complexity

Move this section to mode advanced version of the book.

14.1 Introduction

Motivation: We want to understand *what* is a problem and *when* a problem is *easy/hard*.

Our strategy: An intuitive review of the basic ideas in Computational Complexity theory.

Key concepts:

- Problem types: optimization problems, decision problems, feasibility problems.
- Instance (of a problem)
- A problem is a collection of instances.
- Size of an instance
- Algorithm
- Running time of an algorithm; worst-case time complexity
- Complexity classes: P and NP

14.2 Problem, instance, size

14.2.1. Problem, instance

Definition 14.1: 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 define next.

Definition 14.2: Instance

Is a specific case of a problem. In other words, we can say “ $\text{Instance} \in \text{Problem}$ ”.

14.2.2. 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.

14.2.3. Size of an instance

The size of an instance is the *amount of information* required to represent the instance (in the computer).

Definition 14.3: Binary size/length

Is the number of bits that are needed in order to give the problem to a computer.

Examples of sizes:

- Size of an integer/rational number.
- Size of a rational matrix.
- Size of a graph (node-edge matrix representation).

14.3 Algorithms, running time, Big-O notation**14.3.1. Basics****Definition 14.4: Algorithm**

List of instructions to solve a problem.

Definition 14.5: Running time of an algorithm

Is the number of steps (as a function of the size) that the algorithm takes in order to solve an instance.

14.3.2. Worst-time complexity

Given an algorithm to solve a problem P , the *running time of*, as a function of the size $\sigma \in \mathbb{Z}_+$ will be defined as follows:

- The (generic) running time will be a function $f : \mathbb{Z}_+ \rightarrow \mathbb{Z}_+$.
- Given σ , the function f is defined as follows:

$$f(\sigma) = \max\{\text{running time of } z \text{, where } \text{size}(z) \leq \sigma\}.$$

Remark: This is a very conservative/pessimistic measure of running time.

14.3.3. Big-O notation

A function $f : \mathbb{Z}_+ \rightarrow \mathbb{Z}_+$ belongs to the class of functions $O(g(n))$ (that is, $f \in O(g(n))$) if there exists $c > 0$, $n_0 \in \mathbb{Z}_+$ such that

$$f(n) \leq cg(n), \quad \text{for all } n \geq n_0.$$

We usually say “ f is $O(g(n))$ ” or “ f is order $O(g(n))$ ”.

14.3.4. Examples

- Basic examples of Big-O notation: $O(1)$, $O(n^k)$, $O(c^n)$, $O(\log(n))$, etc.
- An illustration of the fact that the running time depends on the size of the instance: the algorithm for the binary knapsack problem that is $O(nb)$ is not polynomial, since the size of the instance is $\log(b)$.

14.4 Basics

Definition 14.6: Polynomial time algorithm

An algorithm is said to be a polynomial time algorithm if its running time is $O(n^k)$ for some $k \geq 1$ (where n represents the size of a generic instance).

Remark: *Polynomial time algorithms* are also known as *Polytime algorithms*.

Definition 14.7: Decision problem

A decision problem is any problem whose only acceptable answers are either YES or NO (but not both at the same time).

Some examples: feasibility problems, decision version of optimization problems, etc.

14.5 Complexity classes

We will introduce 3 complexity classes (For see at least 495 more classes, please see http://complexityzoo.uwaterloo.ca/Complexity_Zoo).

14.5.1. Polynomial time problems

Definition 14.8: The class \mathcal{P}

Is the set of all decision problems for which a YES or NO answer for a particular instance can be obtained in polytime.

Remark: For a particular problem P , there are 3 possibilities: (1) $P \in \mathcal{P}$, (2) $P \notin \mathcal{P}$, and $P \not\in \mathcal{P}$ (i.e., we don't know).

Examples:

- Shortest path
- Max flow
- Min cut
- Matroid optimization
- Matchings
- Linear programming

14.5.2. Non-deterministic polynomial time problems

Definition 14.9: The class NP

Is the set of all decision problems for which a YES answer for a particular instance can be verified in polytime.

Examples:

- All problems in \mathcal{P}
- Integer programming
- TSP
- Binary knapsack
- Maximum independent set
- Subset sum
- Partition
- SAT, k -SAT
- Clique

14.5.3. Complements of problems in NP

Definition 14.10: The class NP

Is the set of all decision problems for which a NO answer for a particular instance can be verified in polytime.

Examples:

- All problems in \mathcal{P}
- PRIMES
- Every “complement” of an NP problem

Remark: Actually, $\text{PRIMES} \in \text{NP}$ (see *Pratt's certificates*), even better, it was recently proven that $\text{PRIMES} \in \mathcal{P}$ (by Manindra Agrawal, Neeraj Kayal, Nitin Saxena in 2004).

14.6 Relationship between the classes

14.6.1. A basic result

Theorem 14.11

The following relationship holds:

$$\mathcal{P} \subseteq \text{NP} \cap \text{coNP}.$$

14.6.2. An \$1,000,000 open question

The question “Is $\mathcal{P} = \text{NP}$?” is one of the most important problems in mathematics and computer science. A correct answer is worth 1 Million dollars! Most people believe that $\mathcal{P} \neq \text{NP}$.

14.7 Comparing problems, Polynomial time reductions

Motivation: We would like to solve problem P_1 by efficiently *reducing* it to another problem P_2 (why? Perhaps we know how to solve P_2 !!!).

Definition 14.12: Polynomial time reductions

Let P_1, P_2 be decision problems. We say that P_1 is polynomially reducible to P_2 (denoted $P_1 \leq_{\mathcal{P}} P_2$) if there exists a function

$$f : P_1 \rightarrow P_2$$

such that

- (a) For all $w \in P_1$ the answer to w is YES if and only if the answer to $f(w)$ is YES.
- (b) For all $w \in P_1$ $f(w)$ can be computed in polynomial time w.r.t to $\text{size}(w)$. In particular, we must have that $\text{size}(f(w))$ is polynomially bounded by $\text{size}(w)$.

Remarks:

- (a) In the above definition “ f efficiently transforms an instance of P_1 into an instance of P_2 ”. In particular, if we know how to solve problem P_2 , then we can solve problem P_1 .
- (b) Therefore, the notation $P_1 \leq_{\mathcal{P}} P_2$ makes sense: we are saying that P_1 is “easier” to solve than P_2 , as any algorithm for P_2 would work for P_1 .

14.8 Comparing problems, Polynomial time reductions

14.8.1. Definition

Motivation: We would like to solve problem P_1 by efficiently *reducing* it to another problem P_2 (why? Perhaps we know how to solve P_2 !!!).

Definition 14.13: Polynomial time reductions

Let P_1, P_2 be decision problems. We say that P_1 is polynomially reducible to P_2 (denoted $P_1 \leq_{\mathcal{P}} P_2$) if there exists a function

$$f : P_1 \rightarrow P_2$$

such that

- (a) For all $w \in P_1$ the answer to w is YES if and only if the answer to $f(w)$ is YES.
- (b) For all $w \in P_1$ $f(w)$ can be computed in polynomial time w.r.t to $\text{size}(w)$. In particular, we must have that $\text{size}(f(w))$ is polynomially bounded by $\text{size}(w)$.

Remarks:

- (a) In the above definition “ f efficiently transforms an instance of P_1 into an instance of P_2 ”. In particular, if we know how to solve problem P_2 , then we can solve problem P_1 .
- (b) Therefore, the notation $P_1 \leq_{\mathcal{P}} P_2$ makes sense: we are saying that P_1 is “easier” to solve than P_2 , as any algorithm for P_2 would work for P_1 .

14.8.2. Basic properties

Proposition 14.14

Let P_1, P_2 be two problems such that $P_1 \leq_{\mathcal{P}} P_2$ and assume that $P_2 \in \mathcal{P}$. Then $P_1 \in \mathcal{P}$.

Proposition 14.15

Let P_1, P_2 be two problems such that $P_1 \leq_{\mathcal{P}} P_2$ and assume that $P_2 \in NP$. Then $P_1 \in NP$.

Proposition 14.16

Let P_1, P_2, P_3 be three problems assume that $P_1 \leq_{\mathcal{P}} P_2$ and $P_2 \leq_{\mathcal{P}} P_3$. Then $P_1 \leq_{\mathcal{P}} P_3$.

14.9 NP-Completeness

14.9.1. The basics

Definition 14.17: NP-Completeness

A decision problem P is said to be NP-complete if:

- (a) $P \in NP$
- (b) $Q \leq_{\mathcal{P}} P$ for all $Q \in NP$ (that is, every problem Q in NP can be polynomially reduced to P).

Proposition 14.18

If P is NP-complete and $P \in \mathcal{P}$ then $\mathcal{P} = NP$.

14.9.2. Do NP-complete problems exist?

Theorem 14.19: S. Cook, 1971

SAT is NP-complete.

14.10 NP-Hardness

Definition 14.20: NP-Completeness

A problem P is said to be NP-hard if there exists a NP-complete decision problem that can be reduced to it.

Remarks:

- NP-complete problems are NP-hard.
- Problems in NP-hard not need to be decision problems.
- Optimization versions of NP-complete decision problems are NP-hard.
- If P is NP-hard and $P \in \mathcal{P}$ then $\mathcal{P} = \text{NP}$.

14.11 Exercises

- (a) Let P, Q be decision problems such that every instance of Q is an instance of P (that is $\{\text{instances in } Q\} \subseteq \{\text{instances in } P\}$).
- Give an example of P, Q such that $Q \in \mathcal{P}$ and $P \in \text{NP} - \text{complete}$.
 - Give an example of P, Q such that $Q, P \in \text{NP} - \text{complete}$.

Note: You must prove that your problem belongs to the corresponding class unless we have proved or sketched the proof of that fact in class.

Solution:

- Let P be the *Knapsack problem*. Then P is NP-complete (see Problem 5).
 - Let Q be the special case of the *Knapsack problem* where all weights are 1, that is, $a_1, \dots, a_n = 1$. The following algorithm decides this special case:
ALGORITHM:
 - List the objects $1, \dots, n$ in decreasing order according to c_1, \dots, c_n .

2. Select the first b objects from that list. Call this set S .
3. If $\sum_{i \in S} c_i \leq k$, then the output of the algorithm is YES. Else, the output is NO.

Clearly, this algorithm is correct and runs in polynomial time w.r.t. to the instance. Hence, $Q \in \mathcal{P}$.

- ii.
 - Let P be SAT.
 - Let Q be 3-SAT.

We showed in class that both P and Q are NP-complete problems.

(b) A *Hamiltonian cycle* in a graph $G = (V, E)$ is a simple cycle that contains all the vertices. A *Hamiltonian $s - t$ path* in a graph is a simple path from s to t that contains all of the vertices. The associated decision problems are:

- **Hamiltonian Cycle Input:** $G = (V, E)$. **Question:** Does there exist a hamiltonian cycle in G ?
 - **Hamiltonian Path Input:** $G = (V, E)$, $s, t \in V, s \neq t$. **Question:** Does there exist a hamiltonian $s-t$ path in G ?
- i. Given that *Hamiltonian Cycle* is NP-complete, prove that *Hamiltonian Path* is NP-complete.
 - ii. Given that *Hamiltonian Cycle* is NP-complete, prove that the optimization version of the TSP problem is NP-hard.

Solution:

i. **Step 1: Hamiltonian Path is in NP.**

It is clear that *Hamiltonian Path* is in NP, the certificate to a YES answer is the path itself. Given a Hamiltonian path from s to t . We only need to travel along it to check that in fact it visits every vertex once and that starts in s and ends in t . This takes $O(|E|)$ time.

Step 2: $\text{Hamiltonian Cycle} \leq_P \text{Hamiltonian Path}$.

Given an instance $G = (V, E)$ of *Hamiltonian Cycle*, we construct an instance of *Hamiltonian Path* as follows:

- Let $v \in V$, the we construct the graph $G' = (V', E')$, where:
 - $V' = V \setminus \{v\} \cup \{v_1, v_2\}$ (this takes $O(1)$).
 - $E' = (E \setminus \{\{v, u\} : \{v, u\} \in E\}) \cup \{\{v_1, u\} : \{v, u\} \in E\} \cup \{\{v_2, u\} : \{v, u\} \in E\}$ (this takes $O(|V|)$).
- The instance is: $G' = (V', E')$, $s = v_1$, $t = v_2$.

Notice the construction takes polynomial time w.r.t. the size of the instance.

Finally, the fact that a YES to an instance of *Hamiltonian Cycle* is equivalent to a YES to the associated *Hamiltonian Path* instance follows by noticing that there exists a Hamiltonian cycle of the form

$$vu_1u_2 \dots u_{n-1}v$$

in G if and only if there exists and Hamiltonian v_1-v_2 path in G' of the form

$$v_1u_1u_2 \dots u_{n-1}v_2$$

in G' .

Note: we are denoting $n = |V|$ and the notation $u_0u_1 \dots u_k$ represents the path/cycle that uses the edges $\{u_0, u_1\} \{u_1, u_2\} \dots \{u_{k-1}, u_k\}$ (in that order).

Conclusion: *Hamiltonian Path* is NP-complete.

- ii. Given an instance $G = (V, E)$ of *Hamiltonian Cycle*, the following algorithm decides whether the answer to this instance is yes or no:

ALGORITHM:

1. Given $V = \{v_1, \dots, v_n\}$, consider the cities $\{1, \dots, n\}$.
2. Construct the objective function c given by:

$$c_{ij} = \begin{cases} 0, & \{v_i, v_j\} \in E \\ 1, & \text{else.} \end{cases}$$

3. Solve the TSP instance given above. Let α be the cost of the optimal tour.
4. If $\alpha = 0$, then the output of the algorithm is YES. Else, the output is NO.

The algorithm is correct since the definition of the objective function implies that the optimal tour has cost equals to zero if and only if the tour only travels between pairs of cities associated to edges in E .

Notice that the algorithm only need to solve the TSP problem once and that every other step takes polynomial time. Therefore, this is a valid polynomial time reduction since if we were able to solve the optimization version of the TSP in polynomial time, we would also be able to decide *Hamiltonian Cycle* in polynomial time.

Conclusion: the optimization version of the TSP problem is NP-hard.

- (c) Given that the *node packing problem* is NP – complete, show that the following problems are also NP – complete:
- i. *Node cover*: **Input:** $G = (V, E)$, $k \in \mathbb{Z}_+$. **Question:** Does there exist a set $S \subseteq V$ of size at most k such that every edge of G is incident to a node of S ?
 - ii. *Uncapacitated facility location*. **Input:** sets M, N and integers k, c_{ij}, f_j for $i \in M, j \in N$. **Question:** Is there a set $S \subseteq N$ such that $\sum_{i \in M} \min_{j \in S} c_{ij} + \sum_{j \in S} f_j \leq k$?

Recall that *Node packing problem* is: **Input:** $G = (V, E)$, $k \in \mathbb{Z}_+$. **Question:** Does there exist an independent set of size at least k in G ?

Solution:i. **Step 1: Node cover is in NP.**

It is clear that *Node cover* is in NP, the certificate is the node cover itself. Verifying that the set of nodes is a cover can be done by checking that every edge is connected to a node in the given set. This takes $O(|E||V|)$ time.

Step 2: Node packing \leq_P Node cover.

Given an instance $G = (V, E)$, $l \in \mathbb{Z}_+$ of *Node Packing*, we construct an instance of *Node cover* as follows:

- We construct the graph $G' = (V', E')$, where:
 - $V' = V$ (this takes $O(1)$).
 - $E' = E$ (this takes $O(1)$).
- We take $k = |V| - l$ (this takes $O(1)$).
- The instance is: $G' = (V', E')$, $k \in \mathbb{Z}_+$.

Notice the construction takes polynomial time w.r.t. the size of the instance.

Finally, the fact that a YES to an instance of *Node packing* is equivalent to a YES to the associated *Node cover* instance follows by noticing that a set $U \subseteq V$ is a node packing in G if and only if no edge in E has both end points in U , which is equivalent to say that every edge in E has at least one end point in $V \setminus U$. Equivalently, this is saying that the set $V \setminus U$ is a node cover in G' .

Conclusion: *Node cover* is NP-complete.

ii. **Step 1: Uncapacitated facility location is in NP.**

It is clear that *Uncapacitated facility location* is in NP, because given $S \subseteq N$, we can verify in $O(|M||N| + |N|)$ if

$$\sum_{i \in M} \min_{j \in S} c_{ij} + \sum_{j \in S} f_j \leq k$$

Step 2: Node packing \leq_P Uncapacitated facility location.

Given an instance $G = (V, E)$, $l \in \mathbb{Z}_+$ of *Node Packing*, we construct an instance of *Uncapacitated facility location* as follows:

- $M = E$, $N = V$ (this takes $O(|E|)$).
- The objective

$$c_{ij} = \begin{cases} |V| + 1, & \text{if } j = uv, \text{ where } u, v \neq i \\ 0, & \text{otherwise.} \end{cases}$$

(This takes $O(|E|^2)$.)

- $k = |V| - l$ (this takes $O(1)$).
- $f_j = 1$, for all $j \in N$ (this takes $O(|V|)$).

Notice the construction takes polynomial time w.r.t. the size of the instance.

- **YES to Node Packing \Rightarrow YES to Uncapacitated facility location:**

If U is a node packing, with $|U| \geq l$, then $S = V \setminus U$ is a vertex cover, hence for all $i \in M$:

$$\min\{c_{ij} : j \in S\} = 0.$$

And

$$\sum_{j \in S} f_j = |S| = |V| - |U| \leq |V| - l \leq k.$$

This implies

$$\sum_{i \in M} \min_{j \in S} c_{ij} + \sum_{j \in S} f_j \leq k.$$

Thus, the set $S \subseteq N$ gives a YES answer to *Uncapacitated facility location*.

- **YES to Uncapacitated facility location \Rightarrow YES to Node Packing:**

There exists $S \subseteq N$ such that

$$\sum_{i \in M} \min_{j \in S} c_{ij} + \sum_{j \in S} f_j \leq k.$$

By definition of the reduction from node packing, we have

- (i) For all $i \in M$, $\min\{c_{ij} : j \in S\} \leq |V|$, which implies $\min\{c_{ij} : j \in S\} = 0$.
- (ii) Let $U = V \setminus S$. By (i), $\sum_{j \in S} f_j = |S| = |V| - |U|$.
- (iii) By (i), if $u, v \in U$, then we must have $\{u, v\} \notin E$.
- (iv) By (ii), we have $|U| \geq l$.

This implies that the set $U \subseteq V$ is a node packing with $|U| \geq l$, which gives a YES answer to *Node Packing*.

Conclusion: *Uncapacitated facility location* is NP-complete.

15. Exponential Size Formulations

Chapter 15. Exponential Size Formulations

60% complete. Goal 80% completion date: August 20

Notes:

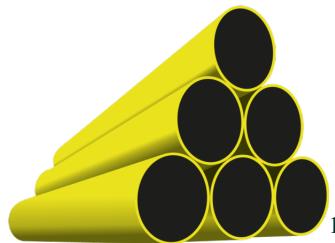
Although typically models need to be a reasonable size in order for us to code them and send them to a solver, there are some ways that we can allow having models of exponential size. The first example here is the cutting stock problem, where we will model with exponentially many variables. The second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

$$\begin{aligned}
 & \min \sum_{j=1}^N z_j \\
 \text{s.t. } & \sum_{i=1}^k L_i x_{ij} \leq L z_j \text{ for } j = 1, \dots, N \\
 & \sum_{j=1}^N x_{ij} \geq d_i \text{ for } i = 1, \dots, k \\
 & z_j \in \{0, 1\} \text{ for } j = 1, \dots, N \\
 & x_{ij} \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k, j = 1, \dots, N
 \end{aligned} \tag{15.1}$$

Exercise 15.2: Show Bound

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

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

$$z_1 \geq z_2 \geq z_3 \geq \cdots \geq z_N. \quad (15.3)$$

See ?? for code for this formulation.

