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: October 24, 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 Mathmatics" 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 LaTeXusing 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

Todo list

Major things to do: - Re-evaluate each section and timeline for completion.,Work on Section 3 - Write a sensitivity analysis section - Create code and link it to examples - Build out the Excel and Python sections - Decide what Linear Algebra Appendices to add **Minor things to do:** - Clean up preambles - Standardize LP formatting - Clean up metadata/references for all figures - Fix Compile errors - Add unnumbered "Resources and References" section to end of each chapter. **Things completed:** - Make slides template

1. Introduction

This document will reflect somewhat the lectures given in ISE 6414 in Fall 2022.

2. Resources and Notation

Chapter 2. 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 Optimizaiton Notes by Komei Fukuda
- Convex Optimization by Bord 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 PolerJosefa 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 Orielly.com.
- Lesson files for the Winston Book
- Video instructions for solver and an example workbook
- youtube-OR-course

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

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

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

A sample sentence:

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

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

3. Mathematical Programming

Chapter 3. Mathematical Programming
50% complete. Goal 80% completion date: August 20
Notes: This chapter is meant to be an introduction to all the types of deterministic problems that we might discuss. It should list many applications, have a number of pictures, and describe how and where these types of problems are used.

Add intro that explains the format of problems, i.e., what the complexity comment means in each problem and add pointer to section on computational complexity.

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.

Outcomes

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

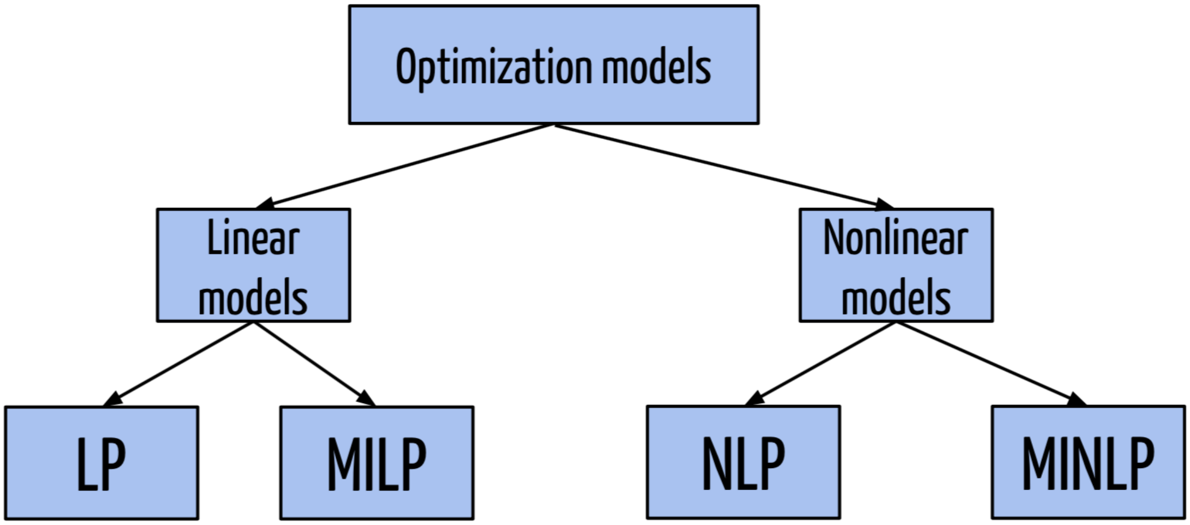
Along with each problem class, we will associate a complexity class for the general version of the problem. See section 17 for a discussion of complexity classes. Although we will often state that input data for a problem comes from $\mathbb{R}$, when we discuss complexity of such a problem, we actually mean that the data is rational, i.e., from $\mathbb{Q}$, and is given in binary encoding.

3.1 Linear Programming (LP)

Describe applications and andd images

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

[1] **problem-class-diagram**, from **problem-class-diagram**. **problem-class-diagram**, **problem-class-diagram**.



© problem-class-diagram[1]

Figure 3.1: problem-class-diagram

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

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{3.1}$$

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

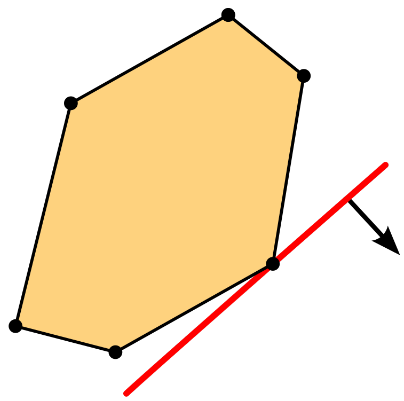
Linear Programming (LP) Standard Form:

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

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{3.2}$$

Can shrink figure if there are not a lot of numbers of greek letters involved.



© wiki/File/linear-programming.png[2]

Figure 3.2: Linear programming constraints and objective.

Figure 3.2

Move this to simplex chapter

Exercise 3.1:

Start with a problem in form given as (3.1) and convert it to standard form (3.2) by adding at most m many new variables and by enlarging the constraint matrix A by at most m new columns.

3.2 Mixed-Integer Linear Programming (MILP)

Mixed-integer linear programming will be the focus of Sections 16,19, 20, 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):

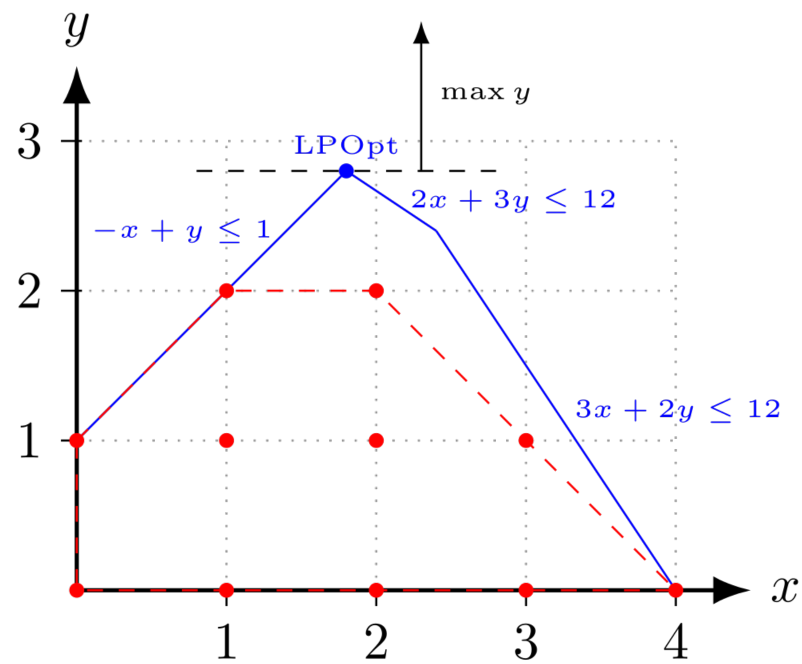
NP-Complete

[2]**wiki/File/linear-programming.png**, from **wiki/File/linear-programming.png**. **wiki/File/linear-programming.png**, **wiki/File/linear-programming.png**.

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

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

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



© **wiki/File/integer-programming.png**[3]

Figure 3.3: Comparing the LP relaxation to the IP solutions.

Figure 3.3

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

[3]**wiki/File/integer-programming.png**, from **wiki/File/integer-programming.png**. **wiki/File/integer-programming.png**, **wiki/File/integer-programming.png**.

is

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

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

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{Z}^n \times \mathbb{R}^d \end{aligned} \tag{3.3}$$

3.3 Non-Linear Programming (NLP)

NLP:

NP-Hard

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

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d \end{aligned} \tag{3.1}$$

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.

3.3.1. Convex Programming

Here the functions are all **convex!**

Convex Programming:

Polynomial time (P) (typically)

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

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1,\dots,m \\ & x \in \mathbb{R}^d \end{aligned} \tag{3.2}$$

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

3.3.2. Non-Convex Non-linear Programming

Move this to later chapter on complexity of NLP

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

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & \cancel{x \in \{0,1\}^n} \\ & x_i(1 - x_i) = 0 \quad \text{for } i = 1, \dots, n \end{aligned} \tag{3.3}$$

3.3.3. Machine Learning

Many machine learning problems fall into this catrgory. Todo: describe applications, give references, etc.

3.4 Mixed-Integer Non-Linear Programming (MINLP)

Fill in this section with formulas and discuss applications. Most notable applications are Electrical Grid problems and Pooling problems. Find applications at Optimization and Engineering `https://link.springer.com/journal/11081/volumes-and-issues/21-4`

3.4.1. Convex Mixed-Integer Non-Linear Programming

3.4.2. 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

3.5 The Simplex Method

The Simplex Method

Section 3.5. The Simplex Method
Integrate this to next chapter (chapter on 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.*

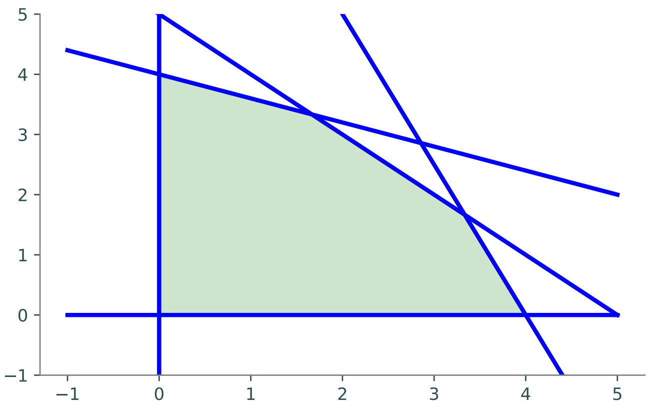
Standard Form

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

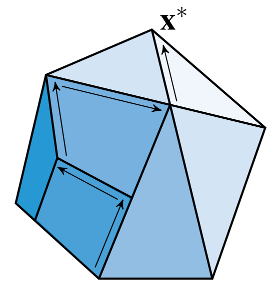
$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^{\mathsf{T}}\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

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 9.1). The Simplex Algorithm jumps among the vertices of the feasible region searching for an optimal point. It does this by moving along the edges of the feasible region in such a way that the objective function is always increased after each move.



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



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

Figure 3.4: If an optimal point exists, it is one of the vertices of the polyhedron. The simplex algorithm searches for optimal points by moving between adjacent vertices in a direction that increases the value of the objective function until it finds an optimal vertex.

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

The Simplex Solver

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

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

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

Accepting a Linear Program

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

Problem 3.2: 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* $A\mathbf{x} \preceq \mathbf{b}$ *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 ith 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

$$\bar{A} = \begin{bmatrix} A & I_m \end{bmatrix},$$

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

$$D = \begin{bmatrix} 0 & \bar{\mathbf{c}}^\mathsf{T} \\ \mathbf{b} & -\bar{A} \end{bmatrix} \tag{3.1}$$

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 3.3: 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.

3.5.1. 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:

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

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 jth column of the dictionary and that the negative entries of this column are $D_{i_1,j}, D_{i_2,j}, \ldots, D_{i_k,j}$, we calculate the ratios

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

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 a 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

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

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 3.4: Bland's Rule

hoose 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:

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

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

Problem 3.5: Pivoting

dd 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`*.*

3.5.2. 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 ith entry of the index list has the value $T_{i+1,0}$.

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

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

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

$$x_0 = 1.6$$
$$x_1 = .2.$$

Problem 3.6: SimplexSolver.solve()

rite 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.

3.5.3. Exercises

EXERCISE 1.0 (LEARN LT$_{\text{E}}$X) Learn to use lt$_{\text{E}}X$ 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 pdfteTEX (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

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

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

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

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 is 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.

3.5.4. 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 $1,2,\ldots,m$ with availabilities b_i and per unit costs c_i. We are given a set of products $j,2,\ldots,m$ 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 $1,2,\ldots,n$ 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 $t = 1,2,\ldots,T$. 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 investmentvehicle k at the start of period t must be held in that investment k for all remaining time periods, and it generates income $v^k_{t,t}, v^k_{t,t+1}, \ldots, v^k_{tT}$, 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 $v^k_{t,T}$. 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.

4. Simplex Method

Chapter 4. Simplex Method
10% complete. Goal 80% completion date: January 20, 2023
Notes: This section hasn't been cleaned at all. This needs to be looked at and cleaned up.

Definition 4.1: Standard Form

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

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

Definition 4.2: 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 4.3: 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]

♠

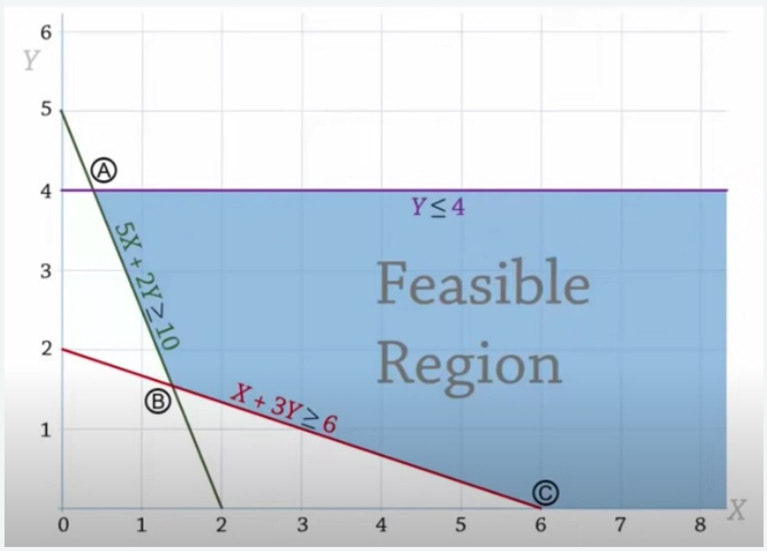
Definition 4.4: Basic solution

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 4.5

Consider the problem

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



We begin by converting this problem to standard form.