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:

INPUT TABLE OF COMPUTATIONS

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

$$\begin{aligned} \sum_{i=1}^k L_i a_i &\leq L \\ a_i &\in \mathbb{Z}_+ \text{ for all } i = 1, \dots, k \end{aligned} \quad (15.4)$$

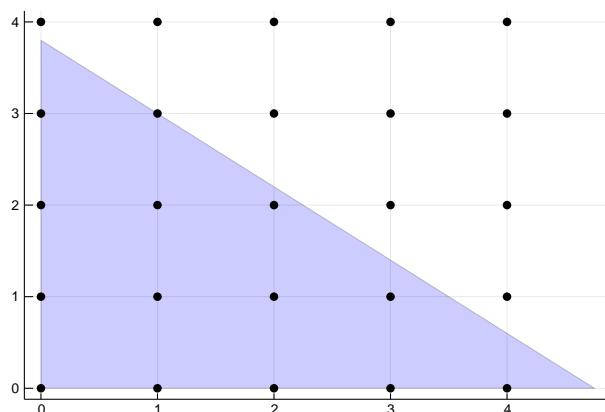
In our running example, we have

$$\begin{aligned} 4a_1 + 5a_2 + 6a_3 &\leq 19 \\ a_i &\in \mathbb{Z}_+ \text{ for all } i = 1, \dots, 3 \end{aligned} \quad (15.5)$$

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

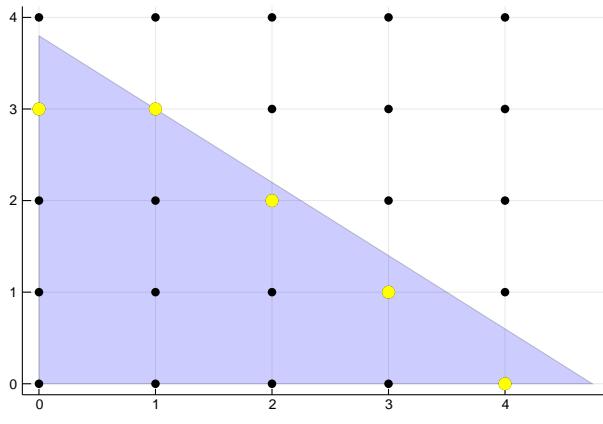
$$P = \{(a_1, a_2) : 4a_1 + 5a_2 \leq 19, a_1 \geq 0, a_2 \geq 0\} \quad (15.6)$$

which we can see here:



© knapsack_fig.pdf²

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.

© knapsack_fig_maximal.pdf³

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

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

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

$$\min \sum_{i=1}^p x_i \quad (15.8)$$

$$\text{such that } Ax \geq \begin{bmatrix} 12 \\ 15 \\ 22 \end{bmatrix} \quad (15.9)$$

$$x \in \mathbb{Z}_+^p \quad (15.10)$$

15.1.2. Column Generation

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

Thus we can write it as

$$\begin{aligned} \min \quad & (c_N - c_B A_B^{-1} A_N) x_N + c_B A_B^{-1} b \\ \text{s.t.} \quad & x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ & x \geq 0 \end{aligned} \quad (15.11)$$

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

$$\begin{aligned} \min \quad & (1_N - 1_B A_B^{-1} N) x_N + 1_B A_B^{-1} b \\ \text{s.t.} \quad & x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ & x \geq 0 \end{aligned} \quad (15.12)$$

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

$$(1 - 1_B A_B^{-1} A_N^i) x_i \quad (15.13)$$

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

$$1 - 1_B A_B^{-1} a < 0 \Rightarrow 1 < 1_B A_B^{-1} a \quad (15.14)$$

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.

$$\begin{aligned} \max \quad & 1_B A_B^{-1} a \\ \text{s.t.} \quad & \sum_{i=1}^k L_i a_i \leq L \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k \end{aligned} \quad (15.15)$$

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.

$$\begin{aligned} \max \quad & \dots \dots a \\ \text{s.t.} \quad & 4a_1 + 5a_2 + 6a_3 \leq 19 \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k \end{aligned} \quad (15.18)$$

15.1.3. Cutting Stock - Multiple widths

Resources

Gurobi has an excellent demonstration application to look at: [Gurobi - Cutting Stock Demo](#)
[Gurobi - Multiple Master Rolls](#)

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 Spanning Trees

Resources

See [Abdelmaguid2018] for a list of 11 models for the minimum spanning tree and a comparison using CPLEX.

15.3 Traveling Salesman Problem

Resources

See math.watloo.ca for excellent material on the TSP.

See also this chapter *A Practical Guide to Discrete Optimization*.

Also, watch this excellent talk by Bill Cook "Postcards from the Edge of Possibility":
[Youtube!](https://www.youtube.com/watch?v=JyfXWzqjwIY)

Google Maps!

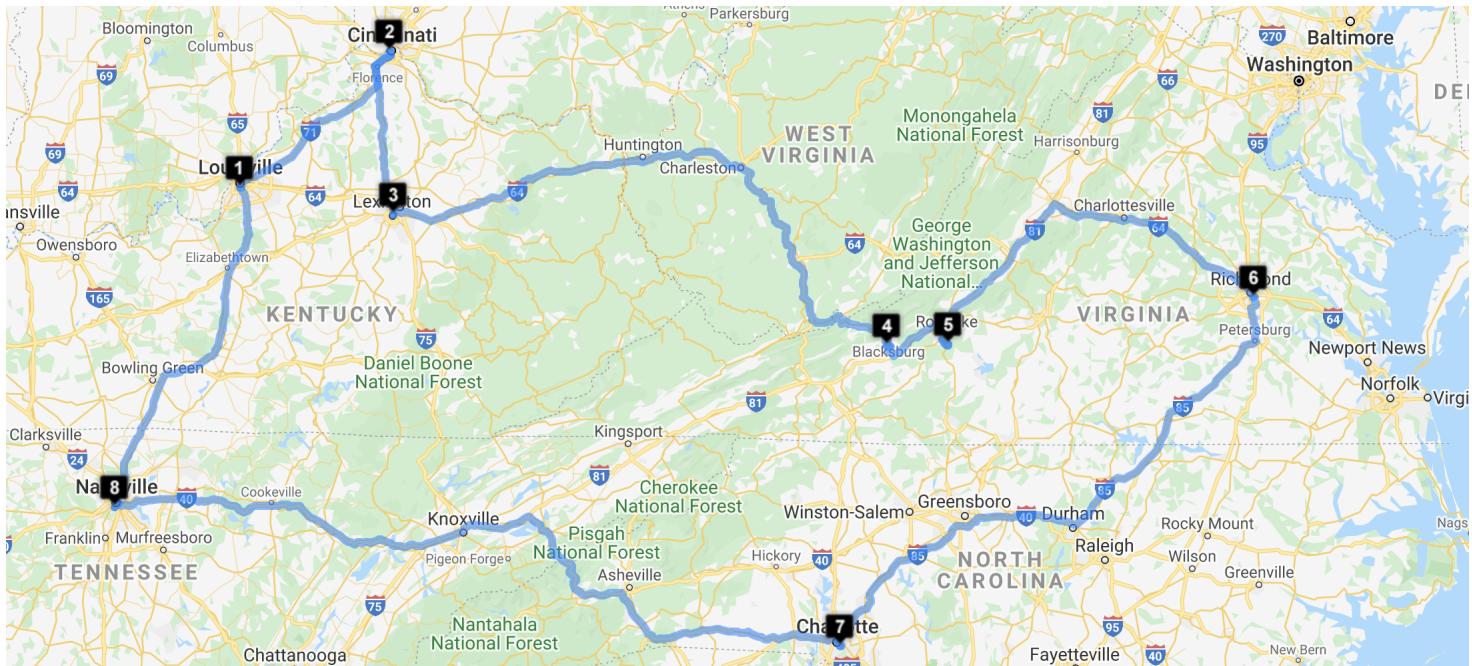
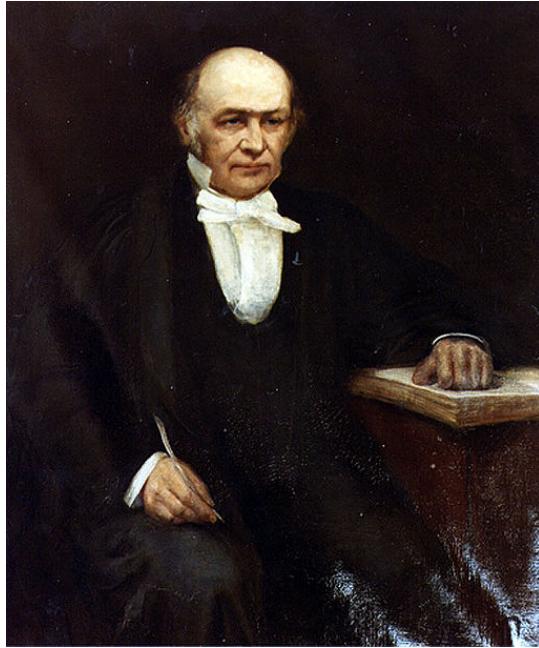


Figure 15.1: Optimal tour through 8 cities. Generated by Gebweb - Optimap. See it also on Google Maps!.

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 $i \rightarrow j$.



© wiki/File/William_Rowan_Hamilton_painting.jpg⁴

Figure 15.2: wiki/File/William_Rowan_Hamilton_painting.jpg

A *tour*, or Hamiltonian cycle (see Figure 15.2), 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.

ADD TSP FIGURE

In the figure, the nodes N are the cities and the arcs A are the directed paths city $i \rightarrow$ 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:

⁴wiki/File/William_Rowan_Hamilton_painting.jpg, from wiki/File/William_Rowan_Hamilton_painting.jpg, wiki/File/William_Rowan_Hamilton_painting.jpg, wiki/File/William_Rowan_Hamilton_painting.jpg.

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (15.1)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \text{ [incoming arc]} \quad (15.2)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \text{ [no self loops]} \quad (15.3)$$

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

ADD SUBTOURS FIGURE

15.3.1. 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: TSP with 4 nodes

Distance Matrix:

A \ B	1	2	3	4

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

Example 15.6: MTZ model for TSP with 4 nodes

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 15.7: MTZ model for TSP with 5 nodes

$$\min \quad x_{1,2} + 2x_{1,3} + 3x_{1,4} + 4x_{1,5} + x_{2,1} + x_{2,3} + 2x_{2,4} + 2x_{2,5} + 2x_{3,1} + \\ x_{3,2} + 4x_{3,4} + x_{3,5} + 3x_{4,1} + 2x_{4,2} + 4x_{4,3} + 2x_{4,5} + \\ 4x_{5,1} + 2x_{5,2} + x_{5,3} + 2x_{5,4}$$

Subject to

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

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

$$\begin{aligned} x_{1,1} &= 0 \\ x_{2,2} &= 0 \\ x_{3,3} &= 0 \\ x_{4,4} &= 0 \\ x_{5,5} &= 0 \end{aligned}$$

$$\begin{aligned} u_1 &= 1 \\ 2 \leq u_i \leq 5 &\quad \forall i \in \{1, 2, 3, 4, 5\} \\ u_2 + 1 &\leq u_3 + 5(1 - x_{2,3}) \\ u_2 + 1 &\leq u_4 + 5(1 - x_{2,4}) \\ u_2 + 1 &\leq u_5 + 5(1 - x_{2,5}) \\ u_3 + 1 &\leq u_2 + 5(1 - x_{3,2}) \\ u_3 + 1 &\leq u_4 + 5(1 - x_{3,4}) \\ u_4 + 1 &\leq u_2 + 5(1 - x_{4,2}) \\ u_4 + 1 &\leq u_3 + 5(1 - x_{4,3}) \\ u_3 + 1 &\leq u_5 + 5(1 - x_{3,5}) \\ u_4 + 1 &\leq u_5 + 5(1 - x_{4,5}) \\ u_5 + 1 &\leq u_2 + 5(1 - x_{5,2}) \\ u_5 + 1 &\leq u_3 + 5(1 - x_{5,3}) \\ u_5 + 1 &\leq u_4 + 5(1 - x_{5,4}) \end{aligned}$$

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

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

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 15.8: 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 15.9: 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 inequalties 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

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

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

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

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

15.3.2. 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:

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (15.19)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (15.20)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (15.21)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (15.22)$$

$$\sum_{(i,j) \in S} x_{ij} \leq |S| - 1 \quad \text{for all subtours } S \quad [\text{prevents subtours}] \quad (15.23)$$

$$x_{ij} \in \{0, 1\} \quad \text{for all } i, j \in N \quad (15.24)$$

Distance Matrix:

A \ B	1	2	3	4
1	0	1	2	3
2	1	0	1	2
3	2	1	0	4
4	3	2	4	0

Example 15.10: DFJ Model for $n = 4$ nodes

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

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 15.11

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. WE WILL DISCUSS THIS IN A FUTURE SECTION ON *cutting planes* .

15.3.3. Traveling Salesman Problem - Branching Solution

We will see in the next section

- That the constraint (15.1)-(15.3) always produce integer solutions as solutions to the linear relaxation.
- A way to use branch and bound (the topic of the next section) in order to avoid subtours.

15.3.4. Traveling Salesman Problem Variants

15.3.4.1. 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-Traveling Salesman Problem - MTZ Model - :

Python Code

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (15.33)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (15.34)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \text{ [incoming arc]} \quad (15.35)$$

$$\sum_{j \in N} x_{Dj} = m \quad \text{for all } i \in N \text{ [outgoing arc]} \quad (15.36)$$

$$\sum_{i \in N} x_{iD} = m \quad \text{for all } j \in N \text{ [incoming arc]} \quad (15.37)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \cup \{D\} \text{ [no self loops]} \quad (15.38)$$

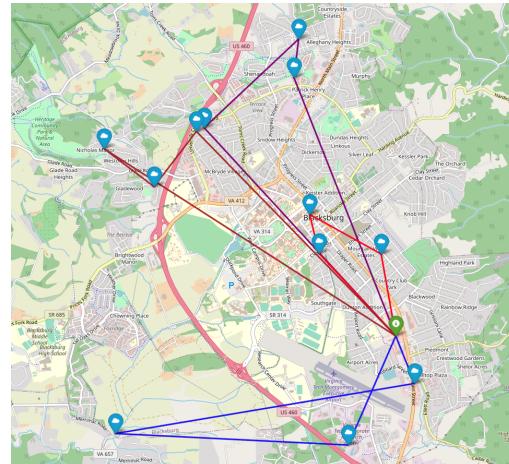
$$u_i + 1 \leq u_j + n(1 - x_{ij}) \quad \text{for all } i, j \in N, i, j \neq 1 \text{ [prevents subtours]} \quad (15.39)$$

$$u_1 = 1 \quad (15.40)$$

$$2 \leq u_i \leq T \quad \text{for all } i \in N, i \neq 1 \quad (15.41)$$

$$u_i \in \mathbb{Z} \quad \text{for all } i \in N \quad (15.42)$$

$$x_{ij} \in \{0, 1\} \quad \text{for all } i, j \in N \cup \{D\} \quad (15.43)$$



© m-tsp_solution⁵

Image html

⁵m-tsp_solution, from m-tsp_solution. m-tsp_solution, m-tsp_solution.

15.3.4.2. 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

$$u_2 + 1 = u_3.$$

15.4 Vehicle Routing Problem (VRP)

The VRP is a generalization of the TSP and comes in many many forms. The major difference is now we may consider multiple vehicles visiting the around cities. Obvious examples are creating bus schedules and mail delivery routes.

Variations of this problem include

- Time windows (for when a city needs to be visited)
- Prize collecting (possibly not all cities need to be visited, but you gain a prize for visiting each city)
- Multi-depot vehicle routing problem (fueling or drop off stations)
- Vehicle rescheduling problem (When delays have been encountered, how do you adjust the routes)
- Inhomogeneous vehicles (vehicles have different abilities (speed, distance, capacity, etc.).)

To read about the many variants, see: *Vehicle Routing: Problems, Methods, and Applications*, Second Edition. Editor(s): Paolo Toth and Daniele Vigo. MOS-SIAM Series on Optimization.

For one example of a VRP model, see GUROBI Modeling Examples - technician routing scheduling.

15.4.1. Case Study: Bus Routing in Boston

Review this case study after studying algorithms and heuristics for integer programming.

[https://www.informs.org/Impact/O.R.-Analytics-Success-Stories/
Optimized-school-bus-routing-helps-school-districts-design-better-policies](https://www.informs.org/Impact/O.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=LFeeaNp_rbY

15.5 Steiner Tree Problem

Model 1

$$\min \sum_{(u,v) \in E} w_{uv} x_{uv}$$

such that

$$\begin{aligned} x_t &= 1 && \forall t \in T \\ 2x_{uv} - x_u - x_v &\leq 0 && \forall (u, v) \in E \\ x_v - \sum_{(u,v) \in E} x_{uv} &\leq 0 \forall v \in V \\ \sum_{(u,v) \in \delta(S)} x_{uv} &\geq x_w \forall S \subseteq V, \forall w \in S \end{aligned}$$

Model 2

$$\begin{aligned} \min \sum_{(u,v) \in E} w_{uv} x_{uv} \\ x_t &= 1 && \forall t \in T \\ 2x_{uv} - x_u - x_v &\leq 0 && \forall (u, v) \in E \\ x_v - \sum_{(u,v) \in E} x_{uv} &\leq 0 && \forall v \in V \\ x_{uv} + x_{vu} &\leq 1 \\ \sum_{v \in V} x_v - \sum_{(u,v) \in E} x_{uv} &= 1 \\ nx_{uv} + l_v - l_u &\geq 1 - n(1 - x_{vu}) && \forall (u, v) \in E \\ nx_{vu} + l_u - l_v &\geq 1 - n(1 - x_{uv}) && \forall (u, v) \in E \end{aligned}$$

Optimizing the shop footprint:

Step 1: Machine learning model predicts effect of opening or closing shops

Step 2: ILP Breaks down shop into manageable clusters

Step 3: ILP for optimal footprint planning

15.6 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

15.6.1. Google maps data

Blog - Python | Calculate distance and duration between two places using google distance matrix API

15.6.2. TSP In Excel

TSP with excel solver

16. Algorithms to Solve Integer Programs

Chapter 16. Algorithms to Solve Integer Programs

50% complete. Goal 80% completion date: September 20

Notes:

16.1 LP to solve IP

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:

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

Theorem 16.1: LP Bounds

It always holds that

$$z_{IP}^* \leq z_{LP}^*. \quad (16.1)$$

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

$$x_{LP}^* = x_{IP}^* \quad \text{and} \quad z_{LP}^* = z_{IP}^*. \quad (16.2)$$

Example 16.2

Consider the problem

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

16.1.1. Rounding LP Solution can be bad!

Resources

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

Consider the two variable knapsack problem

$$\max 3x_1 + 100x_2 \quad (16.3)$$

$$x_1 + 100x_2 \leq 100 \quad (16.4)$$

$$x_i \in \{0, 1\} \text{ for } i = 1, 2. \quad (16.5)$$

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?

Cool trick! Using two different strategies gives you at least a $1/2$ approximation to the optimal solution.

16.1.2. 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.

16.1.3. Fractional Knapsack

The fractional knapsack problem has an exact greedy algorithm.

Resources

- [Youtube!](#)
- [Blog](#)

16.2 Branch and Bound

Resources

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.

16.2.1. Algorithm

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.
 - a: If x^* is integer, stop!
 - b: Otherwise, choose fractional entry x_i^* and branch onto subproblems: (i) $x_i \leq \lfloor x_i^* \rfloor$ and (ii) $x_i \geq \lceil x_i^* \rceil$.
- 3: Solve LP relaxation of any subproblem.
 - a: If LP relaxation is infeasible, prune this node as "**Infeasible**"
 - b: If $z^* < LB$, prune this node as "**Suboptimal**"
 - c: x^* is integer, prune this nodes as "**Integer**" and update $LB = \max(LB, z^*)$.
 - d: Otherwise, choose fractional entry x_i^* and branch onto subproblems: (i) $x_i \leq \lfloor x_i^* \rfloor$ and (ii) $x_i \geq \lceil x_i^* \rceil$.
- Return to step 2 until all subproblems are pruned.

- 4: Return best integer solution found.

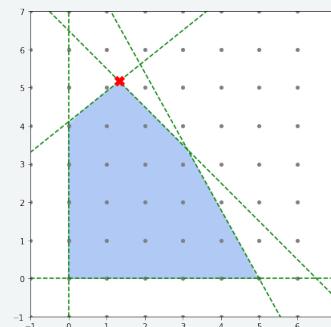
Here is an example of branching on general integer variables.

Example 16.3

Consider the two variable example with

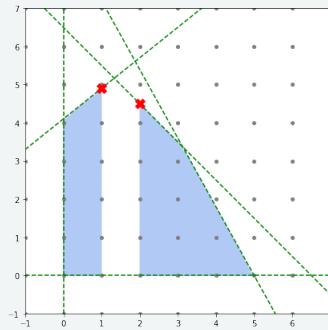
$$\begin{aligned}
 & \max -3x_1 + 4x_2 \\
 & 2x_1 + 2x_2 \leq 13 \\
 & -8x_1 + 10x_2 \leq 41 \\
 & 9x_1 + 5x_2 \leq 45 \\
 & 0 \leq x_1 \leq 10, \text{ integer} \\
 & 0 \leq x_2 \leq 10, \text{ integer}
 \end{aligned}$$

$$x = [1.33, 5.167] \text{ obj} = 16.664$$



$$x = [1, 4.9] \text{ obj} = 16.5998$$

$$x = [2, 4.5] \text{ obj} = 12.0$$



© branch-and-bound2²

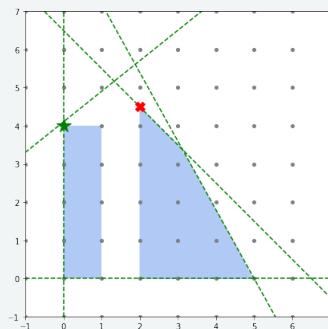
branch-and-bound1, from **branch-and-bound1**. **branch-and-bound1**, **branch-and-bound1**.
branch-and-bound2, from **branch-and-bound2**. **branch-and-bound2**, **branch-and-bound2**.

Example 16.4: Example continued

Infeasible Region

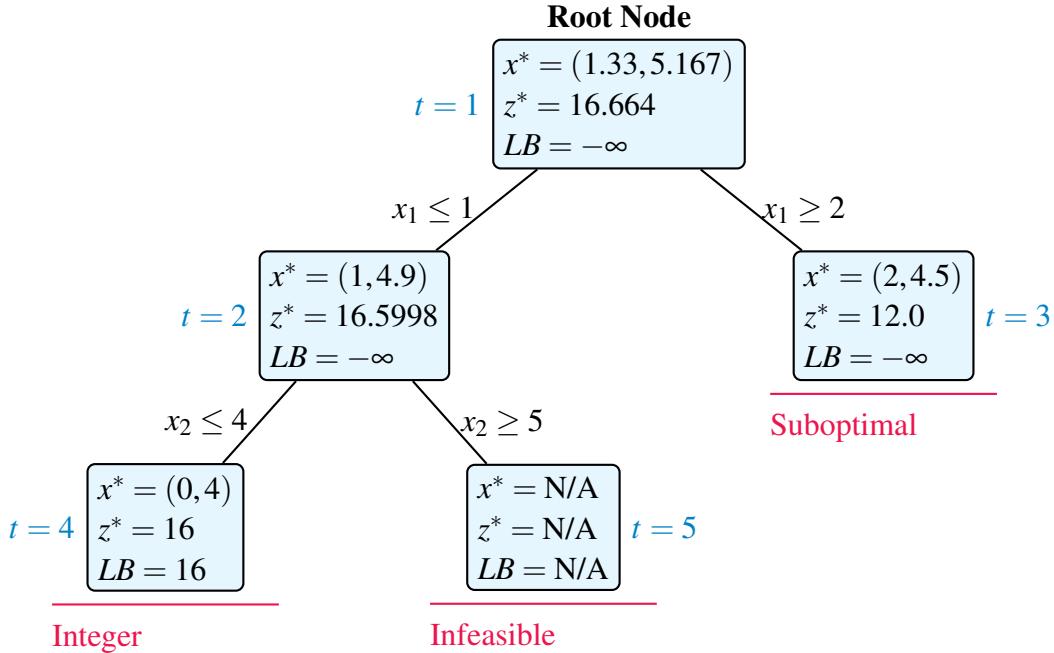
$$x = [0, 4] \text{ obj} = 16.0$$

$$x = [2, 4.5] \text{ obj} = 12.0$$



© branch-and-bound3³

branch-and-bound3, from **branch-and-bound3**. **branch-and-bound3**, **branch-and-bound3**.



16.2.2. Knapsack Problem and 0/1 branching

Consider the problem

$$\begin{aligned}
 & \max \quad 16x_1 + 22x_2 + 12x_3 + 8x_4 \\
 & \text{s.t. } 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \\
 & \quad 0 \leq x_i \leq 1 \quad i = 1, 2, 3, 4 \\
 & \quad x_i \in \{0, 1\} \quad i = 1, 2, 3, 4
 \end{aligned}$$

Question: What is the optimal solution if we remove the binary constraints?

$$\begin{aligned}
 & \max \quad c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 \\
 & \text{s.t. } a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 \leq b \\
 & \quad 0 \leq x_i \leq 1 \quad i = 1, 2, 3, 4
 \end{aligned}$$

Question: How do I find the solution to this problem?

$$\begin{aligned}
 & \max \quad c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 \\
 & \text{s.t. } (a_1 - A)x_1 + (a_2 - A)x_2 + (a_3 - A)x_3 + (a_4 - A)x_4 \leq 0 \\
 & \quad 0 \leq x_i \leq m_i \quad i = 1, 2, 3, 4
 \end{aligned}$$

Question: How do I find the solution to this problem?

Consider the problem

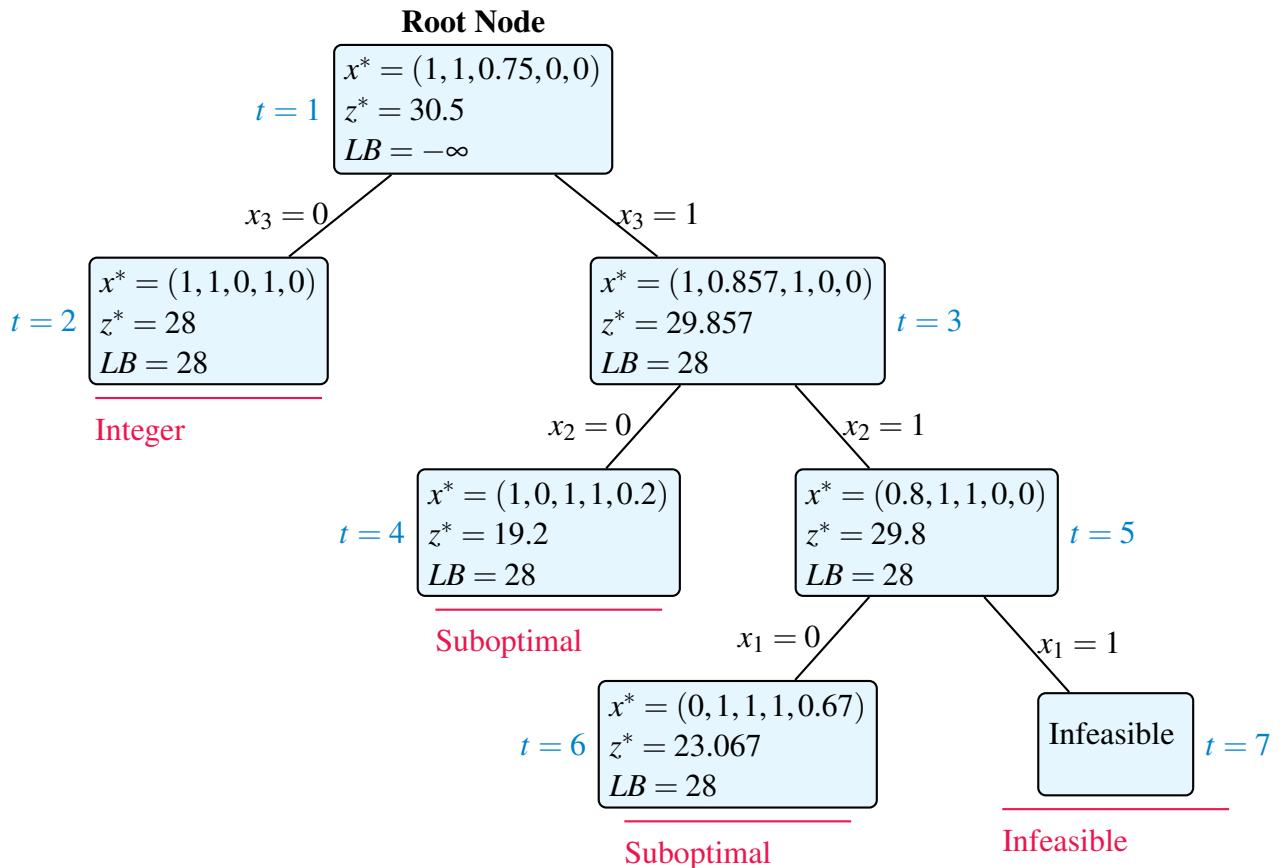
$$\begin{aligned} \max \quad & 16x_1 + 22x_2 + 12x_3 + 8x_4 \\ \text{s.t. } & 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \\ & 0 \leq x_i \leq 1 \quad i = 1, 2, 3, 4 \\ & x_i \in \{0, 1\} \quad i = 1, 2, 3, 4 \end{aligned}$$

We can solve this problem with branch and bound.

The optimal solution was found at $t = 5$ at subproblem 6 to be $x^* = (0, 1, 1, 1)$, $z^* = 42$.

Example: Binary Knapsack Solve the following problem with branch and bound.

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



16.2.3. Traveling Salesman Problem solution via Branching

escribe solving TSP via a generalized branching method that removes subtours (instead of adding constraints).

16.3 Cutting Planes

Cutting planes are inequalities $\pi^\top x \leq \pi_0$ that are valid for the feasible integer solutions that the 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.

The cutting plane procedure, as demonstrated in Figure 16.1. The procedure is as follows:

- Solve the current LP relaxation.
- If solution is integral, then return that solution. STOP
- Add a cutting plane (or many cutting planes) that cut off the LP-optimal solution.
- Return to Step 1.

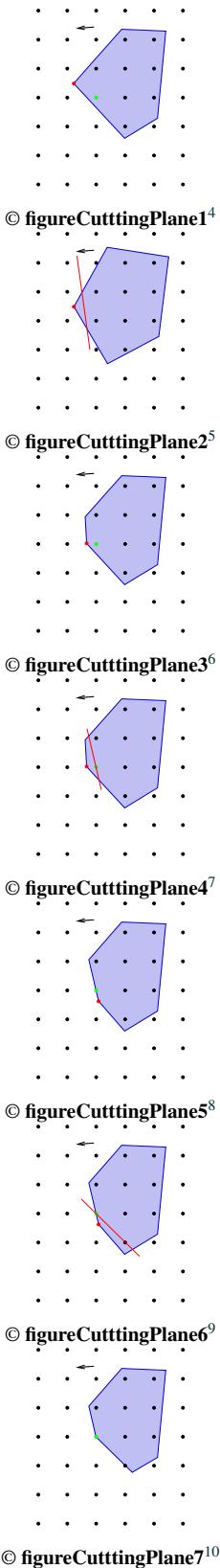


Figure 16.1: The cutting plane procedure.

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

heuristics.

16.3.1. Chvátal Cuts

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

Chvátal Cuts:

Suppose

$$a_1x_1 + \cdots + a_nx_n \leq d \quad (16.1)$$

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

$$\lfloor a_1 \rfloor x_1 + \cdots + \lfloor a_n \rfloor x_n \leq \lfloor d \rfloor \quad (16.2)$$

is valid for the integer points in P , that is, it is valid for the set $P \cap \mathbb{Z}^n$. Equation (16.2) is called a Chvátal Cut.

We will illustrate this idea with an example.

Example 16.5

Recall example ?? . 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.

(a) Divide by 25 and round down both sides:

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

(b) Divide by 10 and round down both sides:

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

(c) Divide by 5 and round down both sides:

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

(d) 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.

16.3.2. Gomory Cuts

Resources

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

Gomory cuts are a type of Chvátal cut that is derived from the simplex tableau. Specifically, suppose that

$$x_i + \sum_{i \in N} \tilde{a}_i x_i = \tilde{b}_i \quad (16.3)$$

is an equation in the optimal simplex tableau.

Gomory Cut:

The Gomory cut corresponding to the tableau row (16.3) is

$$\sum_{i \in N} (\tilde{a}_i - \lfloor \tilde{a}_i \rfloor) x_i \geq \tilde{b}_i - \lfloor \tilde{b}_i \rfloor \quad (16.4)$$

We will solve the following problem using only Gomory Cuts.

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

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

$$\begin{array}{ll} \min & x_1 - 2x_2 \\ \text{s.t.} & -4x_1 + 6x_2 + s_1 = 9 \\ & x_1 + x_2 + s_2 = 4 \\ & x \geq 0, \quad x_1, x_2 \in \mathbb{Z} \end{array} \quad (16.5)$$

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	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 16.6

(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

$$x_1 - 0.1s_1 + -0.6s_2 = 1.5. \quad (16.6)$$

Applying the Gomory Cut (16.4), we have the inequality

$$0.9s_1 + 0.4s_2 \geq 0.5. \quad (16.7)$$

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 16.7: (Gomory Cut in Original Space)

The Gomory Cut (16.7) can be rewritten in the original variables using the equations from (16.5). That is, we can use the equations

$$\begin{aligned} s_1 &= 9 + 4x_1 - 6x_2 \\ s_2 &= 4 - x_1 - x_2, \end{aligned} \quad (16.8)$$

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

$$-3.2x_1 + 5.8x_2 \leq 9.2. \quad (16.9)$$

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

Example 16.8: (Gomory cuts plus new tableau)

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

$$0.9s_1 + 0.4s_2 - s_3 = 0.5. \quad (16.10)$$

Next, we need to solve the linear programming relaxation (where we assume the variables are continuous).

16.3.3. Cover Inequalities

Consider the binary knapsack problem

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

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*

$$\sum_{i \in S} x_i \leq |S| - 1 \Rightarrow x_1 + x_2 + x_3 + x_4 \leq 4 - 1 = 3. \quad (16.11)$$

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

$$x_1 + x_2 \leq 2 - 1 = 1 \quad (16.12)$$

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

$$S = \{1, 4\} \Rightarrow x_1 + x_4 \leq 1 \quad (16.13)$$

and

$$S = \{2, 3, 4\} \Rightarrow x_2 + x_3 + x_4 \leq 2 \quad (16.14)$$



16.4 Branching Rules

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

16.5 Lagrangian Relaxation for Branch and Bound

At each note in the branch and bound tree, we want to bound the objective value. One way to get a good bound can be using the Lagrangian.

Resources

See [Fisher2004] ([link](#)) for a description of this.

16.6 Benders Decomposition

Resources

Benders Decomposition - Julia Opt
Youtube! SCIP lecture

16.7 Literature

16.8 Other material for Integer Linear Programming

Recall the problem on lemonade and lemon juice from Chapter ??:

Problem. Say you are a vendor of lemonade and lemon juice. Each unit of lemonade requires 1 lemon and 2 litres of water. Each unit of lemon juice requires 3 lemons and 1 litre of water. Each unit of lemonade gives a profit of \$3. Each unit of lemon juice gives a profit of \$2. You have 6 lemons and 4 litres of water available. How many units of lemonade and lemon juice should you make to maximize profit?

Letting x denote the number of units of lemonade to be made and letting y denote the number of units of lemon juice to be made, the problem could be formulated as the following linear programming problem:

$$\begin{aligned} \max \quad & 3x + 2y \\ \text{s.t.} \quad & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x \geq 0 \\ & y \geq 0. \end{aligned}$$

The problem has a unique optimal solution at $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1.2 \\ 1.6 \end{bmatrix}$ for a profit of 6.8. But this solution requires us to make fractional units of lemonade and lemon juice. What if we require the number of units to be integers? In other words, we want to solve

$$\begin{aligned} \max \quad & 3x + 2y \\ \text{s.t.} \quad & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x \geq 0 \\ & y \geq 0 \\ & x, y \in \mathbb{Z}. \end{aligned}$$

This problem is no longer a linear programming problem. But rather, it is an integer linear programming problem.

A **mixed-integer linear programming problem** is a problem of minimizing or maximizing a linear function subject to finitely many linear constraints such that the number of variables are finite and at least one of which is required to take on integer values.

If all the variables are required to take on integer values, the problem is called a **pure integer linear programming problem** or simply an **integer linear programming problem**. Normally, we assume the problem data to be rational numbers to rule out some pathological cases.

Mixed-integer linear programming problems are in general difficult to solve yet they are too important to ignore because they have a wide range of applications (e.g. transportation planning, crew scheduling, circuit design, resource management etc.) Many solution methods for these problems have been devised and some of them first solve the **linear programming relaxation** of the original problem, which is the problem obtained from the original problem by dropping all the integer requirements on the variables.

Example 16.9

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

$$\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}, \text{ which is also feasible to (MP). Hence, } \mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix} \text{ is an optimal solution to (MP).}$$

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:

- Set $\text{bestbound} := \infty$, $\mathbf{x}_{\text{best}}^* := \text{N/A}$, $\text{activeproblems} := \{(LP)\}$ where (LP) denotes the linear programming relaxation of (MIP).
- 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.
- Select a problem P from activeproblems and remove it from activeproblems .
- 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}^* .
 - If $z \geq \text{bestbound}$, go to step 2.
 - 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.
 - Set $\mathbf{x}_{\text{best}}^* = \mathbf{x}^*$, $\text{bestbound} = z$ and go to step 2.

Remarks.