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

Thus, we can write this problem in matrix form with

$$\max \begin{bmatrix} -5 \\ -7 \\ 0 \\ 0 \\ 0 \end{bmatrix}^\top \begin{bmatrix} X \\ Y \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \tag{4.1}$$

$$\begin{bmatrix} 1 & 3 & -1 & 0 & 0 \\ 5 & 2 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 10 \\ 4 \end{bmatrix} \tag{4.2}$$

$$(X, Y, s_1, s_2, s_3) \geq 0 \tag{4.3}$$

Definition 4.6: Basic feasible solution

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

Theorem 4.7: 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 4.8: Representation

Consider an LP in standard form, having bfs $b_1, \ldots, b_k$. 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 4.9: Optimal bfs

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

Proof. Let x be an optimal solution. Then

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

where d is 0 or a direction of unboundeness.

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

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

Since

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

for all $i = 1, \ldots, k$, 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. ♠

4.1 The Simplex Method

The Simplex Method

Section 4.1. The Simplex Method
Integrate this to next chapter (chapter on 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.*

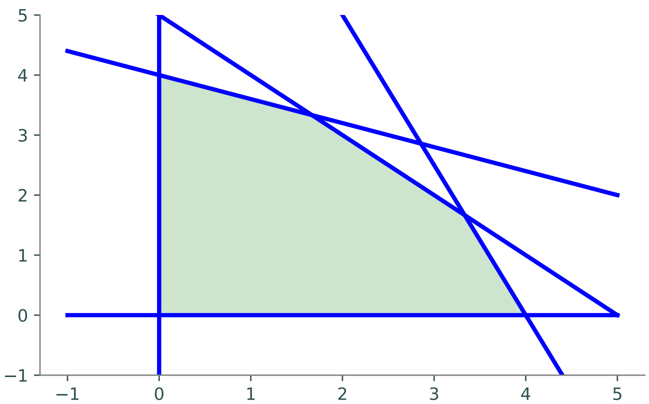
Standard Form

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

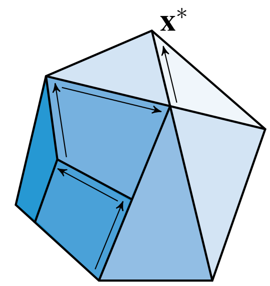
$$\begin{array}{ll} \text{minimize} & \mathbf{c}^\mathsf{T}\mathbf{x} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{array}$$

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 9.1). The Simplex Algorithm jumps among the vertices of the feasible region searching for an optimal point. It does this by moving along the edges of the feasible region in such a way that the objective function is always increased after each move.



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



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

Figure 4.1: If an optimal point exists, it is one of the vertices of the polyhedron. The simplex algorithm searches for optimal points by moving between adjacent vertices in a direction that increases the value of the objective function until it finds an optimal vertex.

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

The Simplex Solver

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

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

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

Accepting a Linear Program

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

Problem 4.10: Check feasibility at the origin.

Write a class that accepts the arrays $\mathbf{c}$, A, *and* $\mathbf{b}$ *of a linear optimization problem in standard form. In the constructor, check that the system is feasible at the origin. That is, check that* $A\mathbf{x} \preceq \mathbf{b}$ *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 ith 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

$$\bar{A} = \begin{bmatrix} A & I_m \end{bmatrix},$$

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

$$D = \begin{bmatrix} 0 & \bar{\mathbf{c}}^{\mathsf{T}} \\ \mathbf{b} & -\bar{A} \end{bmatrix} \tag{4.1}$$

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 4.11: Initialize the dictionary.

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

4.1.1. Pivoting

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

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

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

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

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

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 jth column of the dictionary and that the negative entries of this column are $D_{i_1,j}, D_{i_2,j}, \ldots, D_{i_k,j}$, we calculate the ratios

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

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 a 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

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

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 4.12: Bland's Rule

hoose 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:

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

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

Problem 4.13: Pivoting

dd 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`.

4.1.2. Termination and Reading the Dictionary

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

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

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

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

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

$$x_0 = 1.6$$
$$x_1 = .2.$$

Problem 4.14: SimplexSolver.solve()

rite 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.

4.1.3. Exercises

EXERCISE 1.0 (LEARN LT$_{\text{E}}$X) Learn to use lt$_{\text{E}}X$ 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 pdfteTEX (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

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

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

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

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 is 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.

4.1.4. 2.5 Exercises

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

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

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

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

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

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

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

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

We are given a set of ingredients $1,2,\ldots,m$ with availabilities b_i and per unit costs c_i. We are given a set of products $j,2,\ldots,m$ 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 $1,2,\ldots,n$ 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 $t = 1,2,\ldots,T$. 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 investmentvehicle k at the start of period t must be held in that investment k for all remaining time periods, and it generates income $v^k_{t,t}, v^k_{t,t+1}, \ldots, v^k_{tT}$, 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 $v^k_{t,T}$. 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.

4.2 Finding Feasible Basis

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

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

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

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

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

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.

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

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

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

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 leaves 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:

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

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

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

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

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

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

z	x_1	x_2	s_1	s_2	RHS
1	20	0	0	15	300
0	-2	0	1	-2	0
0	2	1	0	1/2	10

This is an optimal tableau, with an objective function value of 300, If instead of x_1 leaving the basis, suppose s_1 left, this would lead to the following tableau (BV=$\{x_2, x_1\}$).

z	x_1	x_2	s_1	s_2	RHS
1	0	0	10	-5	300
0	0	1	1	-3/2	10
0	1	0	-1/2	1	0

This tableau does not look optimal, yet the objective function value is the same as the optimal solution's. This occurs because the optimal extreme point is a degenerate.

5. Duality

Chapter 5. Duality
0% complete. Goal 80% completion date: January 20, 2023
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:

$$\begin{array}{rll} & \text{maximize } 4x_1 + x_2 + 3x_3 & \\ \text{subject to} & x_1 + 4x_2 \le 2 & \\ & 3x_1 - x_2 + x_3 & \le 4 \\ & x_1, x_2, x_3 & \ge 0 \end{array}$$

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 :

$$\begin{aligned} &\text{maximize } 4x_1 + x_2 + 3x_3 \\ &\text{subject to } y_1 (x_1 + 4x_2 \quad \le 2y_1 \\ &\qquad y_2 (3x_1 - x_2 + x_3) \le 4y_2 \\ &\qquad\qquad x_1, x_2, x_3 \ge 0 \end{aligned}$$

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:

$$(y_1 + 3y_2) x_1 + (4y_1 - y_2) x_2 + y_2 x_3 \le 2y_1 + 4y_2.$$

Now suppose that each y_i is larger than the i th coefficient of the objective function:

$$y_1 + 3y_2 \ge 4, \quad 4y_1 - y_2 \ge 1, \quad y_2 \ge 3.$$

This assumption lets us derive an upper bound on the objective value of any feasible solution:

$$4x_1 + x_2 + 3x_3 \le (y_1 + 3y_2) x_1 + (4y_1 - y_2) x_2 + y_2 x_3 \le 2y_1 + 4y_2.$$

In particular, by plugging in the optimal solution $\left(x_1^*, x_2^*, x_3^*\right)$ for the original LP, we obtain the following upper bound on σ^* :

$$\sigma^* = 4x_1^* + x_2^* + 3x_3^* \le 2y_1 + 4y_2.$$

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.

$$\begin{array}{rr} \text{minimize} & 2y_1 + 4y_2 \\ \text{subject to} & y_1 + 3y_2 \geq 4 \\ & 4y_1 - y_2 \geq 1 \\ y_2 & \geq 3 \\ y_1, y_2 & \geq 0 \end{array}$$

"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.

5.1 The Dual of Linear Program

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

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

and that an LP-solver has found for us the solution $x_1 := 1, x_2 := \frac{1}{2}, x_3 := 0, x_4 := \frac{1}{2}$ of cost 2.5. How can we convince ourselves, or another user, that the solution is indeed optimal, without having to trace the steps of the computation of the algorithm?

Observe that if we have two valid inequalities

$$a \leq b \text{ and } c \leq d$$

then we can deduce that the inequality

$$a + c \leq b + d$$

(derived by "summing the left hand sides and the right hand sides" of our original inequalities) is also true. In fact, we can also scale the inequalities by a positive multiplicative factor before adding them up, so for every non-negative values $y_1, y_2 \geq 0$ we also have

$$y_1 a + y_2 c \leq y_1 b + y_2 d$$

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.

$$\begin{array}{ll} \text{maximize} & c_1 x_1 + \ldots c_n x_n \\ \text{subject to} & \\ & a_{1,1} x_1 + \ldots + a_{1,n} x_n \leq b_1 \\ & \vdots \\ & a_{m,1} x_1 + \ldots + a_{m,n} x_n \leq b_m \\ & x_1 \geq 0 \\ & \vdots \\ & x_n \geq 0 \end{array}$$

For every choice of non-negative scaling factors $y_1, \ldots, y_m$, we can derive the inequality

$$\begin{array}{c} y_1 \cdot (a_{1,1} x_1 + \ldots + a_{1,n} x_n) \\ + \cdots \\ + y_n \cdot (a_{m,1} x_1 + \ldots + a_{m,n} x_n) \\ \leq y_1 b_1 + \cdots y_m b_m \end{array}$$

which is true for every feasible solution $(x_1, \ldots, x_n)$ to the linear program (2). We can rewrite the inequality as

$$\begin{aligned}
&(a_{1,1}y_1 + \cdots a_{m,1}y_m) \cdot x_1 \\
&\quad + \cdots \\
\\
&+ (a_{1,n}y_1 \cdots a_{m,n}y_m) \cdot x_n \\
&\leq y_1 b_1 + \cdots y_m b_m
\end{aligned}$$

So we get that a certain linear function of the x_i is always at most a certain value, for every feasible $(x_1, \ldots, x_n)$. 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 $(x_1, \ldots, x_n)$. 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 $(y_1, \ldots, y_m)$ that satisfies (3), and for every $(x_1, \ldots, x_n)$ that is feasible for (2),

$$\begin{aligned}
&c_1x_1 + \ldots c_nx_n \\
&\leq (a_{1,1}y_1 + \cdots a_{m,1}y_m) \cdot x_1 \\
&\quad + \cdots \\
&+ (a_{1,n}y_1 \cdots a_{m,n}y_m) \cdot x_n \\
&\leq y_1 b_1 + \cdots y_m b_m
\end{aligned}$$

Clearly, we want to find the non-negative values $y_1, \ldots, y_m$ such that the above upper bound is as strong as possible, that is we want to

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

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

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

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

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

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

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

in an equivalent way as

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

If we compute the dual of the above program we get

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

that is,

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

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

$$\operatorname{opt}(LP_1) = \operatorname{opt}(LP_2)$$

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.

5.2 Linear programming duality

Consider the following problem:

$$\begin{array}{rl} \min & \mathbf{c}^\mathsf{T}\mathbf{x} \\ \text{s.t.} & \mathbf{A}\mathbf{x} \geq \mathbf{b}. \end{array} \tag{5.1}$$

In the remark at the end of Chapter **??**, we saw that if (11.1) has an optimal solution, then there exists $\mathbf{y}^* \in \mathbb{R}^m$ such that $\mathbf{y}^* \geq 0$, $\mathbf{y}^{*\mathsf{T}}\mathbf{A} = \mathbf{c}^\mathsf{T}$, and $\mathbf{y}^{*\mathsf{T}}\mathbf{b} = \gamma$ where γ denotes the optimal value of (11.1).

Take any $\mathbf{y} \in \mathbb{R}^m$ satisfying $\mathbf{y} \geq 0$ and $\mathbf{y}^\mathsf{T}\mathbf{A} = \mathbf{c}^\mathsf{T}$. Then we can infer from $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ the inequality $\mathbf{y}^\mathsf{T}\mathbf{A}\mathbf{x} \geq \mathbf{y}^\mathsf{T}\mathbf{b}$, or more simply, $\mathbf{c}^\mathsf{T}\mathbf{x} \geq \mathbf{y}^\mathsf{T}\mathbf{b}$. Thus, for any such $\mathbf{y}$, $\mathbf{y}^\mathsf{T}\mathbf{b}$ gives a lower bound for the objective function value of any feasible solution to (11.1). Since γ is the optimal value of (P), we must have $\gamma \geq \mathbf{y}^\mathsf{T}\mathbf{b}$.

As $\mathbf{y}^{*\mathsf{T}}\mathbf{b} = \gamma$, we see that γ is the optimal value of

$$\begin{array}{rl} \max & \mathbf{y}^\mathsf{T}\mathbf{b} \\ \text{s.t.} & \mathbf{y}^\mathsf{T}\mathbf{A} = \mathbf{c}^\mathsf{T} \\ & \mathbf{y} \geq 0. \end{array} \tag{5.2}$$

Note that (11.2) is a linear programming problem! We call it the **dual problem** of the **primal problem** (11.1). We say that the dual variable y_i is **associated** with the constraint $\mathbf{a}^{(i)\mathsf{T}}\mathbf{x} \geq b_i$ where $\mathbf{a}^{(i)\mathsf{T}}$ denotes the ith row of $\mathbf{A}$.