- Throughout the algorithm, `activeproblems` is a set of subproblems remained to be solved. Note that for each problem P in `activeproblems`, P is a linear programming problem and that any feasible solution to P satisfying the integrality requirements is a feasible solution to (MIP).
- x_{best}^* is the feasible solution to (MIP) that has the best objective function value found so far and `bestbound` is its objective function value. It is often called an **incumbent**.
- In practice, how a problem from `activeproblems` is selected in step 3 has an impact on the overall performance. However, there is no general rule for selection that guarantees good performance all the time.
- In step 5, the problem P is discarded since it cannot contain any feasible solution to (MIP) having a better objective function value than x_{best}^* .
- If step 7 is reached, then x^* is a feasible solution to (MIP) having objective function value better than `bestbound`. So it becomes the current best solution.
- It is possible for the algorithm to never terminate. Below is an example for which the algorithm will never stop:

$$\begin{aligned} \min \quad & x_1 \\ \text{s.t.} \quad & x_1 + 2x_2 - 2x_3 = 1 \\ & x_1, x_2, x_3 \geq 0 \\ & x_1, x_2, x_3 \in \mathbb{Z}. \end{aligned}$$

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	activeproblems	$\mathbf{x}_{\text{best}}^*$	bestbound
1	(P1)	optimal	(P2): $x_3 \leq 0$, $\mathbf{x}^* = \begin{bmatrix} 0 \\ 0 \\ \frac{2}{3} \end{bmatrix}$ (P3): $x_3 \geq 1$	(P2), (P3)	N/A	∞

Iter	solved	status	branching	active problems	\mathbf{x}^* _{best}	bestbound
2	(P2)	infeasible	—	(P3)	N/A	∞
3	(P3)	optimal $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ \frac{5}{5} \\ 1 \end{bmatrix}$	—	—	$\begin{bmatrix} 0 \\ \frac{4}{5} \\ \frac{5}{5} \\ 1 \end{bmatrix}$	1

Exercises

- (a) Suppose that (MP) in Example 16.8 above has x_2 required to be an integer as well. Continue with the computations and determine an optimal solution to the modified problem.
(b) With the help of a solver, determine the optimal value of

$$\begin{aligned} \max \quad & 3x + 2y \\ \text{s.t.} \quad & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x, y \geq 0 \\ & x, y \in \mathbb{Z}. \end{aligned}$$

- (c) Let $\mathbf{A} \in \mathbb{Q}^{m \times n}$ and $\mathbf{b} \in \mathbb{Q}^m$. Let S denote the system

$$\begin{aligned} \mathbf{Ax} &\geq \mathbf{b} \\ \mathbf{x} &\in \mathbb{Z}^n \end{aligned}$$

- i. 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**.)
ii. 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

- (a) An optimal solution to the modified problem is given by $x^* = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.
(b) An optimal solution is $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$. Thus, the optimal value is 6.
(c) i. Since $\mathbf{d} \geq 0$ and $\mathbf{Ax} \geq \mathbf{b}$, we have $\mathbf{d}^\top \mathbf{Ax} \geq \mathbf{d}^\top \mathbf{b}$. If $\mathbf{d}^\top \mathbf{b}$ is an integer, the result follows immediately. Otherwise, note that $\mathbf{d}^\top \mathbf{A} \in \mathbb{Z}^n$ and $\mathbf{x} \in \mathbb{Z}^n$ imply that $\mathbf{d}^\top \mathbf{Ax}$ is an integer. Thus, $\mathbf{d}^\top \mathbf{Ax}$ must be greater than or equal to the least integer greater than $\mathbf{d}^\top \mathbf{b}$.

ii. Take $\mathbf{d} = \begin{bmatrix} \frac{1}{9} \\ 0 \\ \frac{1}{9} \end{bmatrix}$ and apply the result in the previous part.

16.8.1. Other discrete problems

16.8.2. 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.

16.8.3. History of Computation in Combinatorial Optimization

Book: Computing in Combinatorial Optimization by William Cook, 2019

Model	LP Solution
$\begin{array}{ll} \max & x_1 + x_2 \\ \text{subject to} & -2x_1 + x_2 \leq 0.5 \\ & x_1 + 2x_2 \leq 10.5 \\ & x_1 - x_2 \leq 0.5 \\ & -2x_1 - x_2 \leq -2 \end{array}$	<p>© cutting-plane-1-picture¹¹</p>
$\begin{array}{ll} \max & x_1 + x_2 \\ \text{subject to} & -2x_1 + x_2 \leq 0.5 \\ & x_1 + 2x_2 \leq 10.5 \\ & x_1 - x_2 \leq 0.5 \\ & -2x_1 - x_2 \leq -2 \\ & x_1 \leq 3 \end{array}$	<p>© cutting-plane-2-picture¹²</p>
$\begin{array}{ll} \max & x_1 + x_2 \\ \text{subject to} & -2x_1 + x_2 \leq 0.5 \\ & x_1 + 2x_2 \leq 10.5 \\ & x_1 - x_2 \leq 0.5 \\ & -2x_1 - x_2 \leq -2 \\ & x_1 \leq 3 \\ & x_1 + x_2 \leq 6 \end{array}$	<p>© cutting-plane-3-picture¹³</p>

17. Heuristics for TSP

Chapter 17. Heuristics for TSP

50% complete. Goal 80% completion date: October 20

Notes:

Resources



The image shows a YouTube video thumbnail for a visualization of the Traveling Salesman Problem. The thumbnail features a dark map of the United States with various cities marked by dots. The title "Traveling Salesman Problem" is displayed at the top, followed by the subtitle "Discover the shortest route for visiting a group of cities". Below the video player, it says "Traveling Salesman Problem Visualization" and "325,222 views". The video has 3.3K likes and 40 dislikes. There are buttons for "SHARE", "SAVE", and "SUBSCRIBE 357". The channel is "n Sanity" and the video was published on Aug 18, 2013.

- VRP Heuristic Approach Lecture by Flipkart Delivery Hub

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: $O(n^2)$

- (a) Start from any node (lets call this node 1) and label this as your current node.
- (b) Pick the next current node as the one that is closest to the current node that has not yet been visited.
- (c) Repeat step 2 until all nodes are in the tour.

17.1.3. Insertion Method

Insertion Method:

Complexity: $O(n^2)$

- (a) Start from any 3 nodes (lets call this node 1) and label this as your current node.
- (b) Pick the next current node as the one that is closest to the current node that has not yet been visited.
- (c) Repeat step 2 until all nodes are in the 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 $1 \rightarrow \dots \rightarrow 1$.

- (a) Pick distinct nodes $i, j \neq 1$.
- (b) Let s, t and x_1, \dots, x_k be nodes in the tour such that it can be written as

$$1 \rightarrow \dots \rightarrow s \rightarrow i \rightarrow x_1 \rightarrow \dots \rightarrow x_k \rightarrow j \rightarrow t \rightarrow \dots \rightarrow 1.$$

- (c) 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 $i \rightarrow x_1 \rightarrow \dots \rightarrow x_k \rightarrow j$.

- (d) In this process, we

- deleted the edges (s, i) and (j, t)
- added the edges (s, j) and (i, t)

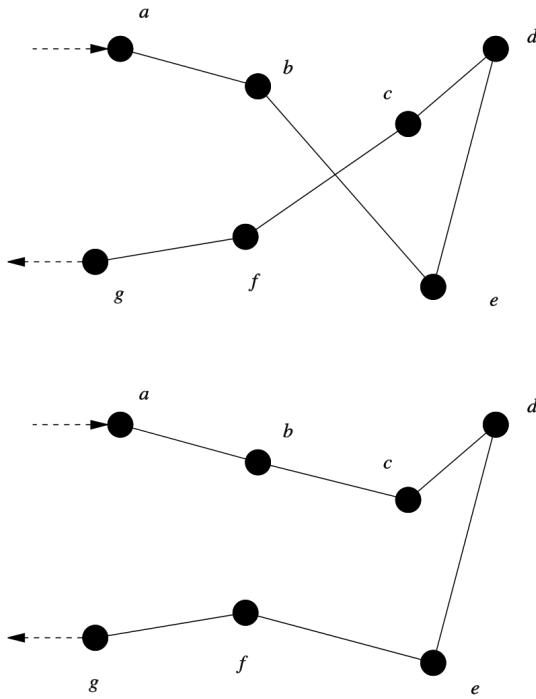
Pictorially, this looks like the following

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.

© wiki/File/2-opt_wiki.png¹**Figure 17.1:** 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.

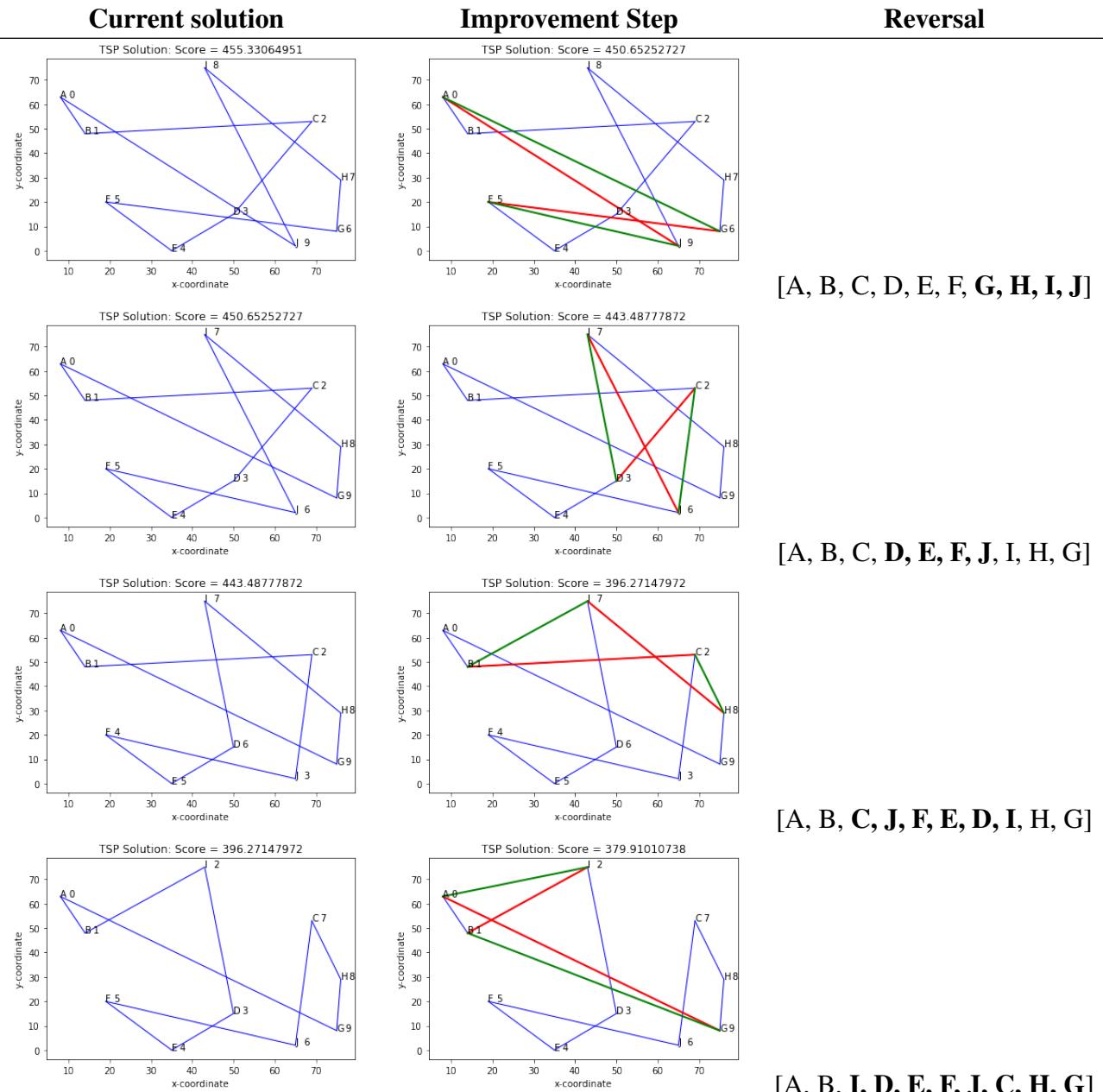
In the context of TSP, we will consider 2-Opt improving moves and the Hill Climbing algorithm for TSP in this case is referred to as the 2-Opt algorithm (also known as the Subtour Reversal Algorithm).

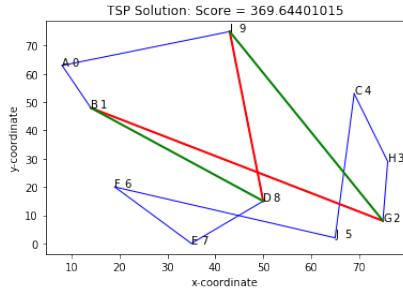
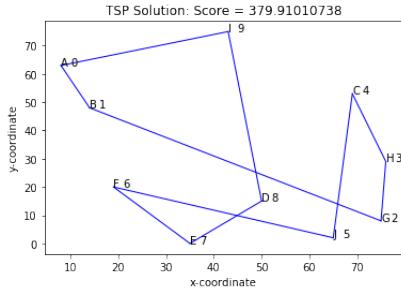
Hill Climbing:

- (a) Start with an initial feasible solution, label it as the current solution.

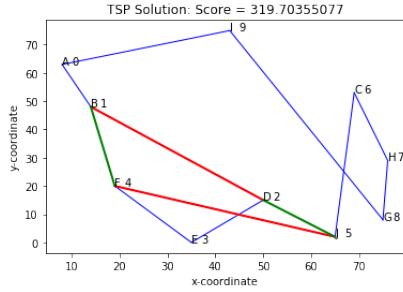
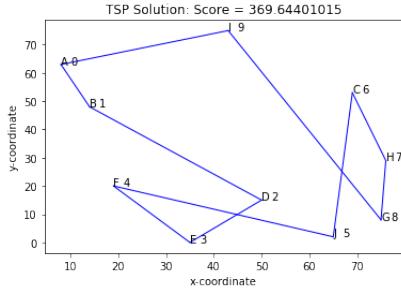
- (b) List all neighbors of the current solution.
 (c) If no neighbor has a better solution, then stop.
 (d) Otherwise, move to the best neighbor and go to Step 2.

Here is an example on the TSP problem with 2-Opt swaps:

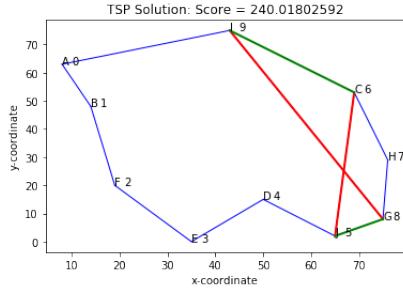
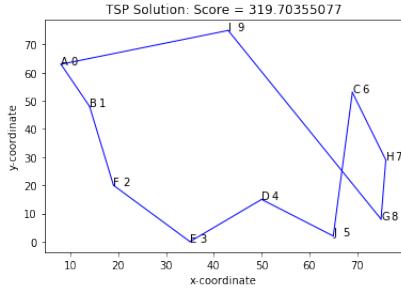




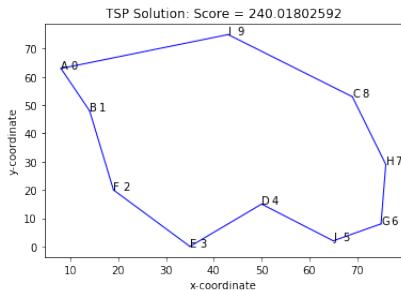
[A, B, G, H, C, J, F, E, D, I]



[A, B, D, E, F, J, C, H, G, I]



[A, B, F, E, D, J, C, H, G, I]



No Improvement

[A, B, F, E, D, J, G, H, C, I]

17.3.2. Simulated Annealing

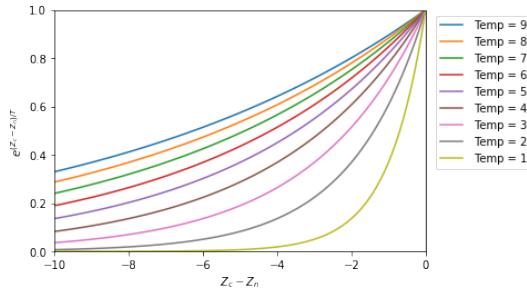
Here is a great python package for TSP Simulated annealing: <https://pypi.org/project/satsp/>.

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 typical 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:

- $T_1 = 0.2Z_c$



© simulated_annealing_temperatures²
Figure 17.2: simulated_annealing_temperatures

- $T_2 = 0.8T_1$
- $T_3 = 0.8T_2$
- $T_4 = 0.8T_3$
- $T_5 = 0.8T_4$

For instance, we could choose to run the algorithm for 10 iterations at each temperature value. The Simulated Annealing algorithm is the following:

Simulated Annealing Outline:

[minimization version]

- (a) Start with an initial feasible solution, label it as the current solution, and compute its objective value Z_c .
- (b) Select a neighbor of the current solution and compute its objective value Z_n .
- (c) Decide whether or not to move to the neighbor:
 - i. If the neighbor is an improvement ($Z_n < Z_c$), **accept the move** and set the neighbor as the current solution.
 - ii. Otherwise, $Z_c < Z_n$ and thus $Z_c - Z_n < 0$. Now with probability $e^{\frac{Z_c - Z_n}{T}}$ accept the move. In detail:
 - Compute the number $p = e^{\frac{Z_c - Z_n}{T}}$
 - Generate a random number $x \in [0, 1]$ from the computer.
 - If $x < p$, then **accept the move**.
 - Otherwise, if $x \geq p$, **reject the move** and stay at the current solution.
- (d) While still iterations left in the schedule, update the temperature T and go to Step 2.
- (e) Return the best found solution during the algorithm.

Figure 17.2

²simulated_annealing_temperatures, from simulated_annealing_temperatures. simulated_annealing_temperatures, simulated_annealing_temperatures.

17.3.3. Tabu Search

Tabu Search Outline:

[minimization version]

- (a) Initialize a *Tabu List* as an empty set: $\text{Tabu} = \{\}$.
- (b) Start with an initial feasible solution, label it as the current solution.
- (c) List all neighbors of the current solution.
- (d) Choose the best neighbor that is not tabu to move too (the move should not be restricted by the set Tabu.)
- (e) Add moves to the Tabu List.
- (f) 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 .
- (g) If no object improvement has been seen for K steps, then Stop.
- (h) 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.

On an instance with 5 nodes:

```

Nearest Neighbor
494
    0.000065 seconds (58 allocations: 2.172 KiB)

Farthest Insertion
494
    0.000057 seconds (49 allocations: 1.781 KiB)

Simulated Annealing
494
    0.000600 seconds (7.81 k allocations: 162.156 KiB)

Math Programming Cbc
494.0
    0.091290 seconds (26.29 k allocations: 1.460 MiB)

Math Programming Gurobi
Academic license - for non-commercial use only
494.0
    0.006610 seconds (780 allocations: 78.797 KiB)

```

One instance on 20 nodes.

```

Nearest Neighbor
790
    0.000162 seconds (103 allocations: 6.406 KiB)

Farthest Insertion
791
    0.000128 seconds (58 allocations: 2.734 KiB)

Simulated Annealing
777
    0.007818 seconds (130.31 k allocations: 2.601 MiB)

Math Programming Cbc
773.0

```

2.738521 seconds (5.76 k allocations: 607.961 KiB)

Math Programming Gurobi
Academic license - for non-commercial use only
773.0
0.238488 seconds (5.68 k allocations: 717.133 KiB)

Nearest Neighbor
1216
0.000288 seconds (142 allocations: 15.141 KiB)

Farthest Insertion
1281
0.000286 seconds (60 allocations: 3.969 KiB)

Simulated Annealing
1227
0.047512 seconds (520.51 k allocations: 10.387 MiB, 19.12% gc time)

Math Programming Cbc
1088.0
6.292632 seconds (20.30 k allocations: 2.111 MiB)

Math Programming Gurobi
Academic license - for non-commercial use only
1088.0
1.349253 seconds (20.16 k allocations: 2.520 MiB)

Part IV

Nonlinear Programming

Part III: Nonlinear Programming

Notes: This Part applies to DORII. Ideally, it will be ready for November 2022.

18. Non-linear Programming (NLP)

Chapter 18. Non-linear Programming (NLP)

50% complete. Goal 80% completion date: November 20

Notes:

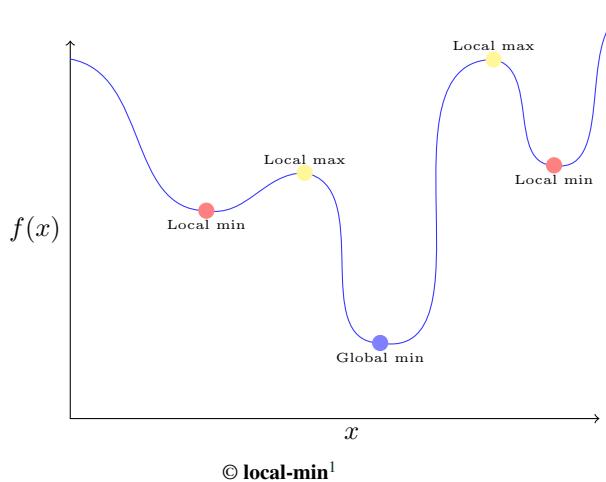
$\min_{x \in \mathbb{R}^n} f(x)$	$\min_{x \in \mathbb{R}^n} f(x)$
Unconstrained Minimization	$f_i(x) \leq 0 \text{ for } i = 1, \dots, m$ Constrained Minimization

- **objective function** $f: \mathbb{R}^n \rightarrow \mathbb{R}$
- may **maximize** f by minimizing the function $g(x) := -f(x)$

Definition 18.1

The 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 $\varepsilon > 0$.



© local-min¹

Theorem 18.2: 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.

¹local-min, from local-min. local-min, local-min.

Definition 18.3: Critical Point

A critical point is a point \bar{x} where $\nabla f(\bar{x}) = 0$.

Theorem 18.4

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., $\nabla f(x^*) = 0$).

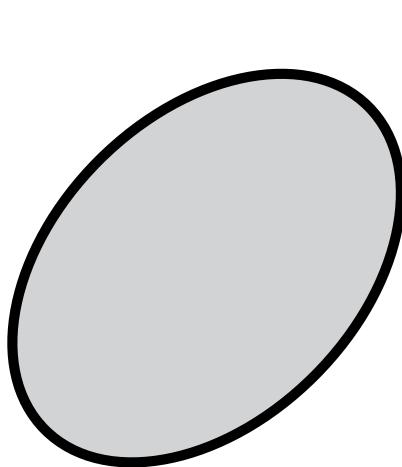
18.1 Convex Sets

Definition 18.5: Convex Combination

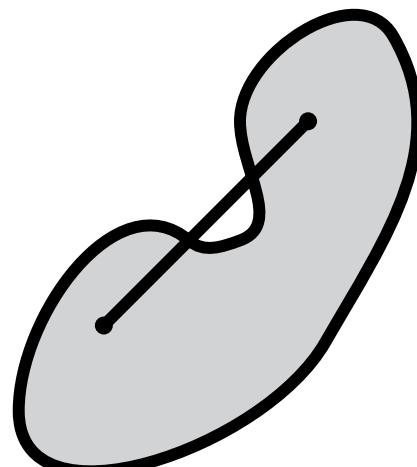
Given two points x, y , a convex combination is any point z that lies on the line between x and y . Algebraically, a convex combination is any point z that can be represented as $z = \lambda x + (1 - \lambda)y$ for some multiplier $\lambda \in [0, 1]$.

Definition 18.6: Convex Set

A set C is convex if it contains all convex combinations of points in C . That is, for any $x, y \in C$, it holds that $\lambda x + (1 - \lambda)y \in C$ for all $\lambda \in [0, 1]$.



(a) Convex Set



(b) Non-convex set

Figure 18.1: Examples of convex and non-convex sets.

Definition 18.7: Convex Sets

A set S is convex if for any two points in S , the entire line segment between them is also contained in S . That is, for any $x, y \in S$

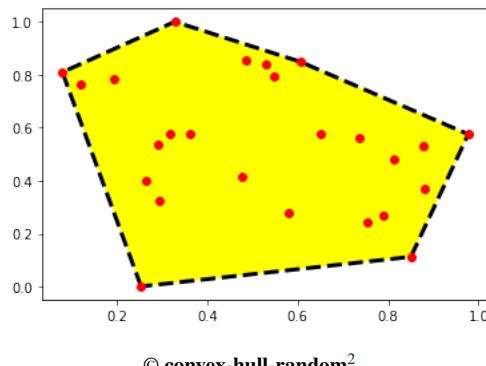
$$\lambda x + (1 - \lambda)y \in S \text{ for all } \lambda \in [0, 1].$$

Examples Convex Sets

- (a) Hyperplane $H = \{x \in \mathbb{R}^n : a^\top x = b\}$
- (b) Halfspace $H = \{x \in \mathbb{R}^n : a^\top x \leq b\}$
- (c) Polyhedron $P = \{x \in \mathbb{R}^n : Ax \leq b\}$
- (d) Sphere $S = \{x \in \mathbb{R}^n : \sum_{i=1}^n x_i^2 \leq 1\}$
- (e) Second Order Cone $S = \{(x, t) \in \mathbb{R}^n \times \mathbb{R} : \sum_{i=1}^n x_i^2 \leq t^2\}$

Definition 18.8: Convex Hull

Let $S \subseteq \mathbb{R}^n$. The convex hull $\text{conv}(S)$ is the smallest convex set containing S .



© convex-hull-random²

Theorem 18.9: Caratheodory's Theorem

Let $x \in \text{conv}(S)$ and $S \subseteq \mathbb{R}^n$. Then there exist $x^1, \dots, x^k \in S$ such that $x \in \text{conv}(\{x^1, \dots, x^k\})$ and $k \leq n + 1$.

18.2 Convex Functions

Convex function are "nice" functions that "open up". They represent an extremely important class of functions in optimization and typically can be optimized over efficiently.

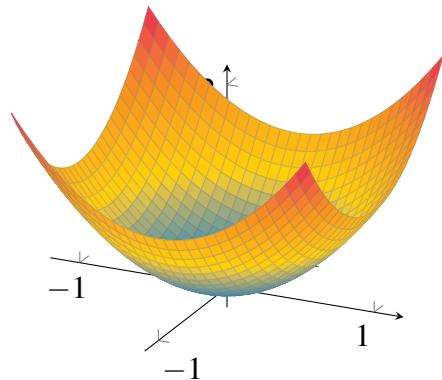


Figure 18.2: Convex Function $f(x,y) = x^2 + y^2$.

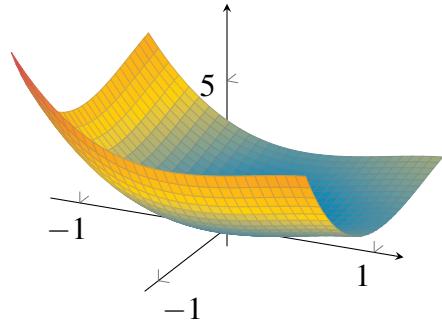


Figure 18.3: Non-Convex Function $f(x,y) = x^2 + y^2 - (x - 0.3)^2 - (y - 0.4)^2$.

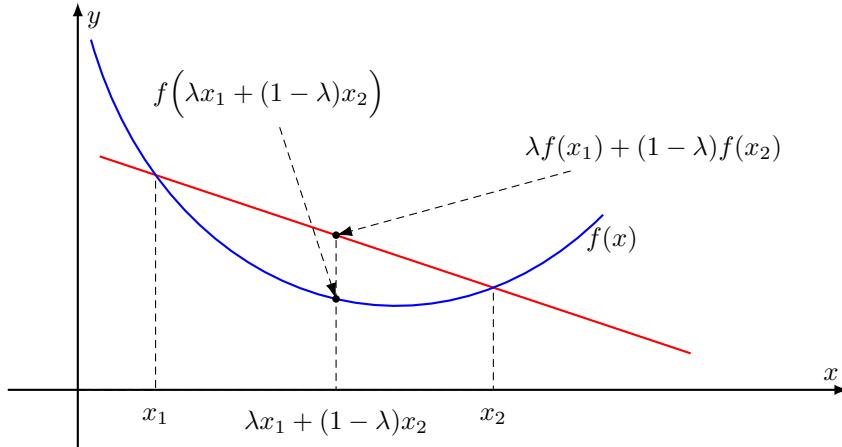
Definition 18.10: Convex Functions

A function $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is convex if for all $x, y \in \mathbb{R}^n$ and $\lambda \in [0, 1]$ we have

$$\lambda f(x) + (1 - \lambda)f(y) \geq f(\lambda x + (1 - \lambda)y). \quad (18.1)$$

³tikz/convexity-definition.pdf, from tikz/convexity-definition.pdf. [tikz/convexity-definition.pdf](https://tex.stackexchange.com/questions/394923/how-one-can-draw-a-convex-function), [tikz/convexity-definition.pdf](https://tex.stackexchange.com/questions/394923/how-one-can-draw-a-convex-function).

³<https://tex.stackexchange.com/questions/394923/how-one-can-draw-a-convex-function>



© tikz/convexity-definition.pdf³

Figure 18.4: Illustration explaining the definition of a convex function.

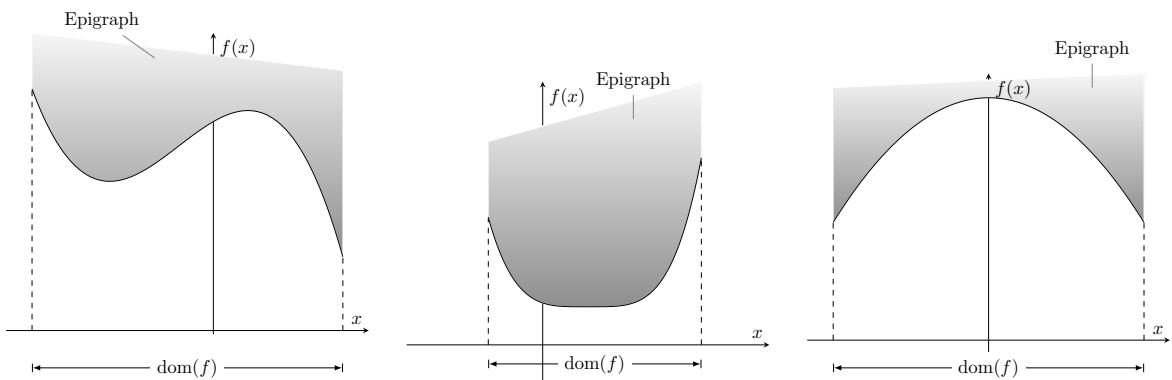
An equivalent definition of convex function are through the epigraph.

Definition 18.11: Epigraph

The epigraph of f is the set $\{(x, y) : y \geq f(x)\}$. This is the set of all points "above" the function.

Theorem 18.12

$f(x)$ is a convex function if and only if the epigraph of f is a convex set.



© epigraph.pdf⁴

⁴epigraph.pdf, from epigraph.pdf. epigraph.pdf, epigraph.pdf.

Example 18.13: Examples of Convex functions

Some examples are

- $f(x) = ax + b$
- $f(x) = x^2$
- $f(x) = x^4$
- $f(x) = |x|$
- $f(x) = e^x$
- $f(x) = -\sqrt{x}$ on the domain $[0, \infty)$.
- $f(x) = x^3$ on the domain $[0, \infty)$.
- $f(x, y) = \sqrt{x^2 + y^2}$
- $f(x, y) = x^2 + y^2 + x$
- $f(x, y) = e^{x+y}$
- $f(x, y) = e^x + e^y + x^2 + (3x + 4y)^6$

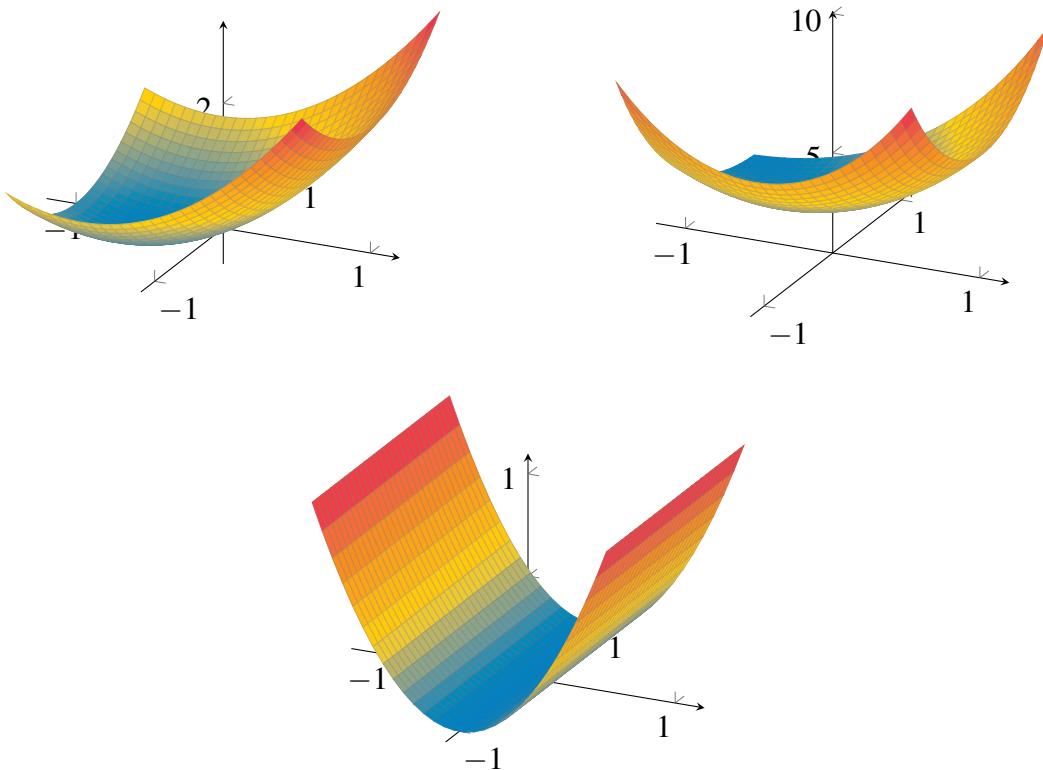


Figure 18.5: Convex Functions $f(x, y) = x^2 + y^2 + x$, $f(x, y) = e^{x+y} + e^{x-y} + e^{-x-y}$, and $f(x, y) = x^2$.

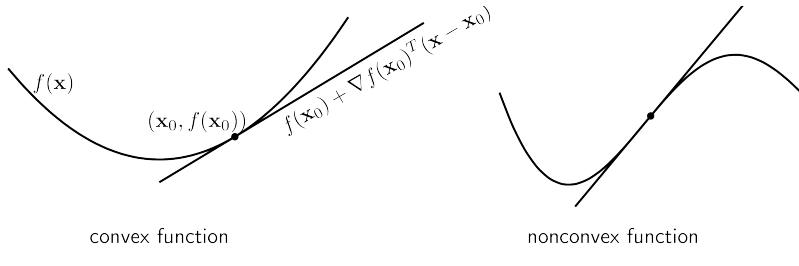
⁵<https://tex.stackexchange.com/questions/261501/function-epigraph-possibly-using-fillbetween>

18.2.1. Proving Convexity - Characterizations

Theorem 18.14: Convexity: First order characterization - linear underestimates

Suppose that $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is differentiable. Then f is convex if and only if for all $\bar{x} \in \mathbb{R}^n$, the linear tangent is an underestimator to the function, that is,

$$f(\bar{x}) + (\bar{x} - x)^T \nabla f(x) \leq f(x).$$



© first-order-convexity⁶

7

Theorem 18.15: Convexity: Second order characterization - positive curvature

We give statements for uni-variate functions and multi-variate functions.

- Suppose $f: \mathbb{R} \rightarrow \mathbb{R}$ is twice differentiable. Then f is convex if and only if $f''(x) \geq 0$ for all $x \in \mathbb{R}$.
- Suppose $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is twice differentiable. Then f is convex if and only if $\nabla^2 f(x) \succcurlyeq 0$ for all $x \in \mathbb{R}^n$.

18.2.2. Proving Convexity - Composition Tricks

Positive Scaling of Convex Function is Convex:

If f is convex and $\alpha > 0$, then αf is convex.

Example: $f(x) = e^x$ is convex. Therefore, $25e^x$ is also convex.

Sum of Convex Functions is Convex:

If f and g are both convex, then $f + g$ is also convex.

Example: $f(x) = e^x, g(x) = x^4$ are convex. Therefore, $e^x + x^4$ is also convex.

⁷<https://machinelearningcoban.com/2017/03/12/convexity/>

Composition with affine function:

If $f(x)$ is convex, then $f(a^\top x + b)$ is also convex.

Example: $f(x) = x^4$ are convex. Therefore, $(3x + 5y + 10z)^4$ is also convex.

Pointwise maximum:

If f_i are convex for $i = 1, \dots, t$, then $f(x) = \max_{i=1, \dots, t} f_i(x)$ is convex.

Example: $f_1(x) = e^{-x}$, $f_2(x) = e^x$ are convex. Therefore, $f(x) = \max(e^x, e^{-x})$ is also convex.

Other compositions:

Suppose

$$f(x) = h(g(x)).$$

- (a) If g is convex, h is convex and **non-decreasing**, then f is convex.
- (b) If g is concave, h is convex and **non-increasing**, then f is convex.

Example 1: $g(x) = x^4$ is convex, $h(x) = e^x$ is convex and non-decreasing. Therefore, $f(x) = e^{x^4}$ is also convex.

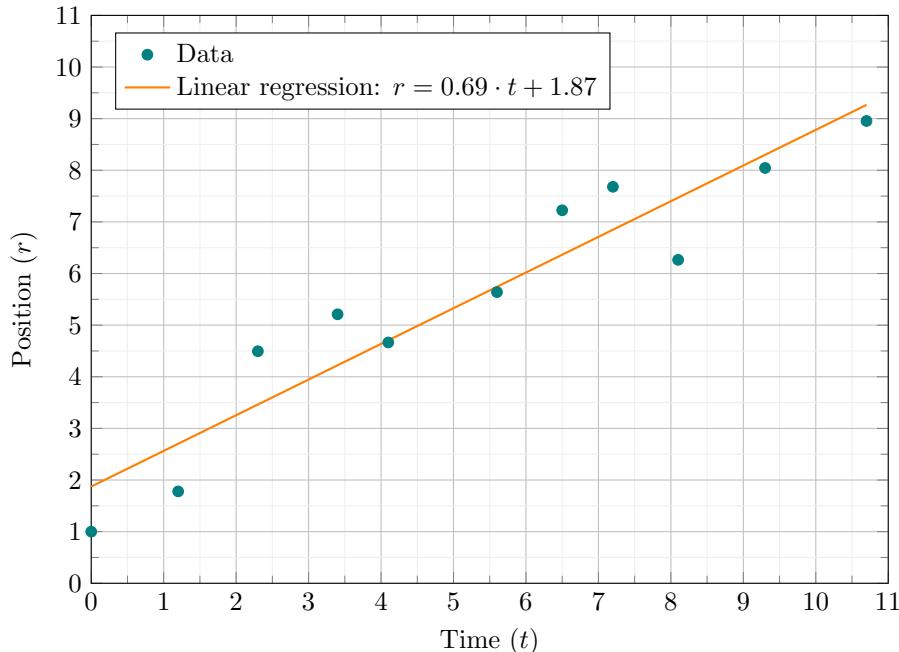
Example 2: $g(x) = \sqrt{x}$ is concave (on $[0, \infty)$), $h(x) = e^{-x}$ is convex and non-increasing. Therefore, $f(x) = e^{-\sqrt{x}}$ is convex on $x \in [0, \infty)$.