In other words, we define the dual problem of (11.1) to be the linear programming problem (11.2). 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 5.1: strong-duality-special

Suppose that (11.1) *has an optimal solution. Then* (11.2) *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 (11.1) 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:

$$\begin{array}{rl} \min & \mathbf{c}^\mathsf{T}\mathbf{x} \\ \text{s.t.} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{A}'\mathbf{x} \leq \mathbf{b}' \\ & \mathbf{A}''\mathbf{x} = \mathbf{b}'' \end{array} \tag{5.3}$$

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 (11.1) 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 (11.3) an inequality of the form $\mathbf{c}^\mathsf{T}\mathbf{x} \geq \gamma$ 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

$$\mathbf{y}^\mathsf{T}\mathbf{A} + \mathbf{y}'^\mathsf{T}\mathbf{A}' + \mathbf{y}''^\mathsf{T}\mathbf{A}'' = \mathbf{c}^\mathsf{T}$$

with $\mathbf{y}^\mathsf{T}\mathbf{b} + \mathbf{y}'^\mathsf{T}\mathbf{b}' + \mathbf{y}''^\mathsf{T}\mathbf{b}''$ 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:

$$\begin{array}{rl} \max & \mathbf{y}^\mathsf{T}\mathbf{b} + \mathbf{y}'^\mathsf{T}\mathbf{b}' + \mathbf{y}''^\mathsf{T}\mathbf{b}'' \\ \text{s.t.} & \mathbf{y}^\mathsf{T}\mathbf{A} + \mathbf{y}'^\mathsf{T}\mathbf{A}' + \mathbf{y}''^\mathsf{T}\mathbf{A}'' = \mathbf{c}^\mathsf{T} \\ & \mathbf{y} \geq 0 \\ & \mathbf{y}' \leq 0. \end{array} \tag{5.4}$$

In fact, we could have derived this dual by applying the definition of the dual problem to

$$\begin{array}{rl} \min & \mathbf{c}^\mathsf{T}\mathbf{x} \\ \text{s.t.} & \begin{bmatrix} \mathbf{A} \\ -\mathbf{A}' \\ \mathbf{A}'' \\ -\mathbf{A}'' \end{bmatrix} \mathbf{x} \geq \begin{bmatrix} \mathbf{b} \\ -\mathbf{b}' \\ \mathbf{b}'' \\ -\mathbf{b}'' \end{bmatrix}, \end{array}$$

which is equivalent to (11.3). The details are left as an exercise.

Observation 2

Consider the primal problem of the following form:

$$\begin{array}{rl}\min & \mathbf{c}^\mathsf{T}\mathbf{x} \\ \text{s.t.} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & x_i \geq 0 \;\; i \in P \\ & x_i \leq 0 \;\; i \in N\end{array} \tag{5.5}$$

where P and N are disjoint subsets of $\{1,\ldots,n\}$. 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 (11.5) as defined under Observation 1, we obtain the dual problem

$$\begin{array}{rll}\max & \mathbf{y}^\mathsf{T}\mathbf{b} & \\ \text{s.t.} & \mathbf{y}^\mathsf{T}\mathbf{a}^{(i)} = c_i & i \in \{1,\ldots,n\}\backslash(P \cup N) \\ & \mathbf{y}^\mathsf{T}\mathbf{a}^{(i)} + p_i = c_i & i \in P \\ & \mathbf{y}^\mathsf{T}\mathbf{a}^{(i)} + q_i = c_i & i \in N \\ & p_i \geq 0 & i \in P \\ & q_i \leq 0 & i \in N\end{array} \tag{5.6}$$

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, $i \in N$:

$$\begin{array}{rll}\max & \mathbf{y}^\mathsf{T}\mathbf{b} & \\ \text{s.t.} & \mathbf{y}^\mathsf{T}\mathbf{a}^{(i)} = c_i & i \in \{1,\ldots,n\}\backslash(P \cup N) \\ & \mathbf{y}^\mathsf{T}\mathbf{a}^{(i)} \leq c_i & i \in P \\ & \mathbf{y}^\mathsf{T}\mathbf{a}^{(i)} \geq c_i & i \in N,\end{array} \tag{5.7}$$

which can be taken as the dual problem of (11.5) instead of (11.6). The advantage here is that it has fewer variables than (11.6).

Hence, the dual problem of

$$\begin{array}{rl}\min & \mathbf{c}^\mathsf{T}\mathbf{x} \\ \text{s.t.} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq 0\end{array}$$

is simply

$$\begin{array}{rl} \max & \mathbf{y}^\mathsf{T}\mathbf{b} \\ \text{s.t.} & \mathbf{y}^\mathsf{T}\mathbf{A} \leq \mathbf{c}^\mathsf{T} \\ & \mathbf{y} \geq 0. \end{array}$$

As we can see from bove, 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.

5.2.1. 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 ${\mathbf{a}^{(i)}}^\mathsf{T}$ denote the ith row of $\mathbf{A}$. Let $\mathbf{A}_j$ denote the jth 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 $\mathbf{c}^\mathsf{T}\mathbf{x}$
- The constraints are

$${\mathbf{a}^{(i)}}^\mathsf{T}\mathbf{x} \sqcup_i b_i$$

  where $\sqcup_i$ is $\leq$, $\geq$, or $=$ for $i = 1,\ldots,m$.
- For each $j \in \{1,\ldots,n\}$, 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 $\mathbf{y}^\mathsf{T}\mathbf{b}$
- For $j = 1,\ldots,n$, the jth constraint is

$$\begin{cases} \mathbf{y}^\mathsf{T}\mathbf{A}_j \leq c_j & \text{if } x_j \text{ is constrained to be nonnegative} \\ \mathbf{y}^\mathsf{T}\mathbf{A}_j \geq c_j & \text{if } x_j \text{ is constrained to be nonpositive} \\ \mathbf{y}^\mathsf{T}\mathbf{A}_j = c_j & \text{if } x_j \text{ is free.} \end{cases}$$

- For each $i \in \{1,\ldots,m\}$, y_i is constrained to be nonnegative if $\sqcup_i$ is $\geq$; y_i is constrained to be non-positive 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:

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

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:

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

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

Theorem 5.2: 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 11.2.1 is also known informally as **strong duality**.

Exercises

1. Write down the dual problem of

$$\begin{array}{rrcrcl} \min & 4x_1 & - & 2x_2 & & \\ \text{s.t.} & x_1 & + & 2x_2 & \geq & 3 \\ & 3x_1 & - & 4x_2 & = & 0 \\ & & & x_2 & \geq & 0. \end{array}$$

2. Write down the dual problem of the following:

$$\begin{array}{rrcrcrcl} \min & & & 3x_2 & + & x_3 & & \\ \text{s.t.} & x_1 & + & x_2 & + & 2x_3 & = & 1 \\ & x_1 & & & - & 3x_3 & \leq & 0 \\ & x_1 & , & x_2 & , & x_3 & \geq & 0. \end{array}$$

3. Write down the dual problem of the following:

$$\begin{array}{rrcrcrcl} \min & x_1 & & & - & 9x_3 & & \\ \text{s.t.} & x_1 & - & 3x_2 & + & 2x_3 & = & 1 \\ & x_1 & & & & & \leq & 0 \\ & & & x_2 & & & & \text{free} \\ & & & & & x_3 & \geq & 0. \end{array}$$

4. Determine all values c_1, c_2 such that the linear programming problem

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

has an optimal solution. Justify your answer

Solutions

1. The dual is

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

2. The dual is

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

3. The dual is

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

4. Let (P) denote the given linear programming problem.

   Note that $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ is a feasible solution to (P). Therefore, by Theorem **??**, it suffices to find all values c_1, c_2 such that

   (P) is not unbounded. This amounts to finding all values c_1, c_2 such that the dual problem of (P) has a feasible solution.

   The dual problem of (P) is

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

   The two equality constraints gives $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{3}{5}c_1 - \frac{1}{5}c_2 \\ -\frac{1}{5}c_1 + \frac{2}{5}c_2 \end{bmatrix}$. Thus, the dual problem is feasible if and only if c_1 and c_2 are real numbers satisfying

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

or more simply,

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

6. Sensitivity Analysis

Chapter 6. Sensitivity Analysis
0% complete. Goal 80% completion date: January 20, 2023
Notes: Need to write this section. Add exmaples from lecture notes. Create code to help generate examples.

7. Multi-Objective Optimization

Chapter 7. Multi-Objective Optimization
10% complete. Goal 80% completion date: January 20 ,2023
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)

7.1 Multi Objective Optimization and The Pareto Frontier

On Dealing with Ties and Multiple Objectives in Linear Programming

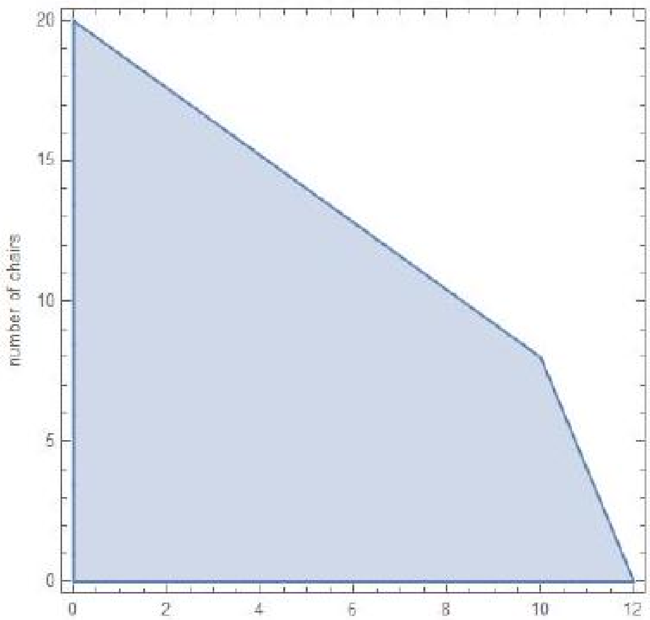
Consider a high end furniture manufacturer which builds dining tables and chairs out of expensive bocote and rosewood.

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 sells for \$7000 while a chair goes for \$1500. To increase profit we want to maximize

$$F(x,y) = 8000x + 2000y$$

over P. Having taken a linear programming class, the manager knows his way around these problems and begins the simplex method:

Maximize $8000x + 2000y$

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

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

$$x, y \geq 0$$

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

Maximize $4x + y$

s.t. $4x + y + s_1 = 48$

$$\frac{f}{2000} \quad 6x + 5y + s_2 = 100$$

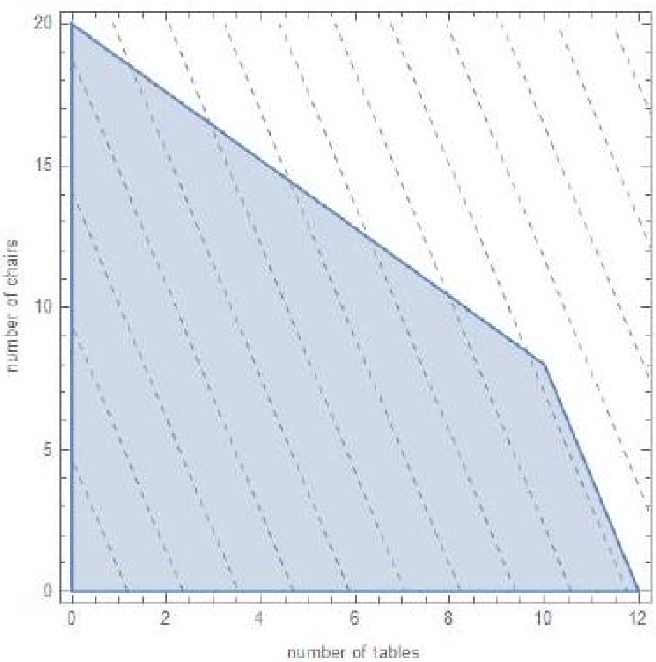
$$x, y \geq 0$$

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 this from the tableau? From the original formulation?

Is the manager's solution really the best?



number of tables 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:

$$\begin{aligned} &\text{Minimize } 10x + 2y \\ &\text{s.t.} \quad 80x + 20y = 960 \\ &\qquad\qquad x \in [10, 12] \\ &\qquad\qquad y \in [0, 8]. \end{aligned}$$

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

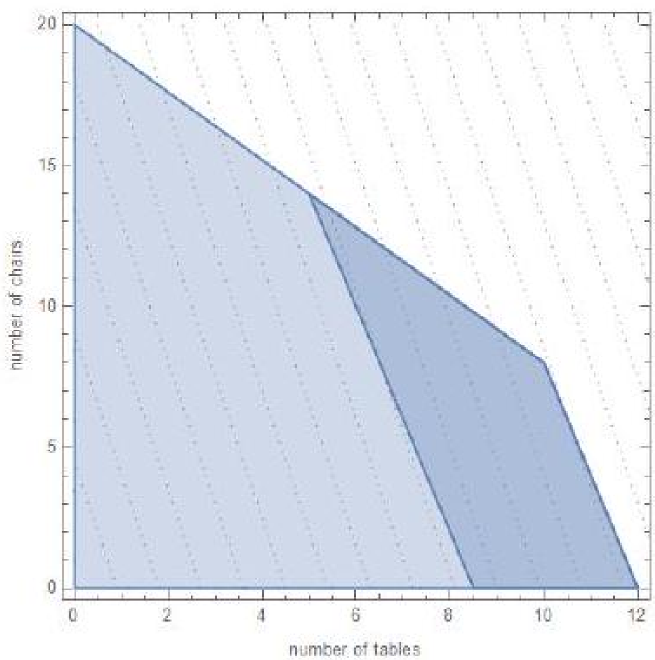
$$\arg\min\{10x + 2y : (x,y) \in \{(12,0),(10,8)\}\}$$

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:

$$\begin{aligned} & \text{Minimize } 10x+2y \\ \text{s.t.} \quad & 8000x+2000y \geq (\alpha)96000 \\ & 80x+20y \leq 960 \\ & 12x+10y \leq 200 \\ & x,y \geq 0 \end{aligned}$$

where $\alpha = 0.7$ This new constraint limits us to solutions which offered at least 70% of maximum possible revenue.



The strategy is the 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 $f_i : P \to \mathbb{R}$ that we seek to maximize, we call a point $\mathbf{x} \in P$ Pareto Optimal or Efficient if there does not exist another point $\overline{\mathbf{x}} \in P$ such that

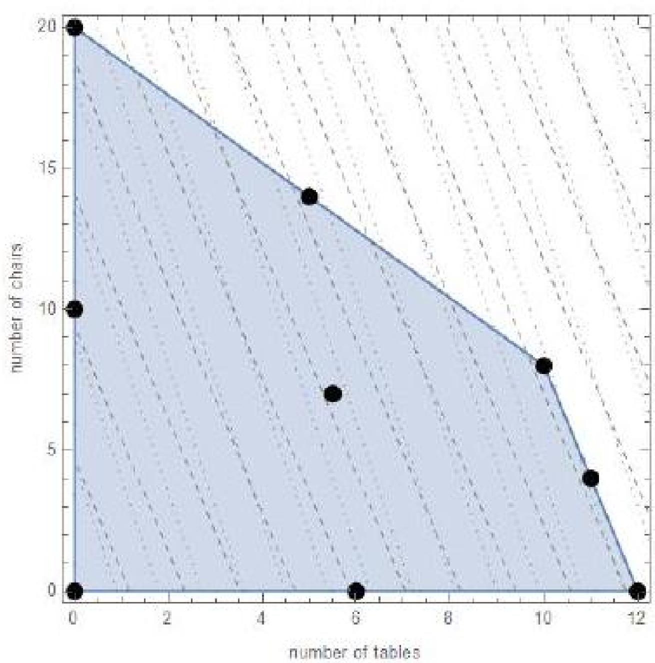
- $f_i(\overline{\mathbf{x}}) > f_i(\mathbf{x})$ for some i and

$\to f_j(\overline{\mathbf{x}}) \geq f_j(\mathbf{x})$ for all $j \neq i$.

That is, we cannot make any objective better without making some other objective worse.

The Pareto Frontier is the set of all Pareto optimal points for some problem. Which of these points is Pareto optimal?

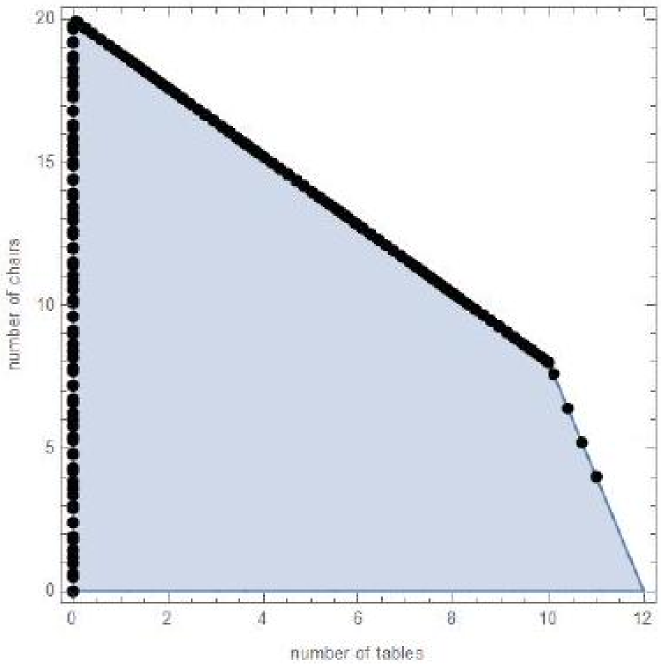
What is the frontier of this problem?



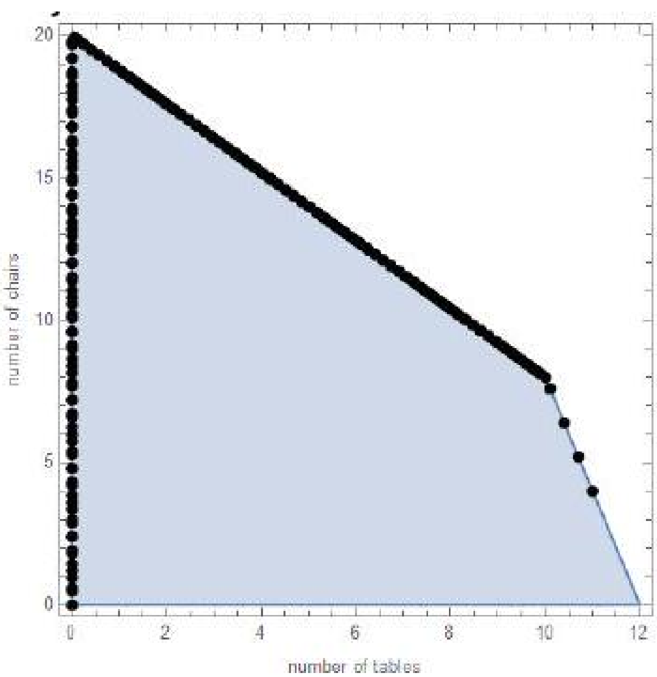
number of tables The rollover method is generalized in Goal Programming

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

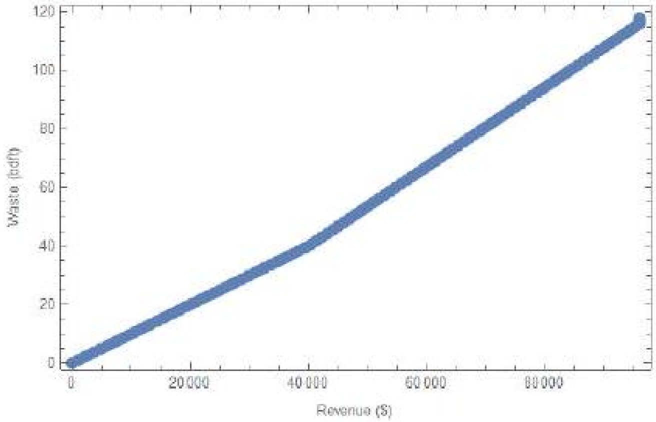
However, this method can generate inefficient solutions if the underlying model is poorly constructed.



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



number of table



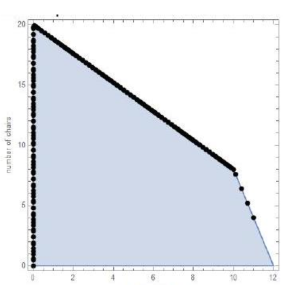
One of the owners of our manufactory decides to explore possible planning himself; he implements the multi-objective method that he remembers, Scalarization by picking some arbitrary constant $\lambda \in [0,1]$ and combining his two objectives like so:

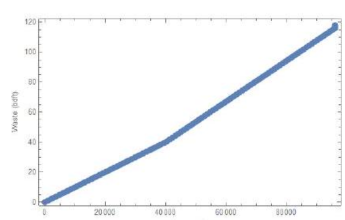
$$
\begin{array}{ll}
\text{Minimize} & \lambda(8000x+2000y)+(1-\lambda)(10x+2y) \\
\text{s.t.} & 80x+20y \leq 960 \\
& 12x+10y \leq 200 \\
& x,y \geq 0
\end{array}
$$

What is the benefit of this method?

Where does it fall short?

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

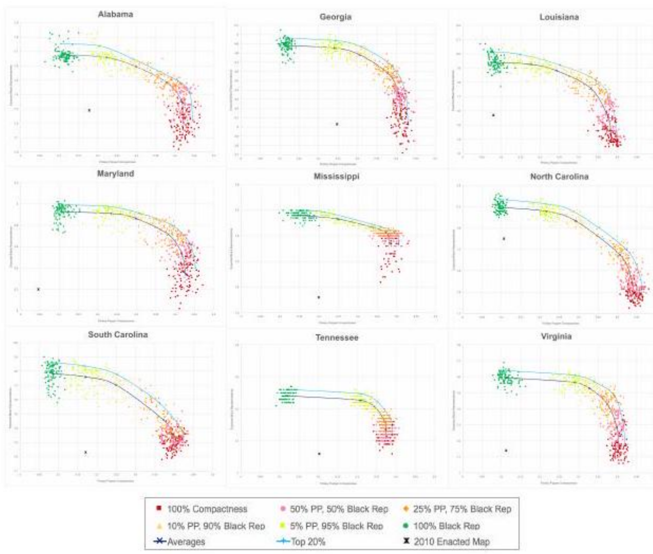




These are all nice ideas, but the problem presented above is neither difficult nor practical.

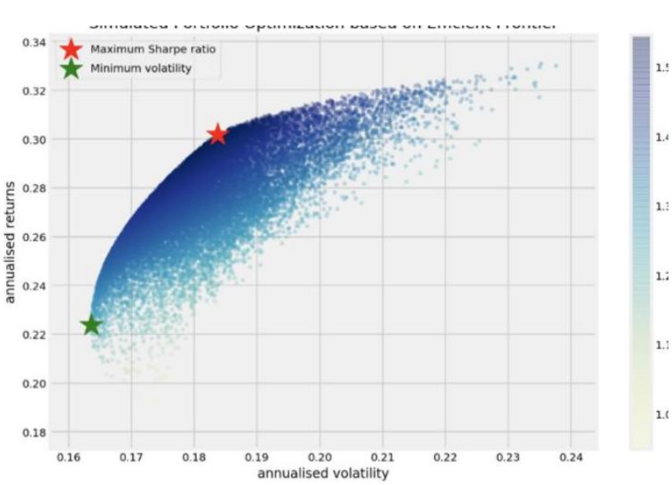
What are some areas that a Pareto frontier would be actually useful?

7.3 Political Redistricting [3]

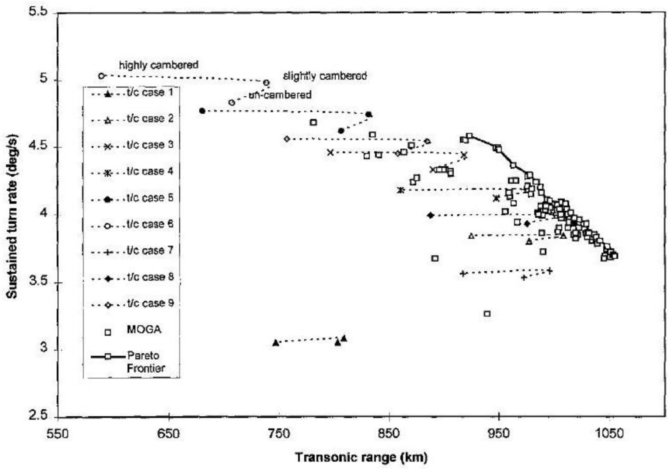


7.4 Portfolio Optimization [5]

7.5 Simulated Portfolio Optimization based on Efficient Frontier



7.6 Aircraft Design [1]



7.7 Vehicle Dynamics [4]

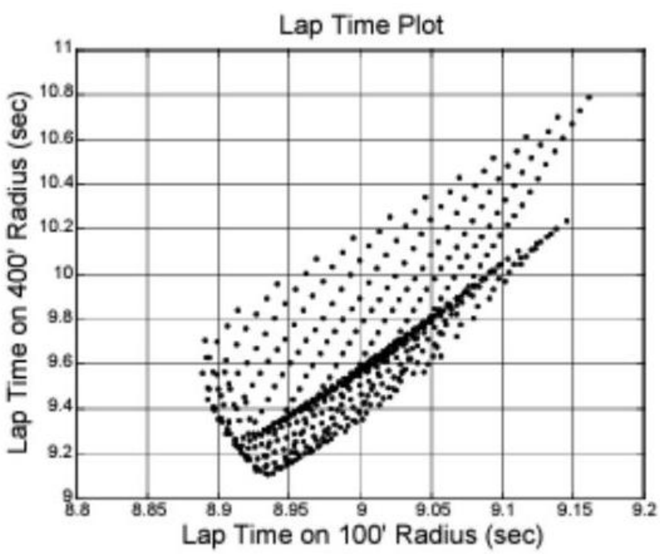
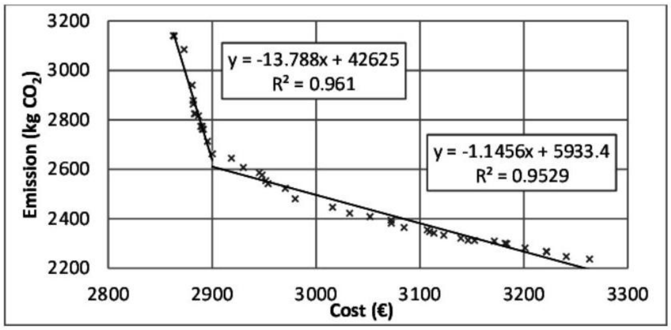


Figure 7: Grid Search Results in the Performance Space

7.8 Sustainable Constriction [2]



7.9 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

8. Graph Algorithms

Chapter 8. Graph Algorithms
10% complete. Goal 80% completion date: August 20
Notes: .