18.3 Convex Optimization Examples

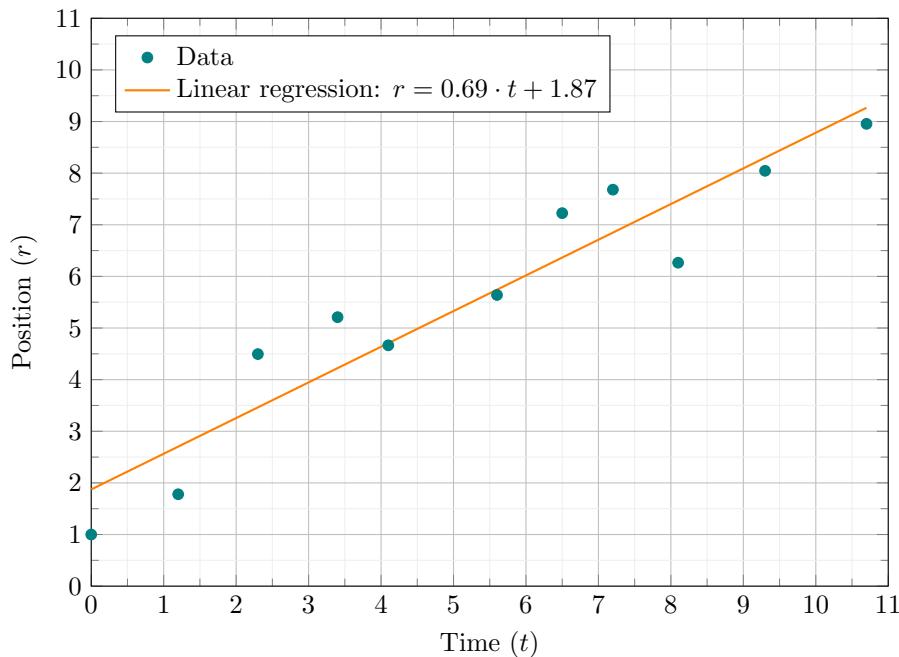
18.3.1. Unconstrained Optimization: Linear Regression

Given data points $x^1, \dots, x^N \in \mathbb{R}^d$ and out values $y^i \in \mathbb{R}$, we want to find a linear function $y = \beta \cdot x$ that best approximates $x^i \cdot \beta \approx y^i$. For example, the data could $x = (\text{time})$ and the output could be $y = \text{position}$.

⁸tikz/linear-regression.pdf, from tikz/linear-regression.pdf. tikz/linear-regression.pdf, tikz/linear-regression.pdf.



© tikz/linear-regression.pdf⁸
Figure 18.6: Line derived through linear regression.



© LinearRegression.pdf⁹
10

As is standard, we choose the error (or "loss") from each data point as the squared error. Hence, we

¹⁰<https://latexdraw.com/linear-regression-in-latex-using-tikz/>

can model this as the optimization problem:

$$\min_{\beta \in \mathbb{R}^d} \sum_{i=1}^N (x^i \cdot \beta - y^i)^2 \quad (18.1)$$

This problem has a nice closed form solution. We will derive this solution, and then in a later section discuss why using this solution might be too slow to compute on large data sets. In particular, the solution comes as a system of linear equations. But when N is really large, we may not have time to solve this system, so an alternative is to use decent methods, discussed later in this chapter.

Theorem 18.16: Linear Regression Solution

The solution to (18.1) is

$$\beta = (X^\top X)^{-1} X^\top Y, \quad (18.2)$$

where

$$X = \begin{bmatrix} x^1 \\ \vdots \\ x^N \end{bmatrix} = \begin{bmatrix} x_1^1 & x_2^1 & \dots & x_d^1 \\ \vdots & \vdots & & \vdots \\ x_1^N & x_2^N & \dots & x_d^N \end{bmatrix}$$

Proof. Solve for $\nabla f(\beta) = 0$.

To be completed....



Resources

<https://www.youtube.com/watch?v=E5RjzSK0fvY>

18.4 Machine Learning - SVM

Support Vector Machine (SVM) is a tool used in machine learning for classifying data points. For instance, if there are red and black data points, how can we find a good line that separates them? The input data that you are given is as follows:

INPUT:

- d -dimensional data points x^1, \dots, x^N
- 1-dimensional labels z^1, \dots, z^N (typically we will use z_i is either 1 or -1)

The output to the problem should be a hyperplane $w^\top x + b = 0$ that separates the two data types (either exact separation or approximate separation).

OUTPUT:

- A d -dimensional vector w
- A 1-dimensional value b

Given this output, we can construct a classification function $f(x)$ as

$$f(x) = \begin{cases} 1 & \text{if } w^\top x + b \geq 0, \\ -1 & \text{if } w^\top x + b < 0. \end{cases} \quad (18.1)$$

There are three versions to consider:

18.4.0.1. Feasible separation

If we only want to a line that separates the data points, we can use the following optimization model.

$$\begin{aligned} \min \quad & 0 \\ \text{such that} \quad & z^i(w^\top x^i + b) \geq 1 \quad \text{for all } i = 1, \dots, N \\ & w \in \mathbb{R}^d \\ & b \in \mathbb{R} \end{aligned}$$

18.4.0.2. Distance between hyperplanes

We want to know a formula for the distance between two parallel hyperplanes. In particular, we will look at the hyperplane $H_0 = \{x : h^\top x = 0\}$ and $H_1 = \{x : h^\top x = 1\}$.

We choose a point $x^0 = 0 \in H_0$, and then find the point in H_1 that minimizes the distance between these two points.

This can be phrased as the optimization problem

$$\begin{aligned} \min \quad & \|x\|_2 \\ \text{s.t.} \quad & h^\top x = 1 \end{aligned}$$

For convenience, we will solve this problem instead:

$$\begin{aligned} \min \quad & \|x\|_2^2 \\ \text{s.t.} \quad & h^\top x = 1 \end{aligned}$$

We will rewrite this problem by removing the equation. We assume, without loss of generality, that $h_n \neq 0$. Otherwise, reorder the variables.

So, since $h_n \neq 0$, we have

$$1 = h_1x_1 + \cdots + h_nx_n$$

and hence

$$x_n = \frac{1 - h_1x_1 - \cdots - h_{n-1}x_{n-1}}{h_n}.$$

Now we can re-write the optimization problem as the unconstrained optimization problem

$$\min f(x) := x_1^2 + \cdots + x_{n-1}^2 + \left(\frac{1 - h_1x_1 - \cdots - h_{n-1}x_{n-1}}{h_n} \right)^2$$

First, note that $f(x)$ is a strictly convex function. Thus, we can find the optimal solution by computing where the gradient vanishes.

$$\nabla f(x) = \begin{bmatrix} \vdots \\ 2x_i + 2\frac{h_i}{h_n} \left(\frac{1 - h_1x_1 - \cdots - h_{n-1}x_{n-1}}{h_n} \right) \\ \vdots \end{bmatrix} = 0$$

But notice what happens when we substitute back in x_n . We obtain

$$\nabla f(x) = \begin{bmatrix} \vdots \\ 2x_i + 2\frac{h_i}{h_n}x_n \\ \vdots \end{bmatrix} = 0$$

Taking the i th equation, we have

$$\frac{x_i}{h_i} = \frac{x_n}{h_n} \quad \text{for all } i = 1, \dots, n-1.$$

Let $\lambda = \frac{x_n}{h_n}$. Then for all $i = 1, \dots, n$, we have $x_i = \lambda h_i$, or in vector form, we have

$$x = \lambda h.$$

Thus, we can go back the original optimization problem and look only at solutions that are of the form $x = \lambda h$.

Plugging this into the original model, we have

$$\begin{aligned} & \min \|\lambda h\|_2 \\ & \text{s.t. } h^\top(\lambda h) = 1 \end{aligned}$$

which is equivalent to

$$\begin{aligned} & \min \lambda \|h\|_2 \\ \text{s.t. } & \lambda = \frac{1}{h^\top h}. \end{aligned}$$

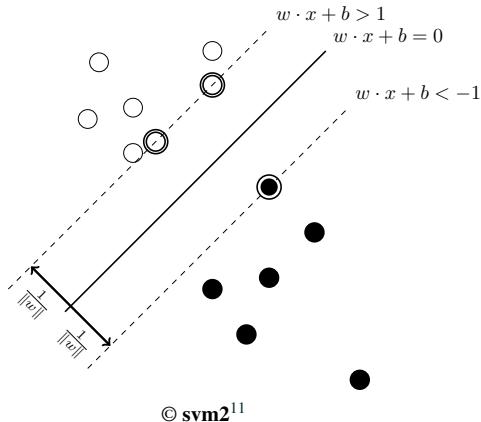
Recall that $h^\top h = \|h\|_2^2$.

Thus, the minimum distance is exactly

$$\frac{1}{\|h\|_2^2} \|h\|_2 = \frac{1}{\|h\|_2}$$

18.4.0.3. SVM

We can modify the objective function to find a best separation between the points. This can be done in the following way



$$\begin{aligned} & \min \|w\|_2^2 \\ \text{such that } & z^i(w^\top x^i + b) \geq 1 \quad \text{for all } i = 1, \dots, N \\ & w \in \mathbb{R}^d \\ & b \in \mathbb{R} \end{aligned}$$

Here, $\|w\|_2^2 = \sum_{i=1}^d w_i^2 = w_1^2 + \dots + w_d^2$.

¹¹svm2, from svm2. svm2, svm2.

18.4.0.4. Approximate SVM

We can modify the objective function and the constraints to allow for approximate separation. This would be the case when you want to ignore outliers that don't fit well with the data set, or when exact SVM is not possible. This is done by changing the constraints to be

$$z^i(w^\top x^i + b) \geq 1 - \delta_i$$

where $\delta_i \geq 0$ is the error in the constraint for datapoint i . In order to reduce these errors, we add a penalty term in the objective function that encourages these errors to be small. For this, we can pick some number C and write the objective as

$$\min \|w\|_2^2 + C \sum_{i=1}^N \delta_i.$$

This creates the following optimization problem

$$\begin{aligned} \min \quad & \|w\|_2^2 + C \sum_{i=1}^N \delta_i \\ \text{such that} \quad & z^i(w^\top x^i + b) \geq 1 - \delta_i && \text{for all } i = 1, \dots, N \\ & w \in \mathbb{R}^d \\ & b \in \mathbb{R} \\ & \delta_i \geq 0 \text{ for all } i = 1, \dots, N \end{aligned}$$

See information about the scikit-learn module for svm here: <https://scikit-learn.org/stable/modules/svm.html>.

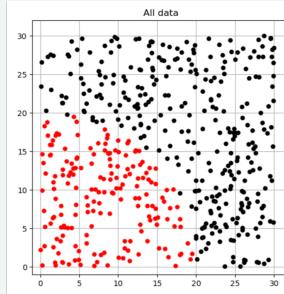
18.4.1. SVM with non-linear separators

Resources

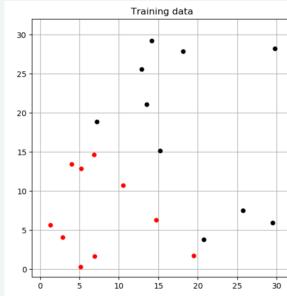
https://www.youtube.com/watch?time_continue=6&v=N1v0golbjSc

Suppose for instance you are given data $x^1, \dots, x^N \in \mathbb{R}^2$ (2-dimensional data) and given labels are dependent on the distance from the origin, that is, all data points x with $x_1^2 + x_2^2 > r$ are given a label $+1$ and all data points with $x_1^2 + x_2^2 \leq r$ are given a label -1 . That is, we want to learn the function

$$f(x) = \begin{cases} 1 & \text{if } x_1^2 + x_2^2 > r, \\ -1 & \text{if } x_1^2 + x_2^2 \leq r. \end{cases} \quad (18.2)$$

Example 18.17

© svm-nonlinear¹²



© svm-nonlinear-training¹³

Here we have a *classification problem where the data cannot be separated by a hyperplane*. On the left, we have all of the data given to use. On the right, we have a subset of the data that we could try using for training and then test our learned function on the remaining data. As we saw in class, this amount of data was not sufficient to properly classify most of the data.

svm-nonlinear, from **svm-nonlinear**. **svm-nonlinear**, **svm-nonlinear**.

svm-nonlinear-training, from **svm-nonlinear-training**.

svm-nonlinear-training,

svm-nonlinear-training.

We cannot learn this classifier from the data directly using the hyperplane separation with SVM in the last section. But, if we modify the data set, then we can do this.

For each data point x , we transform it into a data point X by adding a third coordinate equal to $x_1^2 + x_2^2$. That is

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow X = \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 + x_2^2 \end{pmatrix}. \quad (18.3)$$

In this way, we convert the data x^1, \dots, x^N into data X^1, \dots, X^N that lives in a higher-dimensional space. But with this new dataset, we can apply the hyperplane separation technique in the last section to properly classify the data.

This can be done with other nonlinear classification functions.

18.4.2. Support Vector Machines

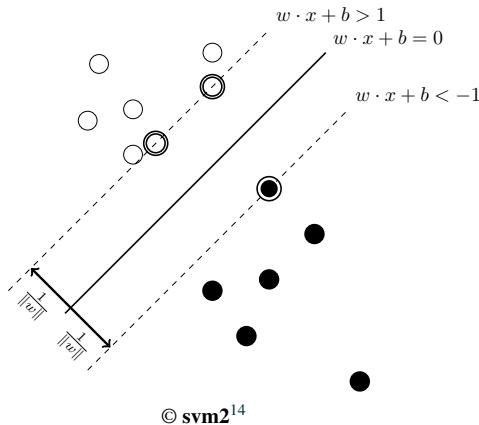
Support Vector Machine - Exact Classification:

Given labeled data (x^i, y_i) for $i = 1, \dots, N$, where $x^i \in \mathbb{R}^d$ and $y^i \in \{-1, 1\}$, find a vector $w \in \mathbb{R}^d$ and a number $b \in \mathbb{R}$ such that

$$x^i \cdot w + b > 0 \quad \text{if } y^i = 1 \quad (18.4)$$

$$x^i \cdot w + b < 0 \quad \text{if } y^i = -1 \quad (18.5)$$

There may exist many solutions to this problem. Thus, we are interested in the "best" solution. Such a solution will maximize the separation between the two sets of points. To consider an equal margin on either side, we set the right hand sides to 1 and -1 and then compute the margin from the hyperplane. Notice that it is sufficient to use 1 and -1 on the right hand sides since any scaling can happen in w and b .



We will show that the margin under this model can be computed as $\frac{2}{\|w\|}$ where $\|w\| = \sqrt{w_1^2 + \dots + w_d^2}$. Hence, maximizing the margin is equivalent to minimizing $w_1^2 + \dots + w_d^2$. We arrive at the model

$$\min \sum_{i=1}^d w_i^2 \quad (18.6)$$

$$x^i \cdot w + b \geq 1 \quad \text{if } y^i = 1 \quad (18.7)$$

$$x^i \cdot w + b \leq -1 \quad \text{if } y^i = -1 \quad (18.8)$$

¹⁴svm2, from **svm2**. **svm2**, **svm2**.

Or even more compactly written as

$$\min \sum_{i=1}^d w_i^2 \quad (18.9)$$

$$y^i(x^i \cdot w + b) \geq 1 \quad \text{for } i = 1, \dots, N \quad (18.10)$$

18.5 Classification

18.5.1. Machine Learning

Resources

https://www.youtube.com/watch?v=bwZ3Qiuj3i8&list=PL9ooVrP1hQOHUfd-g8GUpKI3hH0wM_9Dn&index=13

<https://towardsdatascience.com/solving-a-simple-classification-problem-with-python>

18.5.2. Neural Networks

Resources

<https://www.youtube.com/watch?v=bVQUSndD11U>

<https://www.youtube.com/watch?v=8bNIkfRJZpo>

<https://www.youtube.com/watch?v=Dws9Zveu9ug>

18.6 Box Volume Optimization in Scipy.Minimize

<https://www.youtube.com/watch?v=iSnTtV6b0Gw>

18.7 Modeling

We will discuss a few models and mention important changes to the models that will make them solvable.

IMPORTANT TIPS

- (a) **Find a convex formulation.** It may be that the most obvious model for your problem is actually non-convex. Try to reformulate your model into one that is convex and hence easier for solvers to handle.
- (b) **Intelligent formulation.** Understanding the problem structure may help reduce the complexity of the problem. Try to deduce something about the solution to the problem that might make the problem easier to solve. This may work for special cases of the problem.
- (c) **Identify problem type and select solver.** Based on your formulation, identify which type of problem it is and which solver is best to use for that type of problem. For instance, Gurobi can handle some convex quadratic problems, but not all. Ipopt is a more general solver, but may be slower due to the types of algorithms that it uses.
- (d) **Add bounds on the variables.** Many solvers perform much better if they are provided bounds to the variables. This is because it reduces the search region where the variables live. Adding good bounds could be the difference in the solver finding an optimal solution and not finding any solution at all.
- (e) **Warm start.** If you know good possible solutions to the problem (or even just a feasible solution), you can help the solver by telling it this solution. This will reduce the amount of work the solver needs to do. In JUMP this can be done by using the command `setvalue(x,[2 4 6])`, where here it sets the value of vector x to [2 4 6]. It may be necessary to specify values for all variables in the problem for it to start at.
- (f) **Rescaling variables.** It sometimes is useful to have all variables on the same rough scale. For instance, if minimizing $x^2 + 100^2y^2$, it may be useful to define a new variable $\bar{y} = 100y$ and instead minimize $x^2 + \bar{y}^2$.
- (g) **Provide derivatives.** Working out gradient and hessian information by hand can save the solver time. Particularly when these are sparse (many zeros). These can often be provided directly to the solver.

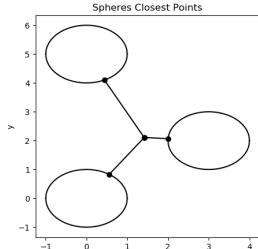
See <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=4982C26EC5F25564BCC239FD3785E2D3?doi=10.1.1.210.3547&rep=rep1&type=pdf> for many other helpful tips on using Ipopt.

18.7.1. Minimum distance to circles

The problem we will consider here is: Given n circles, find a center point that minimizes the sum of the distances to all of the circles.

Minimize distance to circles:

Given circles described by center points (a_i, b_i) and radius r_i for $i = 1, \dots, n$, find a point $c = (c_x, c_y)$ that minimizes the sum of the distances to the circles.



© circles-figure¹⁵

Minimize distance to circles - Model attempt #1:

Non-convex

Let (x_i, y_i) be a point in circle i . Let w_i be the distance from (x_i, y_i) to c . Then we can model the problem as follows:

$$\begin{array}{ll} \min & \sum_{i=1}^3 w_i & \text{Sum of distances} \\ \text{s.t.} & \sqrt{(x_i - a_i)^2 + (y_i - b_i)^2} = r, \quad i = 1, \dots, n & (x_i, y_i) \text{ is in circle } i \\ & \sqrt{(x_i - c_x)^2 + (y_i - c_y)^2} = w_i, \quad i = 1, \dots, n & w_i \text{ is distance from } (x_i, y_i) \text{ to } c \end{array} \quad (18.1)$$

This model has several issues:

- (a) If the center c lies inside one of the circles, then the constraint $\sqrt{(x_i - a_i)^2 + (y_i - b_i)^2} = r$ may not be valid. This is because the optimal choice for (x_i, y_i) in this case would be inside the circle, that is, satisfying $\sqrt{(x_i - a_i)^2 + (y_i - b_i)^2} \leq r$.
 - (b) This model is **nonconvex**. In particular the equality constraints make the problem nonconvex.
- Fortunately, we can relax the problem to make it convex and still model the correct solution. In particular, consider the constraint

$$\sqrt{(x_i - c_x)^2 + (y_i - c_y)^2} \leq w_i.$$

Since we are minimizing $\sum w_i$, it is equivalent to have the constraint

$$\sqrt{(x_i - c_x)^2 + (y_i - c_y)^2} \leq w_i.$$

This is equivalent because any optimal solution makes w_i the smallest it can, and hence will meet that constraint at equality.

What is great about this change, is that it makes the constraint **convex!**. To see this we can write $f(z) = \|z\|_2^2$, $z = (x_i - c_x, y_i - c_y)$. Since $f(z)$ is convex and the transformation into variables x_i, c_x, y_i, c_y is linear, we have that $f(x_i - c_x, y_i - c_y)$ is convex. Then since $-w_i$ is linear, we have that

$$f(x_i - c_x, y_i - c_y) - w_i$$

is a convex function. Thus, the constraint

$$f(x_i - c_x, y_i - c_y) - w_i \leq 0$$

¹⁵circles-figure, from circles-figure. circles-figure, circles-figure.

is a convex constraint.

This brings us to our second model.

Minimize distance to circles - Model attempt #2:

Convex

Let (x_i, y_i) be a point in circle i . Let w_i be the distance from (x_i, y_i) to c . Then we can model the problem as follows:

$$\begin{array}{ll} \min & \sum_{i=1}^3 w_i & \text{Sum of distances} \\ \text{s.t.} & \sqrt{(x_i - a_i)^2 + (y_i - b_i)^2} \leq r, \quad i = 1, \dots, n & (x_i, y_i) \text{ is in circle } i \\ & \sqrt{(x_i - c_x)^2 + (y_i - c_y)^2} \leq w_i, \quad i = 1, \dots, n & w_i \text{ is distance from } (x_i, y_i) \text{ to } c \end{array} \quad (18.2)$$

Lastly, we would like to make this model better for a solver. For this we will

- (a) Add bounds on all the variables
- (b) Change format of non-linear inequalities

Minimize distance to circles - Model attempt #3:

Convex

Let (x_i, y_i) be a point in circle i . Let w_i be the distance from (x_i, y_i) to c . Then we can model the problem as follows:

$$\begin{array}{ll} \min & \sum_{i=1}^3 w_i & \text{Sum of distances} \\ \text{s.t.} & (x_i - a_i)^2 + (y_i - b_i)^2 \leq r^2, \quad i = 1, \dots, n & (x_i, y_i) \text{ is in circle } i \\ & (x_i - c_x)^2 + (y_i - c_y)^2 \leq w_i^2, \quad i = 1, \dots, n & w_i \text{ is distance from } (x_i, y_i) \text{ to } c \\ & 0 \leq w_i \leq u_i \\ & a_i - r \leq x_i \leq a_i + r \\ & b_i - r \leq y_i \leq b_i + r \end{array} \quad (18.3)$$

Example: Minimize distance to circles

Gurobipy

Here we minimize the distance of three circles of radius 1 centered at $(0,0)$, $(3,2)$, and $(0,5)$.

Note: The bounds on the variables here are not chosen optimally.

$$\begin{aligned}
 \min \quad & w_1 + w_2 + w_3 \\
 \text{Subject to} \quad & (x_1 - 0)^2 + (y_1 - 0)^2 \leq 1 \\
 & (x_2 - 3)^2 + (y_2 - 2)^2 \leq 1 \\
 & (x_3 - 0)^2 + (y_3 - 5)^2 \leq 1 \\
 & (x_1 - c_x)^2 + (y_1 - c_y)^2 \leq w_1^2 \\
 & (x_2 - c_x)^2 + (y_2 - c_y)^2 \leq w_2^2 \\
 & (x_3 - c_x)^2 + (y_3 - c_y)^2 \leq w_3^2 \\
 & -1 \leq x_i \leq 10 \quad \forall i \in \{1, 2, 3\} \\
 & -1 \leq y_i \leq 10 \quad \forall i \in \{1, 2, 3\} \\
 & 0 \leq w_i \leq 40 \quad \forall i \in \{1, 2, 3\} \\
 & -1 \leq c_x \leq 10 \\
 & -1 \leq c_y \leq 10
 \end{aligned}$$

18.8 Machine Learning

There are two main fields of machine learning:

- Supervised Machine Learning,
- Unsupervised Machine Learning.

Supervised machine learning is composed of *Regression* and *Classification*. This area is thought of as being given labeled data that you are then trying to understand the trends of this labeled data.

Unsupervised machine learning is where you are given unlabeled data and then need to decide how to label this data. For instance, how can you optimally partition the people in a room into 5 groups that share the most commonalities?

18.9 Machine Learning - Supervised Learning - Regression

See the video lecture information.

18.10 Machine learning - Supervised Learning - Classification

The problem of data *classification* begins with *data* and *labels*. The goal is *classification* of future data based on sample data that you have by constructing a function to understand future data.

Goal: Classification - create a function $f(x)$ that takes in a data point x and outputs the correct label.

These functions can take many forms. In binary classification, the label set is $\{+1, -1\}$, and we want to correctly (as often as we can) determine the correct label for a future data point.

There are many ways to determine such a function $f(x)$. In the next section, we will learn about SVM that determines the function by computing a hyperplane that separates the data labeled $+1$ from the data labeled -1 .

Later, we will learn about *neural networks* that describe much more complicated functions.

Another method is to create a *decision tree*. These are typically more interpretable functions (neural networks are often a bit mysterious) and thus sometimes preferred in settings where the classification should be easily understood, such as a medical diagnosis. We will not discuss this method here since it fits less well with the theme of nonlinear programming.

18.10.1. Python SGD implementation and video

https://github.com/l1Sourcell/Classifying_Data_Using_a_Support_Vector_Machine/blob/master/support_vector_machine_lesson.ipynb

19. NLP Algorithms

Chapter 19. NLP Algorithms

50% complete. Goal 80% completion date: November 20

Notes:

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, $\varepsilon = 10^{-6}$.

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 $\epsilon = \frac{b-a}{2}$. 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

- (a) $x^* \in [a, x_2]$ if $f(x_1) \leq f(x_2)$,
- (b) $x^* \in [x_1, b]$ 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

- (a) Reuse function evaluations,
- (b) 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 ratio 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

- (a) Set $t = 0$
- (b) While $\|f(x^t)\|_2 > \varepsilon$:
 - i. Set $x^{t+1} \leftarrow x^t - \alpha \nabla f(x^t)$.
 - ii. Set $t \leftarrow t + 1$.
- (c) Return x^t .

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

- (a) Set $t = 0$
- (b) While $\|f(x^t)\|_2 > \varepsilon$:
 - i. Set $x^{t+1} \leftarrow x^t - \alpha [\nabla^2 f(x^t)]^{-1} \nabla f(x^t)$.
 - ii. Set $t \leftarrow t + 1$.
- (c) Return x^t .

19.3 Multi-Variate Unconstrained Optimizaiton

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

- (a) 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:

- (a) **Gradient Descent:** $d_t = -\nabla f(x_t)$
- (b) **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:

- (a) Set $t = 0$
- (b) While ... (some stopping criterion)
 - i. Choose i uniformly at random in $\{1, \dots, N\}$.
 - ii. Set $d_t = \nabla f_i(x_t)$
 - iii. 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 19.1: 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

$$\min_{a,b} \sum_{i=1}^N g_i(a,b) \quad (19.2)$$

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*

- (a) **Newton's Method:** $\Delta_k = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k)$
- (b) **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.

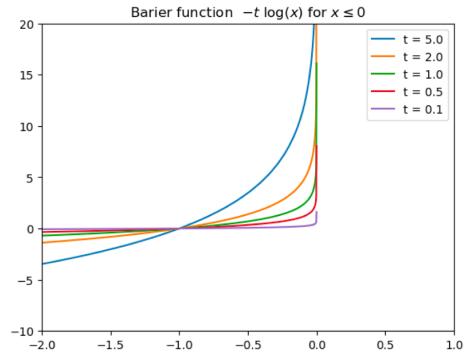
$$\begin{aligned} \min \quad & s \\ \text{s.t.} \quad & f_i(x) \leq s \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d, s \in \mathbb{R} \end{aligned} \tag{19.3}$$

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.

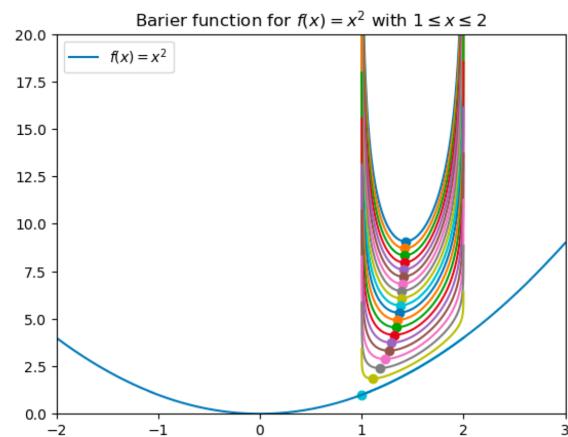
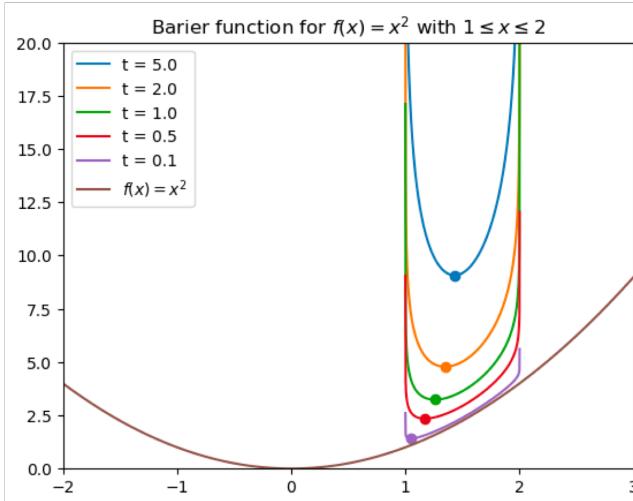
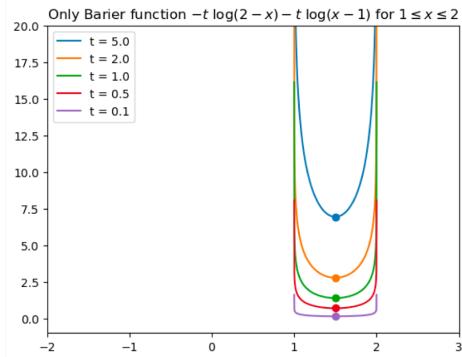
Now, convert this problem into the unconstrained minimization problem:

$$\begin{aligned} \min \quad & f(x) - \phi \sum_{i=1}^m \log(-(f_i(x) - s)) \\ & x \in \mathbb{R}^d, s \in \mathbb{R} \end{aligned} \tag{19.4}$$

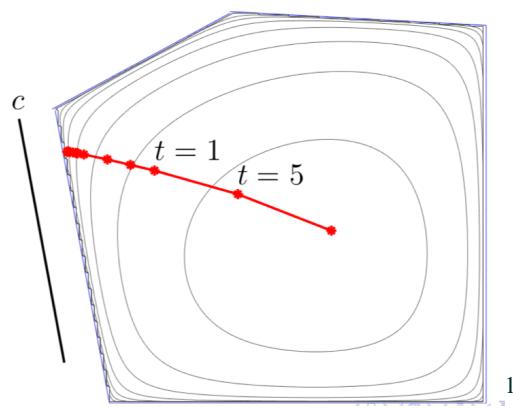
This problem has an easy time of finding an initial feasible solution. For instance, let $x = 0$, and then $s = \max_i f_i(x) + 1$.



Images below: the value t is the value ϕ discussed above



Minimizing $c^T x$ subject to $Ax \leq b$.



¹Image taken from unknown source.

20. Computational Issues with NLP

Chapter 20. Computational Issues with NLP

50% complete. Goal 80% completion date: November 20

Notes:

We mention a few computational issues to consider with nonlinear programs.

20.1 Irrational Solutions

Consider nonlinear problem (this is even convex)

$$\begin{array}{ll} \min & -x \\ \text{s.t.} & x^2 \leq 2. \end{array} \tag{20.1}$$

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)

$$\begin{array}{ll} \min & -x \\ \text{s.t.} & x^2 = 2. \end{array} \tag{20.1}$$

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 Karush-Huhn-Tucker (KKT) Conditions

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

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & g_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d \end{aligned} \tag{20.1}$$

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

(a) (Stationary).

$$-\nabla f(\bar{x}) = \sum_{i=1}^m \bar{\lambda}_i \nabla g_i(\bar{x}) \tag{20.2}$$

(b) (Complimentary Slackness).

$$\bar{\lambda}_i g_i(\bar{x}) = 0 \text{ for } i = 1, \dots, m \tag{20.3}$$

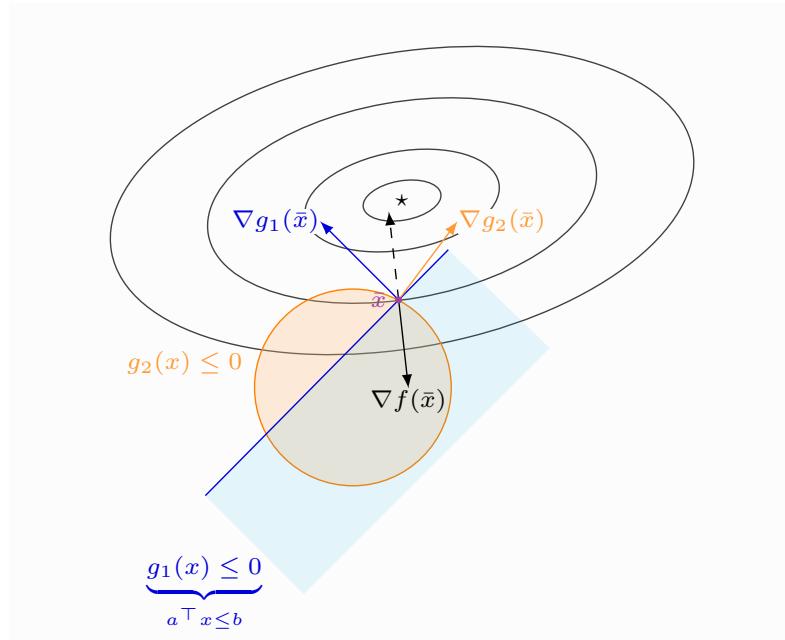
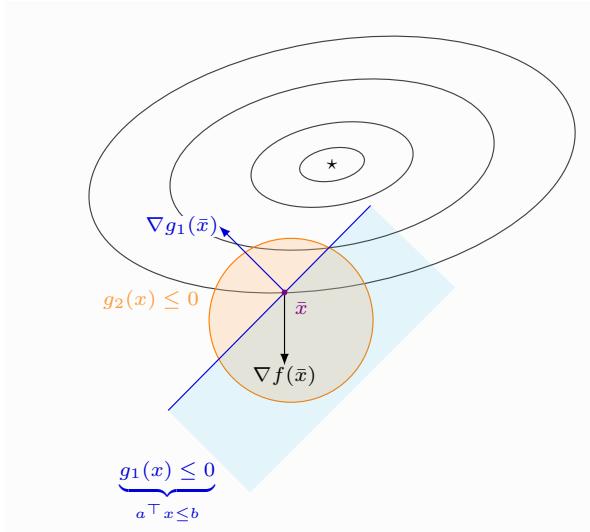
(c) (Primal Feasibility).

$$g_i(\bar{x}) \leq 0 \text{ for } i = 1, \dots, m \tag{20.4}$$

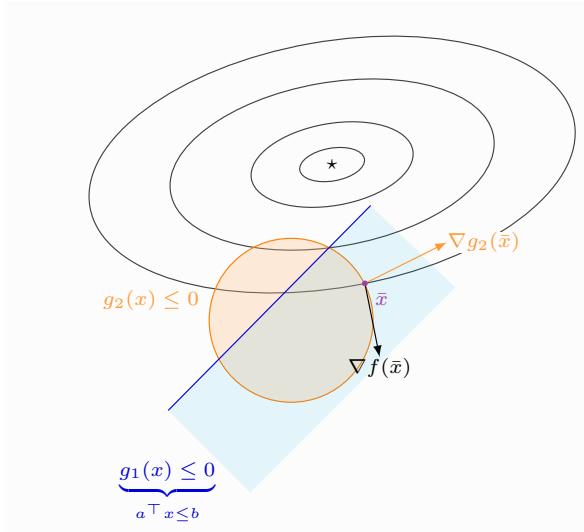
(d) (Dual Feasibility).

$$\bar{\lambda}_i \geq 0 \text{ for } i = 1, \dots, m \tag{20.5}$$

If certain properties are true of the convex program, then every optimizer has these properties. In particular, this holds for Linear Programming.

© tikz/kkt-optimal¹© tikz/kkt-non-optimal1²

tikz/kkt-non-optimal1, from tikz/kkt-non-optimal1.
tikz/kkt-non-optimal1, tikz/kkt-non-optimal1.

© tikz/kkt-non-optimal2³

tikz/kkt-non-optimal2, from tikz/kkt-non-optimal2.
tikz/kkt-non-optimal2, tikz/kkt-non-optimal2.

¹tikz/kkt-optimal, from tikz/kkt-optimal. tikz/kkt-optimal, tikz/kkt-optimal.

20.6 Gradient Free Algorithms

20.6.1. Nelder-Mead

Resources

- *Wikipedia*
- *Youtube*

21. Material to add...

Chapter 21. Material to add...

Decide if we want this material.

21.0.1. Bisection Method and Newton's Method

Resources

- See section 4 of the following nodes: <http://www.seas.ucla.edu/~vandenbe/133A/133A-notes.pdf>

21.1 Gradient Descent

Resources

Recap Gradient and Directional Derivatives:

- <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=PLZHQB0WTQDPD3MizzM2xVFitgF8hE_ab&index=2&t=0s

22. Fairness in Algorithms

Chapter 22. Fairness in Algorithms

Decide if we want to include this chapter or not. No material currently written for it.

Resources

- *Simons Institute - Michael Kearns (University of Pennsylvania) - "The Ethical Algorithm"*

23. One-dimensional Optimization

Chapter 23. One-dimensional Optimization

Todo: Adapt and incorporate this material.

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 23.1). 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 $x^* \in [\tilde{a}, \tilde{b}]$ 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 $x^* \in [\tilde{a}, \tilde{b}]$ 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.

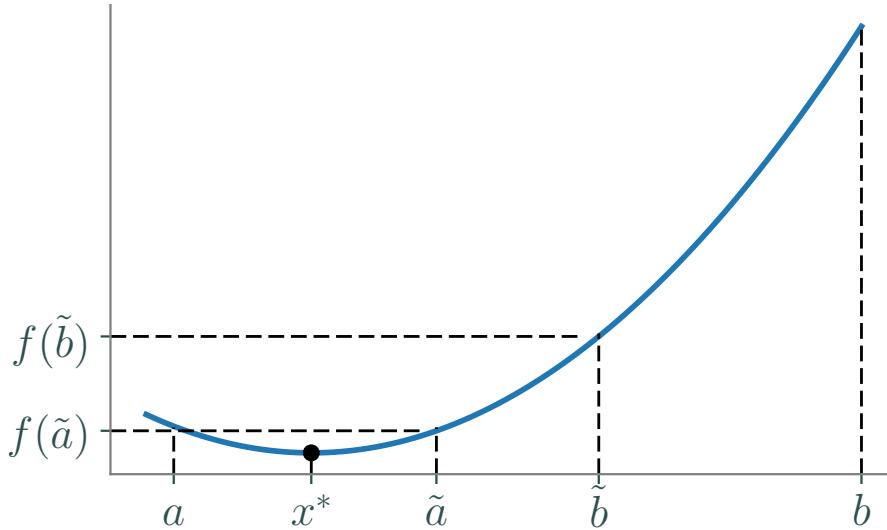


Figure 23.1: The unimodal $f : [a, b] \rightarrow \mathbb{R}$ can be minimized with a golden section search. For the first iteration, $f(\tilde{a}) < f(\tilde{b})$, so $x^* \in [\tilde{a}, \tilde{b}]$. New values of \tilde{a} and \tilde{b} are then calculated from this new, smaller interval.

Algorithm 7 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 23.1: 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 7 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

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{23.1}$$

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 (23.1) 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 (23.1) using x_{k+1} for x . See Figure 23.2.