Write this section.

8.1 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.

8.2 Graphs

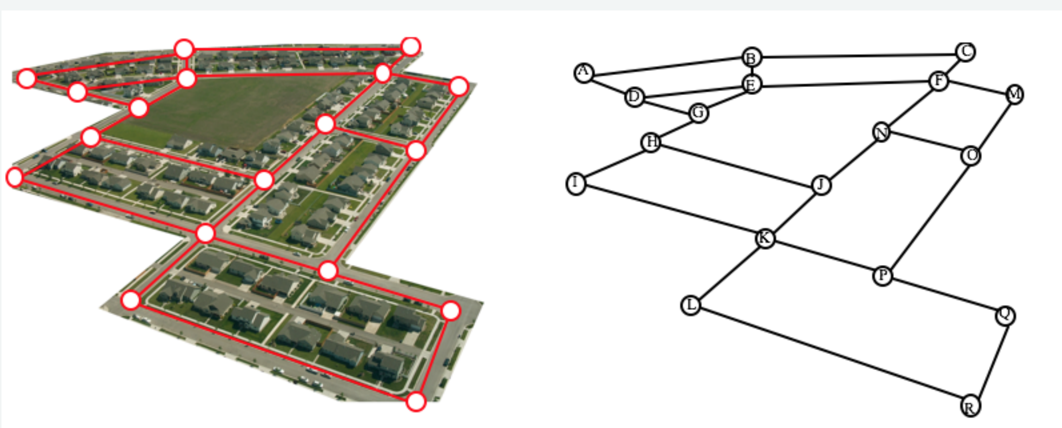
8.2.1. Drawing Graphs

Example 8.1

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.



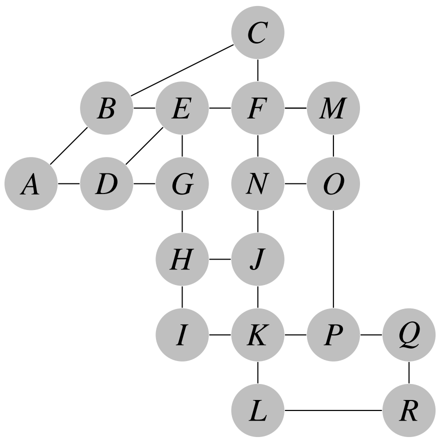
[a]Same Beebe. http://www.flickr.com/photos/sbeebe/2850476641/

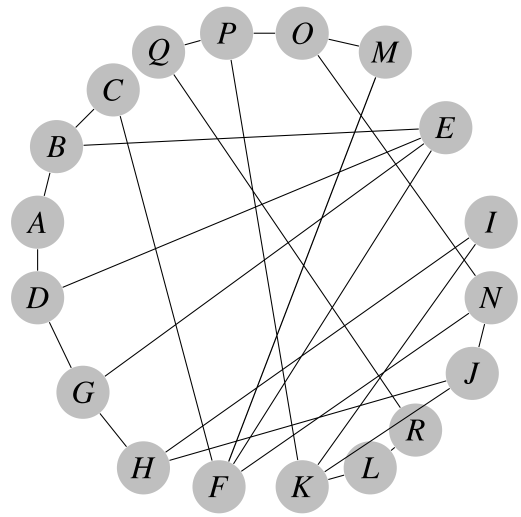
This type of simplified picture is called a **graph**.

Definition 8.2: 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.

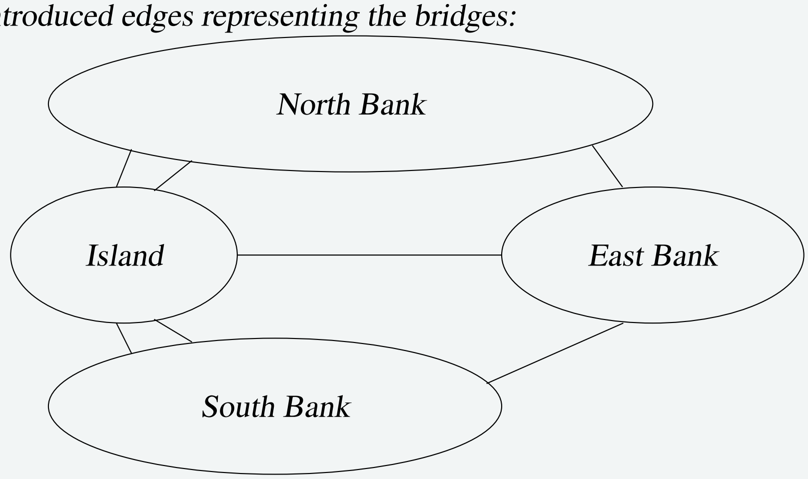
Example 8.3

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

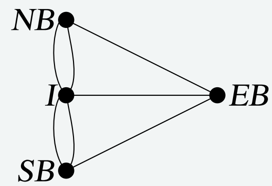
Picture

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.

8.3 Definitions

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

Definition 8.4: Vertex

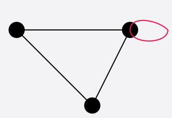
A vertex is a dot in the graph that could represent an intersection of streets, a land mass, or a general location, like "work? or "school". Vertices are often connected by edges. Note that vertices only occur when a dot is explicitly placed, not whenever two edges cross. Imagine a freeway overpass – 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 8.5: 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.

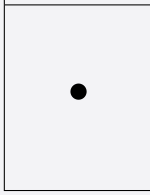
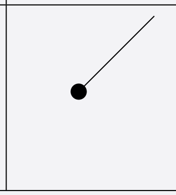
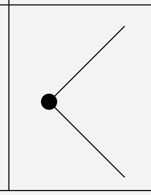
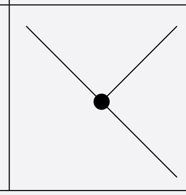
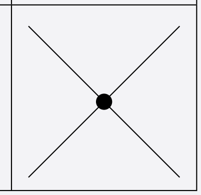
Definition 8.6: Loop

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



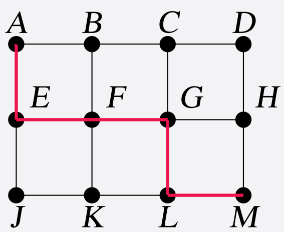
Definition 8.7: Degree of a vertex

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.

Degree 0	Degree 1	Degree 2	Degree 3	Degree 4
				

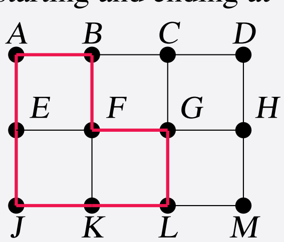
Definition 8.8: Path

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.



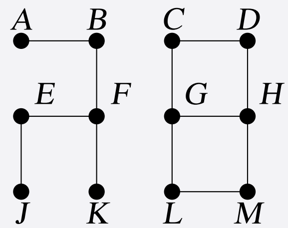
Definition 8.9: Circuit (a.k.a. cycle)

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 8.10: 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.*



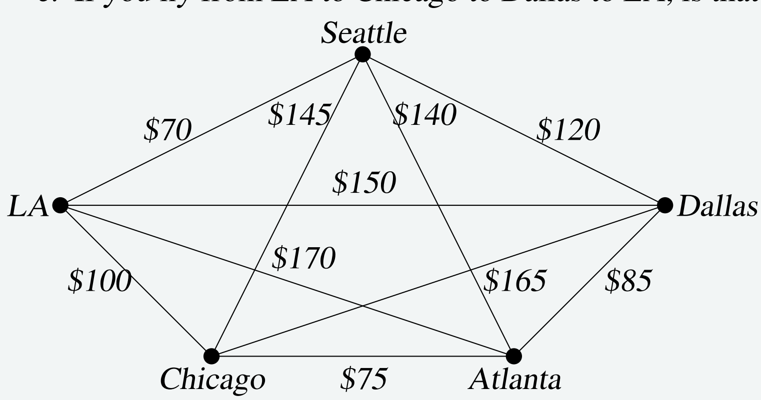
Definition 8.11: Weights

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 8.12

The graph below shows 5 cities. The weights on the edges represent the airfare for a one-way flight between the cities.

a. *How many vertices and edges does the graph have?*
b. *Is the graph connected?*
c. *What is the degree of the vertex representing LA?*
d. *If you fly from Seattle to Dallas to Atlanta, is that a path or a circuit?*
e. *If you fly from LA to Chicago to Dallas to LA, is that a path or a circuit?*



8.4 Shortest Path

Outcomes

- *What is the problem statement?*
- *How to use Dijkstra's algorihtm*
- *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.

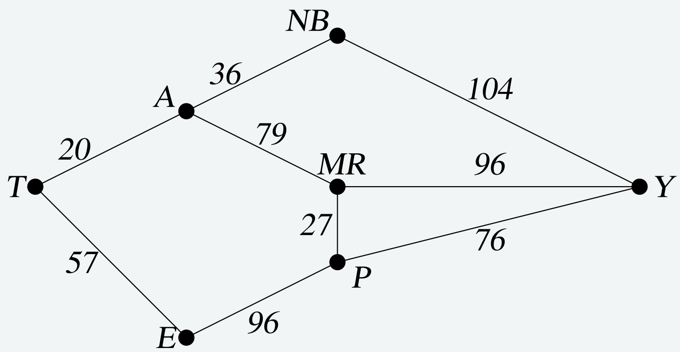
While often it is possible to find a shortest path on a small graph by guess-and-check, our goal in this chapter is to develop methods to solve complex problems in a systematic way by following **algorithms**. An algorithm is a step-by-step procedure for solving a problem. Dijkstra's (pronounced dike-stra)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

Dijkstra's Algorithm

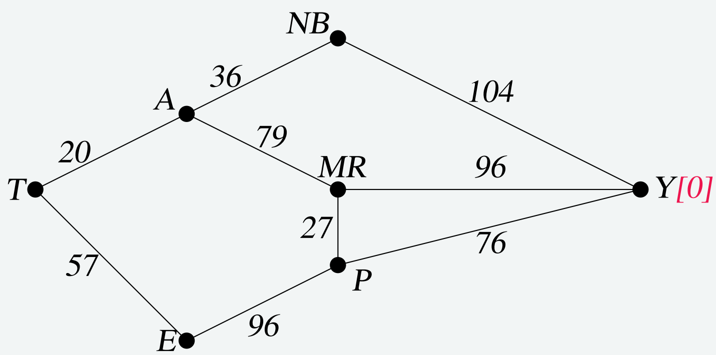
1. Mark the ending vertex with a distance of zero. Designate this vertex as current.
2. Find all vertices leading to the current vertex. Calculate their distances to the end. Since we already know the distance the current vertex is from the end, this will just require adding the most recent edge. Don't 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 8.13

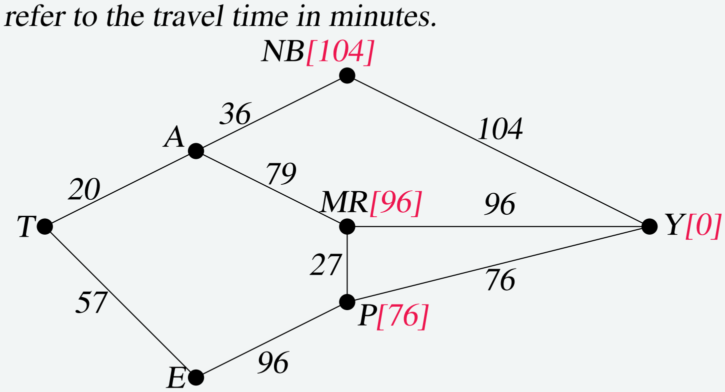
Suppose you need to travel from Yakima, WA (vertex Y) to Tacoma, WA (vertex T) Looking at a map, it looks like driving through Auburn (A) then Mount Rainier (MR) might be shortest, but it's not totally clear since that road is probably slower than taking the major highway through North Bend (NB). A graph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



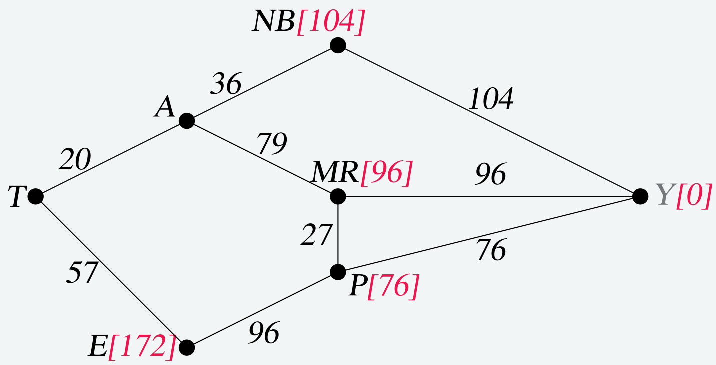
Step 2: For each vertex leading to Y, we calculate the distance to the end. For example, NB is a distance of 104 from the end, and MR is 96 from the end. Remember that distances in this case refer to the travel time in minutes.



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.

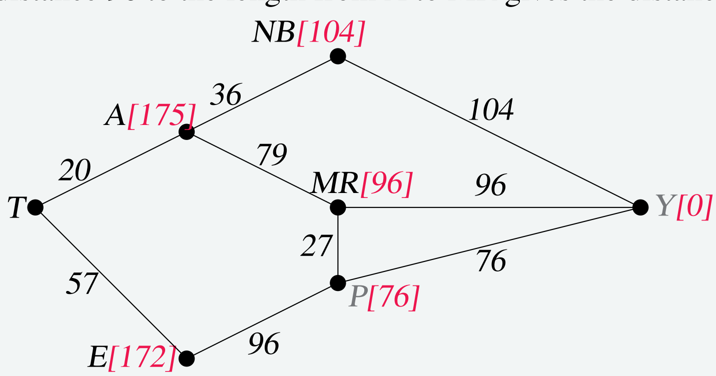
Step 2 (#2): For each vertex leading to P (and not leading to a visited vertex) we find the distance from the end. Since E is 96 minutes from P, and we've already calculated P is 76 minutes from Y, we can compute that E is $96+76=172$ *minutes from Y.*

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.*



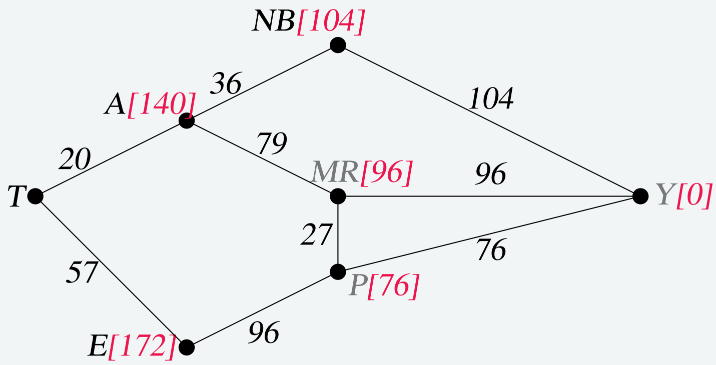
Step 3 & 4 (#2): We mark P as visited, and designate the vertex with the smallest recorded distance as current: MR. Back to step 2.

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.*



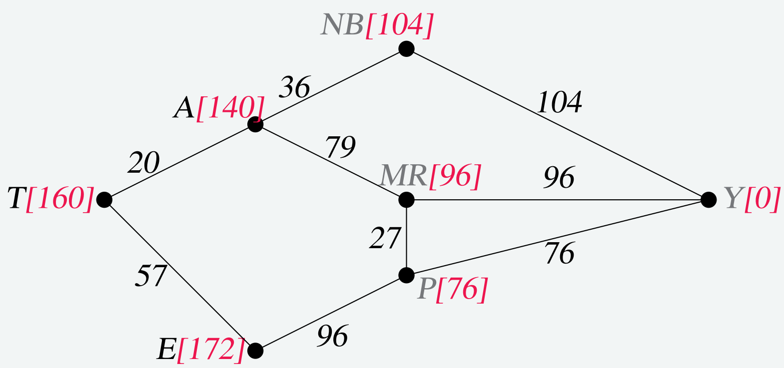
Step 3 & 4 (#3): We mark MR as visited, and designate the vertex with smallest recorded distance as current: NB. Back to step 2.

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 Tacp,a will take 160 minutes. Tracking which sequence of edges yielded 160 minutes, we see the shortest path is Y-NB-A-T.

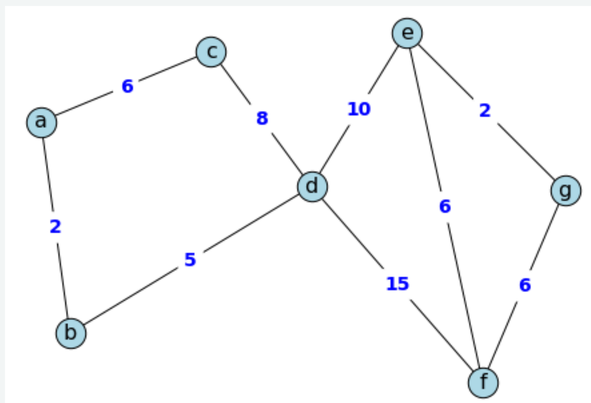
Dijkstra's algorithm is an **optimal algorithm**, meaning that it always produces the actual shortest path, not just a path that is pretty short, provided one exists. This algorithm is also **efficient**, meaning that it can be implemented in a reasonable amount of time. Dijkstra's algorithm takes around V^2 calculations, where V is the number of vertices in a graph[1]. 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.

[1]It can be made to run faster through various optimizations to the implementation.

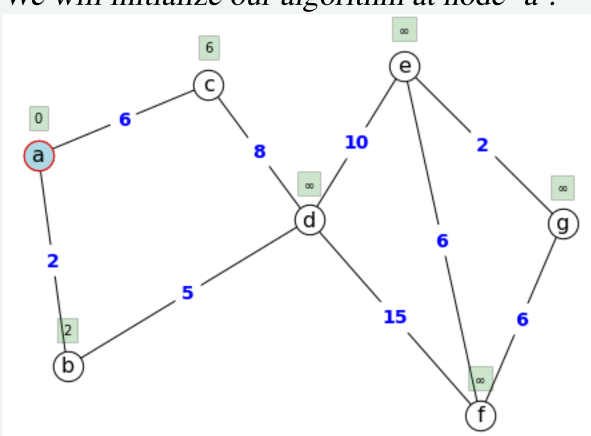
In contrast, an **inefficient** algorithm might try to list all possible paths then compute the length of each path. Trying to list all possible paths could easily take 10^{25} calculations to compute the shortest path with only 25 vertices; that's a 1 with 25 zeros after it! To put that in perspective,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

Example 8.14: Dijkstra's algorithm example

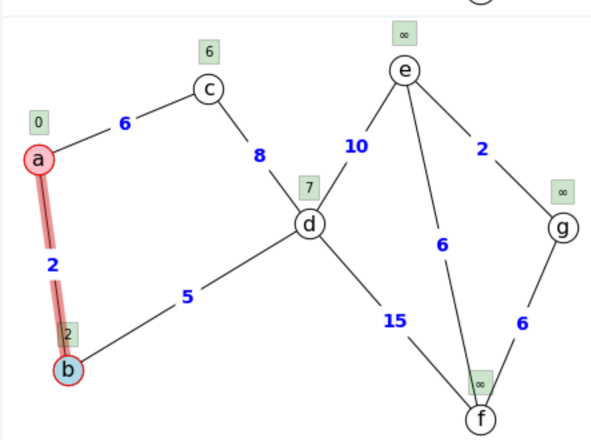
We would like to find a shortest path in the graph from node a to node g. See Code for python code to solve this problem and create these graphics.



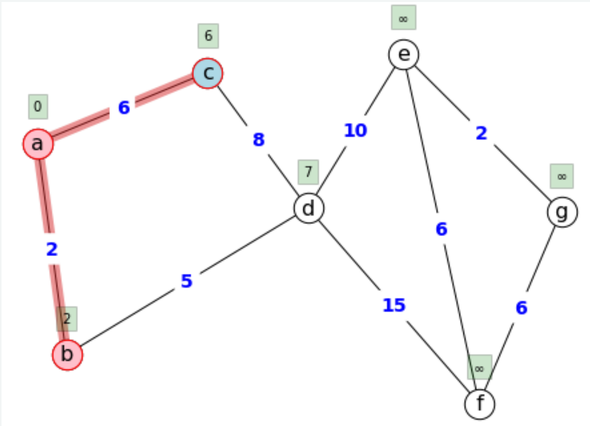
We will initialize our algorithm at node 'a'.



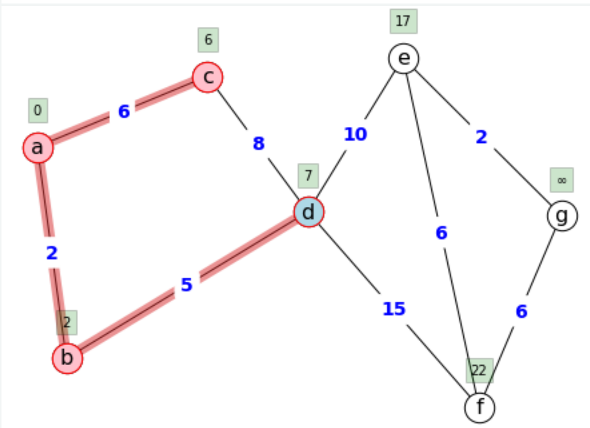
current	a	b	c	d	e	f	g
a	0	2	6	∞	∞	∞	∞



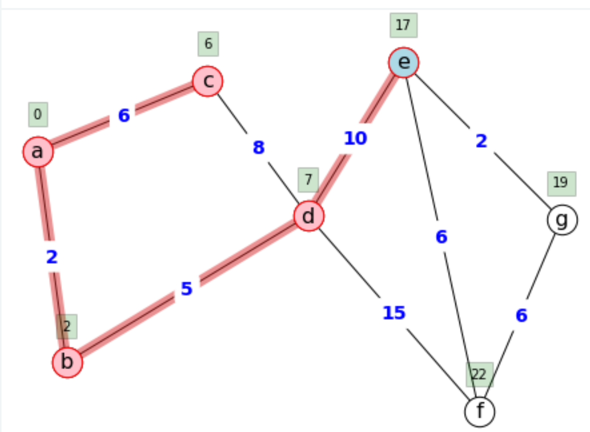
current	a	b	c	d	e	f	g
b	0	2	6	7	∞	∞	∞



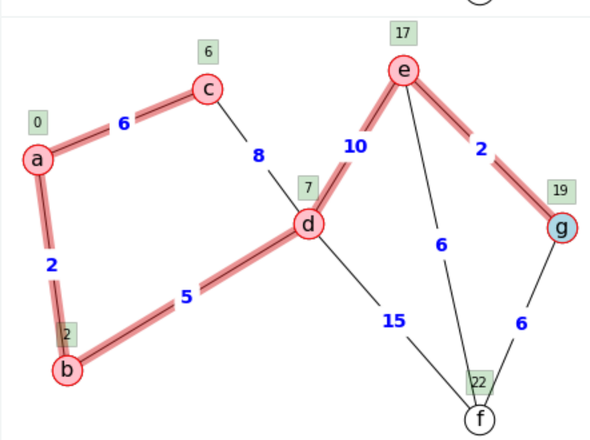
current	a	b	c	d	e	f	g
c	0	2	6	7	∞	∞	∞



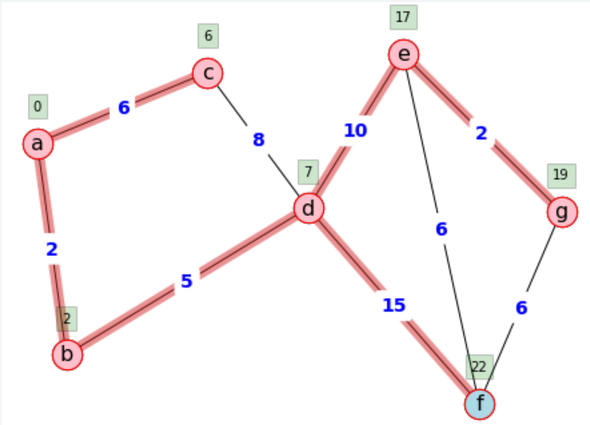
current	a	b	c	d	e	f	g
d	0	2	6	7	17	22	∞



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

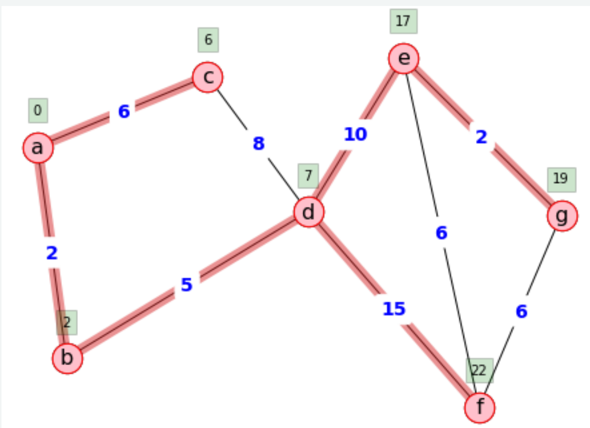


current	a	b	c	d	e	f	g
g	0	2	6	7	17	22	19



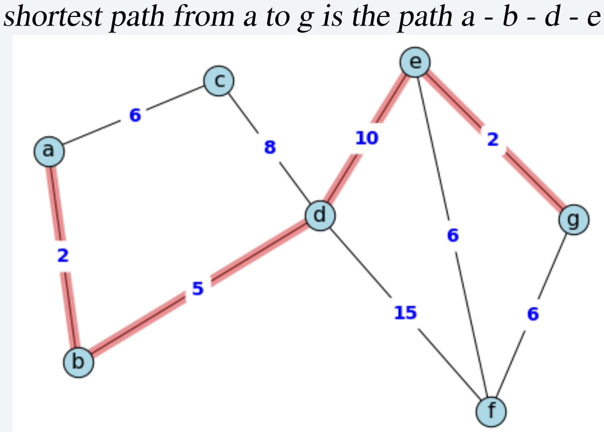
current	a	b	c	d	e	f	g
f	*0*	*2*	*6*	*7*	*17*	*22*	*19*

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 8.15

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			*

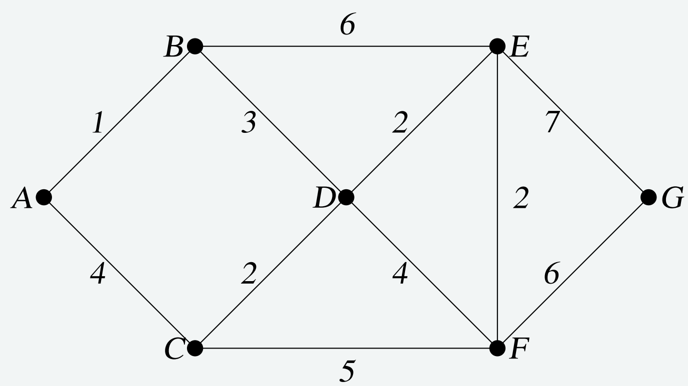
Step 2 (#5): The distance from Atlanta to Baltimore is 14. Adding that to the distance already calculated for Atlanta gives a total distance of $14+40=54$ hours from Baltimore to Bakersfield through Atlanta. Since this is larger than the currently calculated distance, we do not replace the distance for 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 8.16

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



8.5 Spanning Trees

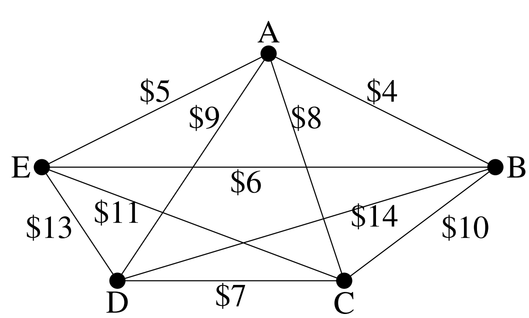
Outcomes

- *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.