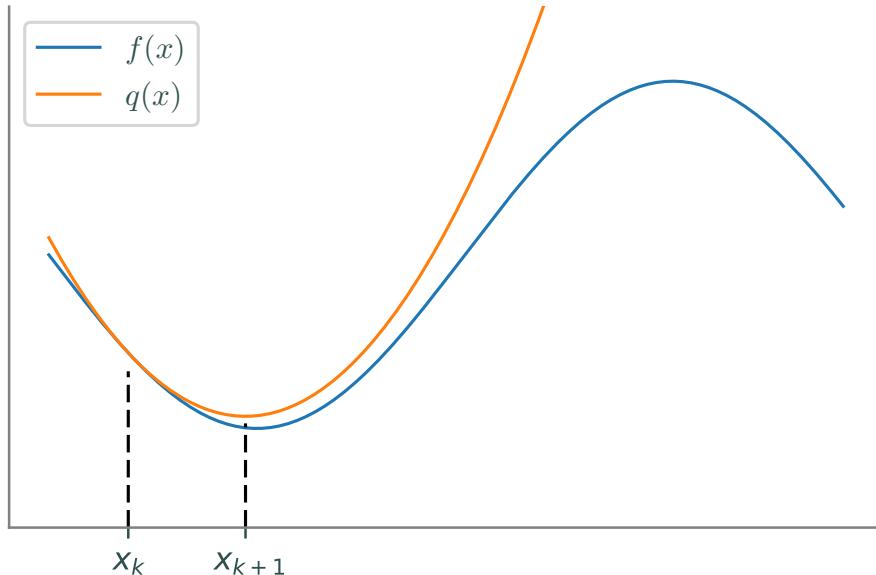


Figure 23.2: A quadratic approximation of f at x_k . The minimizer x_{k+1} of q is close to the minimizer of f .

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 23.2: 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 (23.1) 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

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.

$$f''(x_k) \approx \frac{f'(x_k + x_{k-1} - x_k) - f'(x_k)}{x_{k-1} - x_k} = \frac{f(x_k) - f'(x_{k-1})}{x_k - x_{k-1}} \quad (23.2)$$

Inserting (23.2) into (23.1) results in the complete secant method formula.

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} f'(x_k) = \frac{x_{k-1}f'(x_k) - x_kf'(x_{k-1})}{f'(x_k) - f'(x_{k-1})} \quad (23.3)$$

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 23.3: 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 (23.3) 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.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (23.4)$$

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 (23.2) 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 23.3.

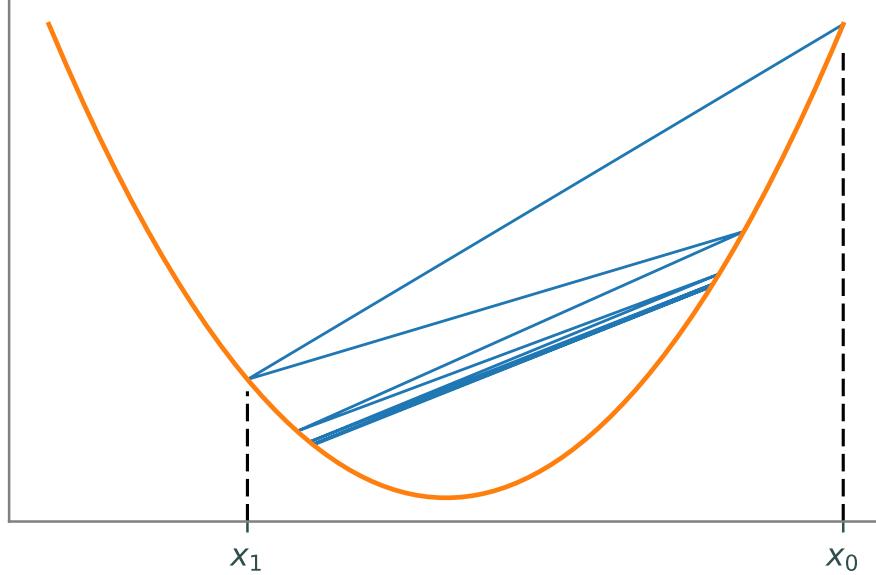


Figure 23.3: If the step size α_k is too large, a descent method may repeatedly overstep the minimizer.

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*:

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k \quad (23.5)$$

$$-Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k \leq -c_2 Df(\mathbf{x}_k)^T \mathbf{p}_k \quad (23.6)$$

where $0 < c_1 < c_2 < 1$ (for the best results, choose $c_1 \ll c_2$). The condition (23.5) 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,

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) = f(\mathbf{x}_k) + \alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k + \mathcal{O}(\alpha_k^2).$$

Thus, a very small α_k will always satisfy (23.5) since $Df(\mathbf{x}_k)^T \mathbf{p}_k < 0$ (as \mathbf{p}_k is a descent direction). The condition (23.6), 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 (23.6) to ensure that α_k is near the minimizer.

$$|Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k| \leq c_2 |Df(\mathbf{x}_k)^T \mathbf{p}_k|$$

The *Armijo–Goldstein conditions* provide another alternative to (23.6):

$$f(\mathbf{x}_k) + (1 - c)\alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k \leq f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c\alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k,$$

where $0 < c < 1$. These conditions are very similar to the Wolfe conditions (the right inequality is (23.5)), but they do not require the calculation of the directional derivative $Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k$.

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 (23.5). This is usually sufficient, but if it finds α 's that are too small, the algorithm can be modified to satisfy (23.6) or one of its variants.

Algorithm 8 Backtracking using the Armijo Rule

```

1: procedure BACKTRACKING( $f, Df, \mathbf{x}_k, \mathbf{p}_k, \alpha, \rho, c$ )
2:    $Dfp \leftarrow Df(\mathbf{x}_k)^T \mathbf{p}_k$                                  $\triangleright$  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 23.4: 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 8. 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 anp
>>> 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 = anp.array([150., .03, 40.])           # Current minimizer ←
    guesss.
>>> p = anp.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.))
```


24. Gradient Descent Methods

Chapter 24. Gradient Descent Methods

Todo: Adapt and incorporate this material.

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 .

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (24.1)$$

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 (24.1), resulting in the following algorithm.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\top \quad (24.2)$$

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 (24.2) 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 24.1: 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.

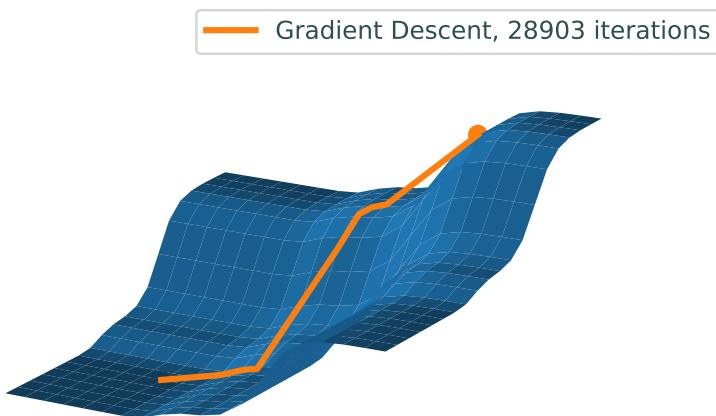


Figure 24.1: On this surface, gradient descent takes an extreme number of iterations to converge to the minimum because it gets stuck in the flat basins of the surface.

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 $\mathbf{x}_i^\top Q \mathbf{x}_j = 0$. 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

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top Q\mathbf{x} - \mathbf{b}^\top \mathbf{x} + c.$$

Because $Df(\mathbf{x})^\top = Q\mathbf{x} - \mathbf{b}$, minimizing f is the same as solving the equation

$$0 = Df(\mathbf{x})^\top = Q\mathbf{x} - \mathbf{b} \Rightarrow Q\mathbf{x} = \mathbf{b},$$

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 24.2. 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.

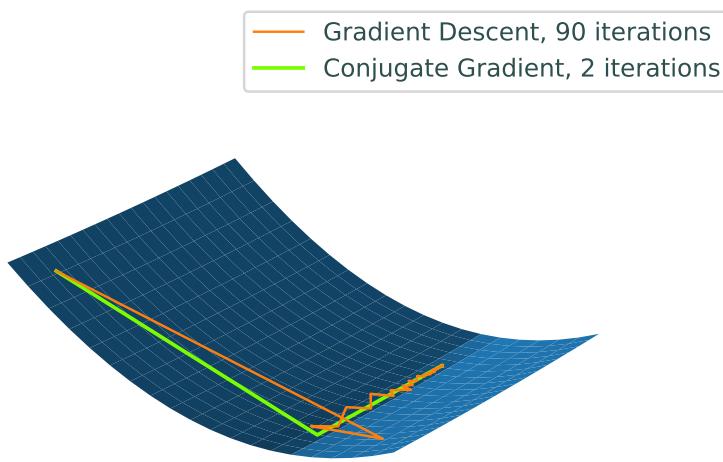


Figure 24.2: Paths traced by Gradient Descent (orange) and Conjugate Gradient (red) on a quadratic surface. Notice the zig-zagging nature of the Gradient Descent path, as opposed to the Conjugate Gradient path, which finds the minimizer in 2 steps.

Algorithm 9

```

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 24.2: 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 9 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 24 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 $\beta_k = Df(\mathbf{x}_k)Df(\mathbf{x}_k)^T/Df(\mathbf{x}_{k-1})Df(\mathbf{x}_{k-1})^T$.

Algorithm 10

```

1: procedure NON-LINEAR CONJUGATE GRADIENT( $f, Df, \mathbf{x}_0, \text{tol}, \text{maxiter}$ )
2:    $\mathbf{r}_0 \leftarrow -Df(\mathbf{x}_0)^T$ 
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:      $\mathbf{r}_k \leftarrow -Df(\mathbf{x}_k)^T$ 
9:      $\beta_k = \mathbf{r}_k^T \mathbf{r}_k / \mathbf{r}_{k-1}^T \mathbf{r}_{k-1}$ 
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 24.3

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 10 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])

```

Regression Problems

A major use of the conjugate gradient method is solving linear least squares problems. Recall that a least squares problem can be formulated as an optimization problem:

$$\mathbf{x}^* = \min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2,$$

where A is an $m \times n$ matrix with full column rank, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{b} \in \mathbb{R}^m$. The solution can be calculated analytically, and is given by

$$\mathbf{x}^* = (A^\top A)^{-1} A^\top \mathbf{b}.$$

In other words, the minimizer solves the linear system

$$A^\top A \mathbf{x} = A^\top \mathbf{b}. \quad (24.3)$$

Since A has full column rank, it is invertible, $A^\top A$ is positive definite, and for any non-zero vector \mathbf{z} , $\mathbf{z}^\top A^\top A \mathbf{z} = \|\mathbf{Az}\|^2 > 0$. As $A^\top A$ is positive definite, conjugate gradient can be used to solve Equation 24.3.

Linear least squares is the mathematical underpinning of *linear regression*. Linear regression involves a set of real-valued data points $\{y_1, \dots, y_m\}$, where each y_i is paired with a corresponding set of predictor variables $\{x_{i,1}, x_{i,2}, \dots, x_{i,n}\}$ with $n < m$. The linear regression model posits that

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_n x_{i,n} + \varepsilon_i$$

for $i = 1, 2, \dots, m$. The real numbers β_0, \dots, β_n are known as the parameters of the model, and the ε_i are independent, normally-distributed error terms. The goal of linear regression is to calculate the parameters that best fit the data. This can be accomplished by posing the problem in terms of linear least squares. Define

$$\mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad A = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}.$$

The solution $\mathbf{x}^* = [\beta_0^*, \beta_1^*, \dots, \beta_n^*]^\top$ to the system $A^\top A \mathbf{x} = A^\top \mathbf{b}$ gives the parameters that best fit the data. These values can be understood as defining the hyperplane that best fits the data.

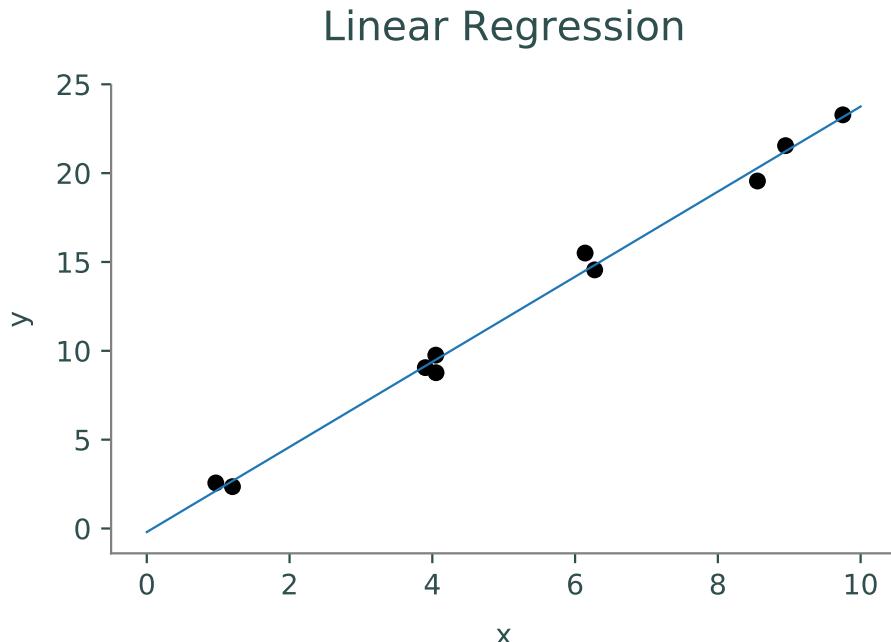


Figure 24.3: Solving the linear regression problem results in a best-fit hyperplane.

Problem 24.4

Using your function from Problem 24, solve the linear regression problem specified by the data contained in the file^a `linregression.txt`. This is a whitespace-delimited text file formatted so that the i -th row consists of $y_i, x_{i,1}, \dots, x_{i,n}$. Use `np.loadtxt()` to load in the data and return the solution to the normal equations.

^aSource: Statistical Reference Datasets website at <http://www.itl.nist.gov/div898/strd/lls/data/LINKS/v-Longley.shtml>.

Logistic Regression

Logistic regression is another important technique in statistical analysis and machine learning that builds off of the concepts of linear regression. As in linear regression, there is a set of predictor variables $\{x_{i,1}, x_{i,2}, \dots, x_{i,n}\}_{i=1}^m$ with corresponding outcome variables $\{y_i\}_{i=1}^m$. In logistic regression, the outcome variables y_i are binary and can be modeled by a *sigmoidal* relationship. The value of the predicted y_i can be thought of as the probability that $y_i = 1$. In mathematical terms,

$$\mathbb{P}(y_i = 1 | x_{i,1}, \dots, x_{i,n}) = p_i,$$

where

$$p_i = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n}))}.$$

The parameters of the model are the real numbers $\beta_0, \beta_1, \dots, \beta_n$. Note that $p_i \in (0, 1)$ regardless of the values of the predictor variables and parameters.

The probability of observing the outcome variables y_i under this model, assuming they are independent, is given by the *likelihood function* $\mathcal{L} : \mathbb{R}^{n+1} \rightarrow \mathbb{R}$

$$\mathcal{L}(\beta_0, \dots, \beta_n) = \prod_{i=1}^m p_i^{y_i} (1 - p_i)^{1-y_i}.$$

The goal of logistic regression is to find the parameters β_0, \dots, β_k that maximize this likelihood function. Thus, the problem can be written as:

$$\max_{(\beta_0, \dots, \beta_n)} \mathcal{L}(\beta_0, \dots, \beta_n).$$

Maximizing this function is often a numerically unstable calculation. Thus, to make the objective function more suitable, the logarithm of the objective function may be maximized because the logarithmic function is strictly monotone increasing. Taking the log and turning the problem into a minimization problem, the final problem is formulated as:

$$\min_{(\beta_0, \dots, \beta_n)} -\log \mathcal{L}(\beta_0, \dots, \beta_n).$$

A few lines of calculation reveal that this objective function can also be rewritten as

$$\begin{aligned} -\log \mathcal{L}(\beta_0, \dots, \beta_n) &= \sum_{i=1}^m \log(1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n}))) + \\ &\quad \sum_{i=1}^m (1 - y_i)(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n}). \end{aligned}$$

The values for the parameters $\{\beta_i\}_{i=1}^n$ that we obtain are known as the *maximum likelihood estimate* (MLE). To find the MLE, conjugate gradient can be used to minimize the objective function.

For a one-dimensional binary logistic regression problem, we have predictor data $\{x_i\}_{i=1}^m$ with labels $\{y_i\}_{i=1}^m$ where each $y_i \in \{0, 1\}$. The negative log likelihood then becomes the following.

$$-\log \mathcal{L}(\beta_0, \beta_1) = \sum_{i=1}^m \log(1 + e^{-(\beta_0 + \beta_1 x_i)}) + (1 - y_i)(\beta_0 + \beta_1 x_i) \tag{24.4}$$

Problem 24.5

Write a class for doing binary logistic regression in one dimension that implement the following methods.

- (a) *fit()*: accept an array $\mathbf{x} \in \mathbb{R}^n$ of data, an array $\mathbf{y} \in \mathbb{R}^n$ of labels (0s and 1s), and an initial guess $\beta_0 \in \mathbb{R}^2$. Define the negative log likelihood function as given in (24.4), then minimize it (with respect to β) with your function from Problem 24 or `opt.fmin_cg()`.

Store the resulting parameters β_0 and β_1 as attributes.

- (b) *predict()*: accept a float $x \in \mathbb{R}$ and calculate

$$\sigma(x) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x))},$$

where β_0 and β_1 are the optimal values calculated in *fit()*. The value $\sigma(x)$ is the probability that the observation x should be assigned the label $y = 1$.

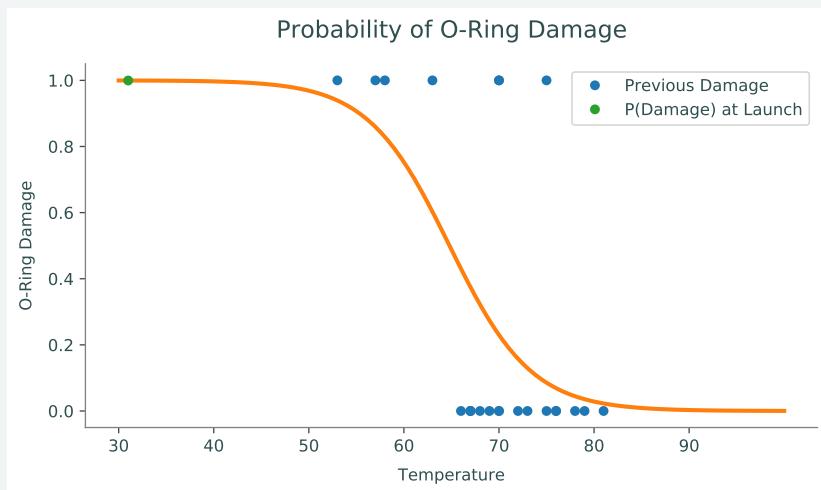
This class does not need an explicit constructor. You may assume that *predict()* will be called after *fit()*.

Problem 24.6

On January 28, 1986, less than two minutes into the Challenger space shuttle's 10th mission, there was a large explosion that originated from the spacecraft, killing all seven crew members and destroying the shuttle. The investigation that followed concluded that the malfunction was caused by damage to O-rings that are used as seals for parts of the rocket engines. There were 24 space shuttle missions before this disaster, some of which had noted some O-ring damage. Given the data, could this disaster have been predicted?

The file `challenger.npy` contains data for 23 missions (during one of the 24 missions, the engine was lost at sea). The first column (x) contains the ambient temperature, in Fahrenheit, of the shuttle launch. The second column (y) contains a binary indicator of the presence of O-ring damage (1 if O-ring damage was present, 0 otherwise).

Instantiate your class from Problem 24 and fit it to the data, using an initial guess of $\beta_0 = [20, -1]^T$. Plot the resulting curve $\sigma(x)$ for $x \in [30, 100]$, along with the raw data. Return the predicted probability (according to this model) of O-ring damage on the day the shuttle was launched, given that it was $31^\circ F$.



A. Linear Algebra

Chapter 1. Linear Algebra

Decide which material to add here.