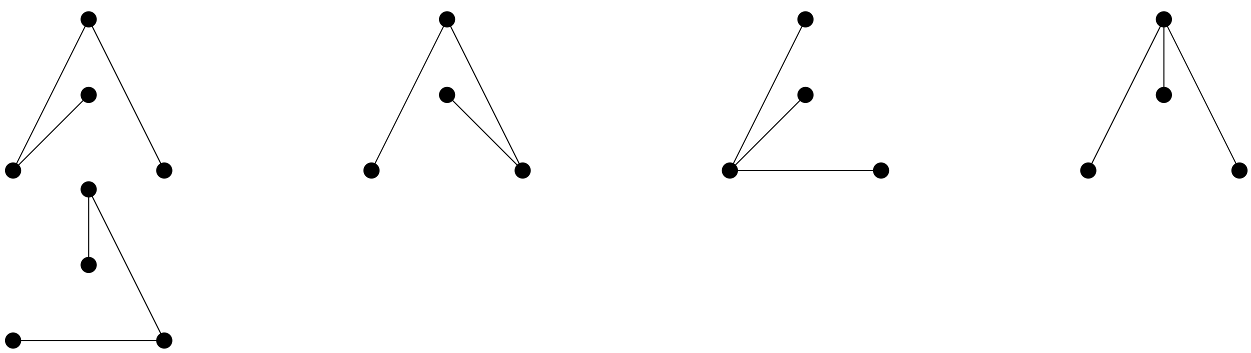


In this case, we don't need to find a circuit, or even a specific path; all we need to do is make sure we can make a call from any office to any other. In other words, we need to be sure there is a path from any vertex to any other vertex.

Definition 8.17: 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 8.18: 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.

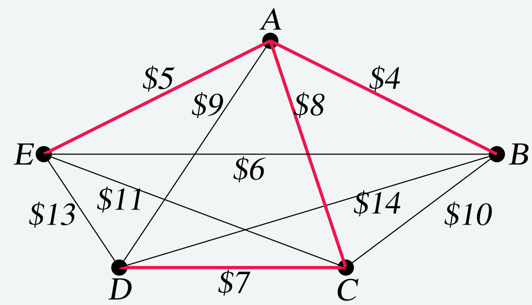
Kruskal's Algorithm

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 8.19

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 8.20

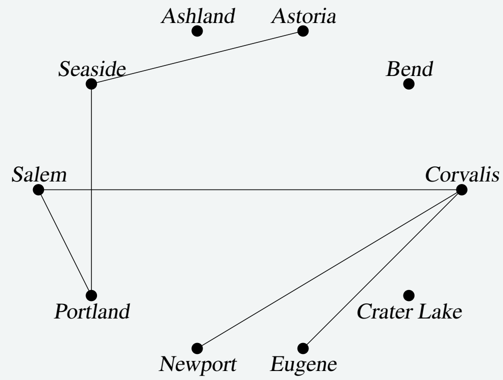
The power company needs to lay updated distribution lines connecting the ten Oregon cities below to the power grid. How can they minimize the amount of new line to lay?

	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.

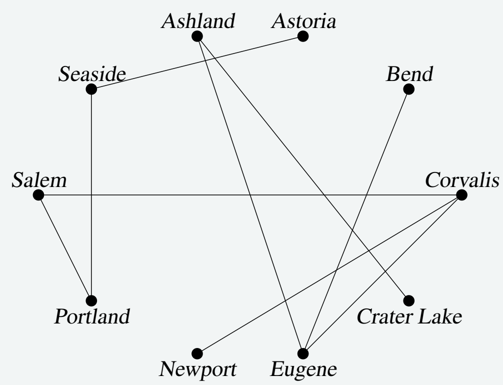
Seaside to Astoria	*17 miles*
Corvallis to Salem	*40 miles*
Portland to Salem	*47 miles*
Corvallis to Eugene	*47 miles*
Corvallis to Newport	*52 miles*
Salem to Eugene	*reject – closes circuit*
Portland to Seaside	*78 miles*

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



Continuing,

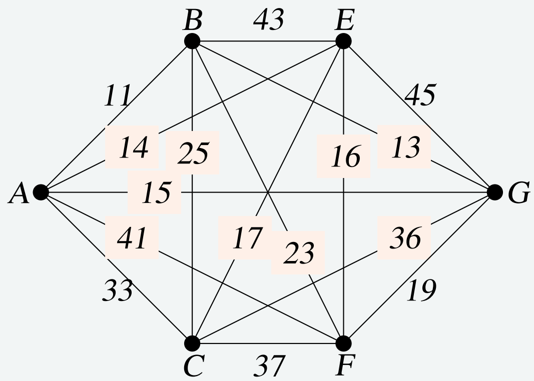
Newport to Salem	*reject*
Corvallis to Portland	*reject*
Eugene to Newport	*reject*
Portland to Astoria	*reject*
Ashland to Crater Lake	*108 miles*
Eugene to Portland	*reject*
Newport to Portland	*reject*
Newport to Seaside	*reject*
Salem to Seaside	*reject*
Bend to Eugene	*128 miles*
Bend to Salem	*reject*
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 8.21: Min Cost Spanning Tree

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



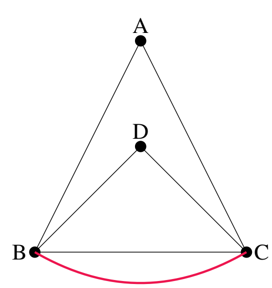
8.6 Exercise Answers

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: ABEGFCDFEDBCA

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.

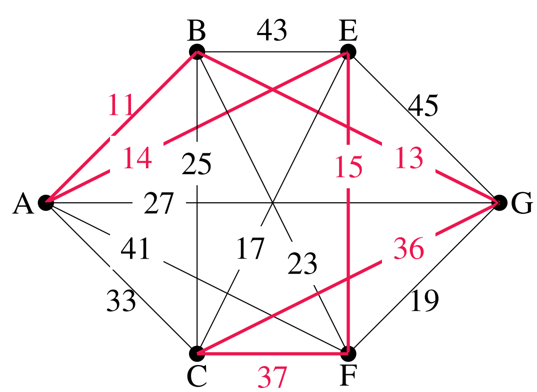
The NNA circuit from B is BEDACFB with time 158 milliseconds.

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 RNNA 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

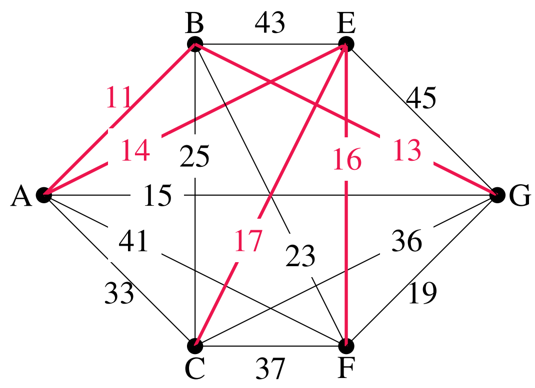
Final circuit: ABGCFEA



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

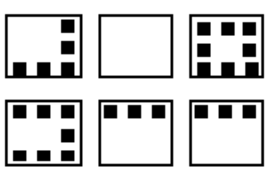
This completes the spanning tree.



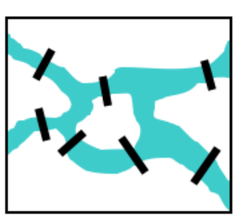
8.7 Additional Exercises

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.



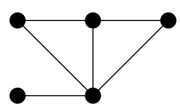
3. The table below shows approximate driving times (in minutes, without traffic) between five cities in the Dallas area. Create a weighted graph representing this data.

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

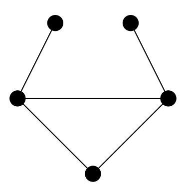
4. Shown in the table below are the one-way airfares between 5 cities[2]. Create a graph showing this data.

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

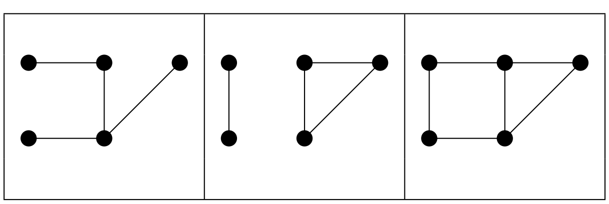
5. Find the degree of each vertex in the graph below.



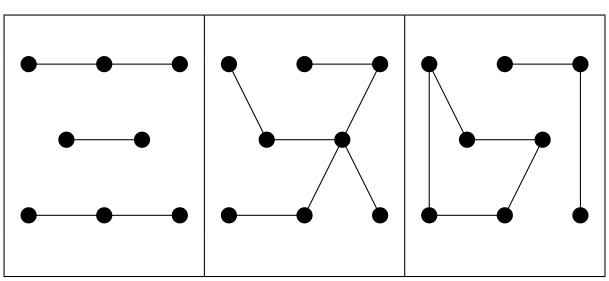
6. Find the degree of each vertex in the graph below.



7. Which of these graphs are connected?

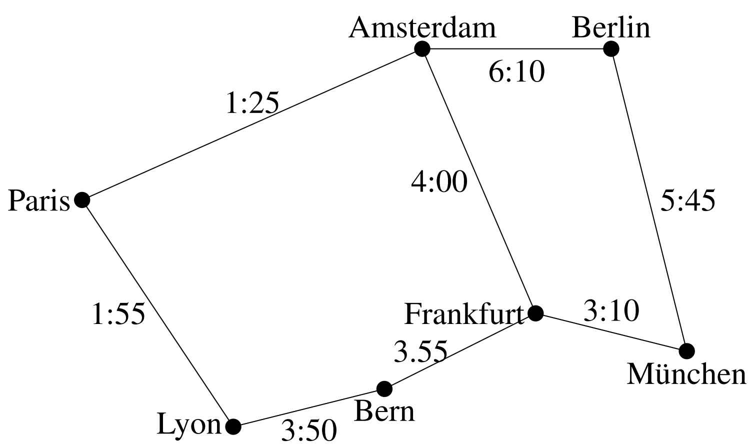


8. Which of these graphs are connected?



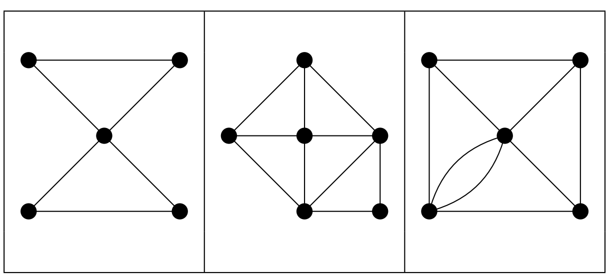
[2]Cheapest fares found when retrieved Sept. 1, 2009 for travel Sept. 22, 2009

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

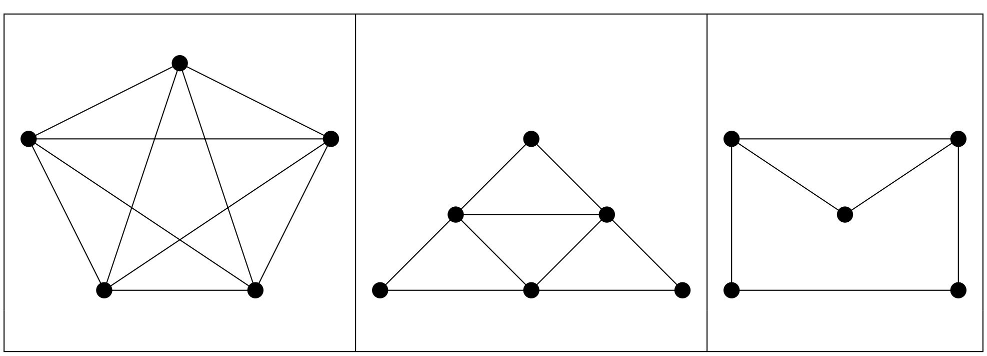


10. Using the graph from the previous problem, find the path with shortest travel time from Paris to München.

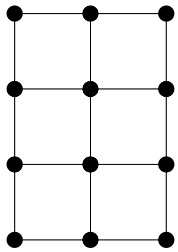
11. Does each of these graphs have an Euler circuit? If so, find it.



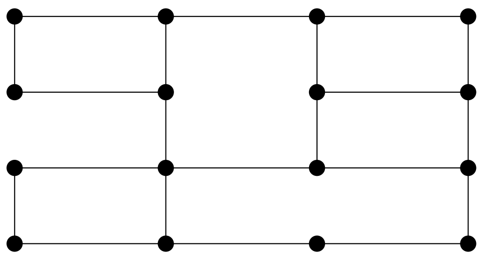
12. Does each of these graphs have an Euler circuit? If so, find it.



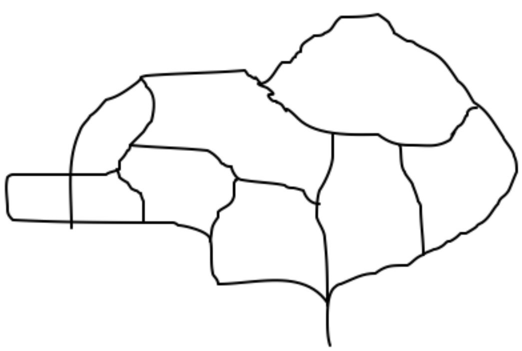
13. Eulerize this graph using as few edge duplications as possible. Then, find an Euler circuit.



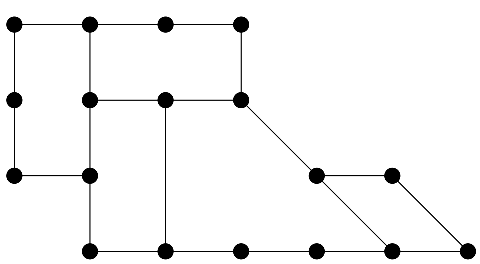
14. Eulerize this graph using as few edge duplications as possible. Then, find an Euler circuit.



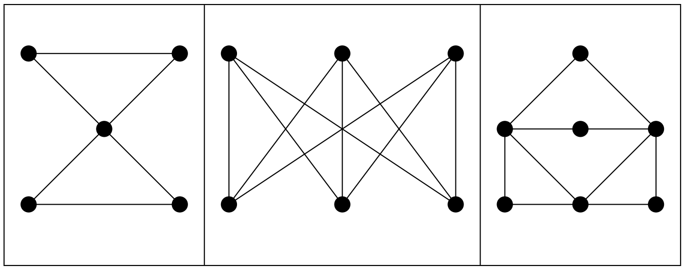
15. The maintenance staff at an amusement park need to patrol the major walkways, shown in the graph below, collecting litter. Find an efficient patrol route by finding an Euler circuit. If necessary, eulerize the graph in an efficient way.



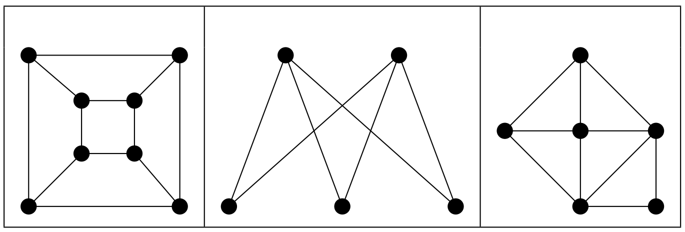
16. After a storm, the city crew inspects for trees or brush blocking the road. Find an efficient route for the neighborhood below by finding an Euler circuit. If necessary, eulerize the graph in an efficient way.



17. Does each of these graphs have at least one Hamiltonian circuit? If so, find one.



18. Does each of these graphs have at least one Hamiltonian circuit? If so, find one.



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

    (a) Using Nearest Neighbor starting in Fort Worth

    (b) Using Repeated Nearest Neighbor

    (c) Using Sorted Edges

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

20. A salesperson needs to travel from Seattle to Honolulu, London, Moscow, and Cairo. Use the table of flight costs from problem #4 to find a route for this person to follow:

    (a) Using Nearest Neighbor starting in Seattle

    (b) Using Repeated Nearest Neighbor

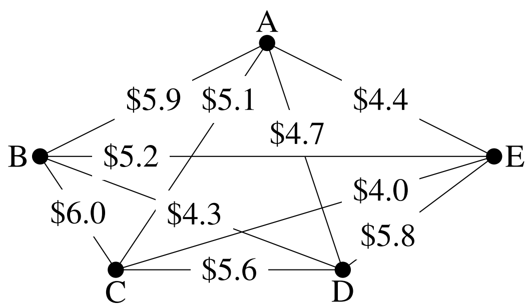
    (c) Using Sorted Edges

21. When installing fiber optics, some companies will install a sonet ring; a full loop of cable connecting multiple locations. This is used so that if any part of the cable is damaged it does not interrupt service, since there is a second connection to the hub. A company has 5 buildings. Costs (in thousands of dollars) to lay cables between pairs of buildings are shown below. Find the circuit that will minimize cost:

    (a) Using Nearest Neighbor starting at building A

(b) Using Repeated Nearest Neighbor

(c) Using Sorted Edges



22. A tourist wants to visit 7 cities in Israel. Driving distances, in kilometers, between the cities are shown below[3]. Find a route for the person to follow, returning to the starting city:

    (a) Using Nearest Neighbor starting in Jerusalem

    (b) Using Repeated Nearest Neighbor

    (c) Using Sorted Edges

	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.

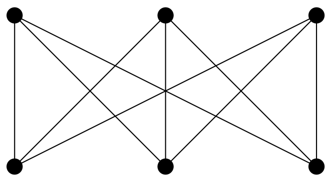
Concepts

26. Can a graph have one vertex with odd degree? If not, are there other values that are not possible? Why?

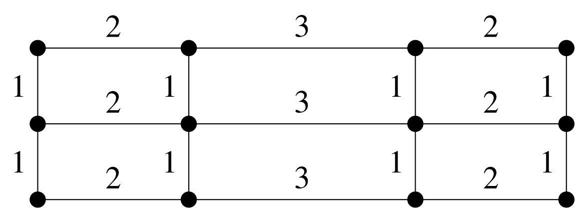
[3] From `http://www.ddtravel-acc.com/Israel-cities-distance.htm`

27. A complete graph is one in which there is an edge connecting every vertex to every other vertex. For what values of n does complete graph with n vertices have an Euler circuit? A Hamiltonian circuit?

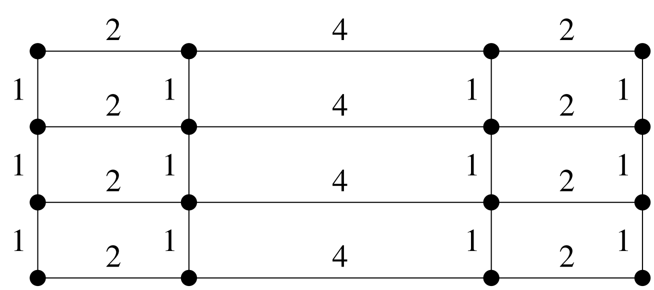
28. Create a graph by drawing n vertices in a row, then another n vertices below those. Draw an edge from each vertex in the top row to every vertex in the bottom row. An example when $n = 3$ is shown below. For what values of n will a graph created this way have an Euler circuit? A Hamiltonian circuit?



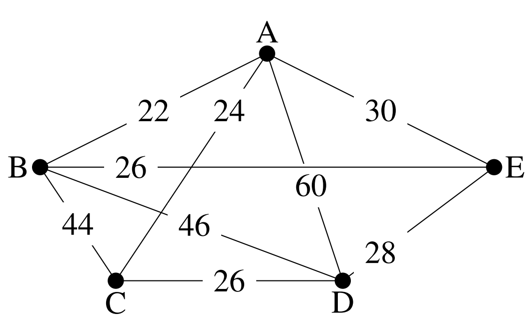
29. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



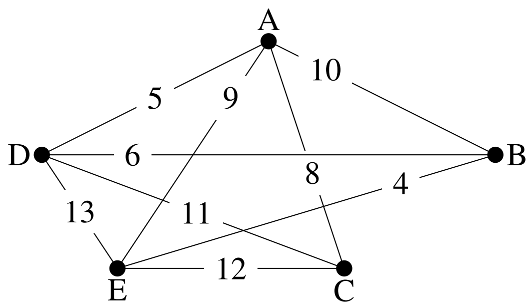
30. Eulerize this graph in the most efficient way possible, considering the weights of the edges.



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



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



Explorations

33. Social networks such as Facebook and LinkedIn can be represented using graphs in which vertices represent people and edges are drawn between two vertices when those people are “friends." The table below shows a friendship table, where an X shows that two people are friends.

	A	B	C	D	E	F	G	H	I
A		X	X			X	X		
B			X		X				
C					X				
D					X				X
E							X		X
F								X	X
G								X	
H									X

(a) Create a graph of this friendship table

(b) Find the shortest path from A to D. The length of this path is often called the “degrees of separation" of the two people.

(c) Extension: Split into groups. Each group will pick 10 or more movies, and look up their major actors (www.imdb.com is a good source). Create a graph with each actor as a vertex, and edges connecting two actors in the same movie (note the movie name on the edge). Find interesting paths between actors, and quiz the other groups to see if they can guess the connections.

34. A spell checker in a word processing program makes suggestions when it finds a word not in the dictionary. To determine what words to suggest, it tries to find similar words. One measure of word similarity is the Levenshtein distance, which measures the number of substitutions, additions, or deletions that are required to change one word into another. For example, the words spit and spot are a distance of 1 apart; changing spit to spot requires one substitution (i for o). Likewise, spit is distance 1 from pit since the change requires one deletion (the s). The word spite is also distance

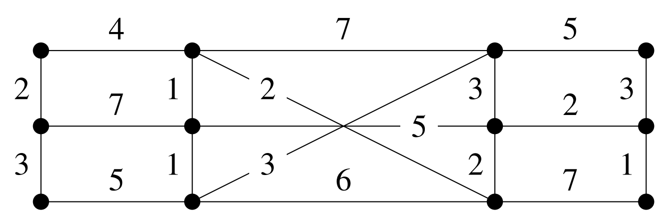
1 from spit since it requires one addition (the e). The word soot is distance 2 from spit since two substitutions would be required.

(a) Create a graph using words as vertices, and edges connecting words with a Levenshtein distance of 1. Use the misspelled word “moke" as the center, and try to find at least 10 connected dictionary words. How might a spell checker use this graph?

(b) Improve the method from above by assigning a weight to each edge based on the likelihood of making the substitution, addition, or deletion. You can base the weights on any reasonable approach: proximity of keys on a keyboard, common language errors, etc. Use Dijkstra?s algorithm to find the length of the shortest path from each word to “moke". How might a spell checker use these values?

35. The graph below contains two vertices of odd degree. To eulerize this graph, it is necessary to duplicate edges connecting those two vertices.

(a) Use Dijkstra’s algorithm to find the shortest path between the two vertices with odd degree. Does this produce the most efficient eulerization and solve the Chinese Postman Problem for this graph?



(b) Suppose a graph has n odd vertices. Using the approach from part a, how many shortest paths would need to be considered? Is this approach going to be efficient?

8.7.1. 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

Notes, References, and Resources

Resources

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

Part III

Integer Programming

Part II: Integer Programming

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

9. Integer Programming Formulations

Chapter 9. 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.*

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.

9.1 Knapsack Problem

The *knapsack problem* can take different forms depending on if the variables are binary or integer. The binary version means that there is only one item of each item type that can be taken. This is typically illustrated as a backpack (knapsack) and some items to put into it (see Figure 16.1), but has applications in many contexts.

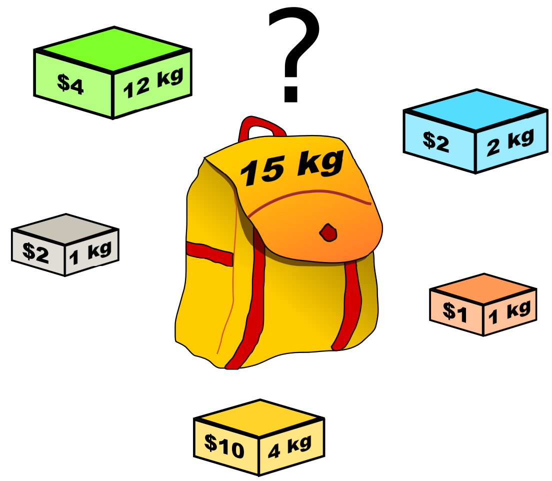
Binary Knapsack Problem:

NP-Complete

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

$$\begin{aligned} \max \ & c^\top x \\ \text{s.t.} \ & a^\top x \le b \\ & x \in \{0,1\}^n \end{aligned} \tag{9.1}$$

[1]**wiki/File/knapsack**, from **wiki/File/knapsack**. **wiki/File/knapsack**, **wiki/File/knapsack**.



© wiki/File/knapsack[1]

Figure 9.1: Knapsack Problem: which items should we choose take in the knapsack that maximizes the value while respecting the 15kg weight limit?

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 16.1.

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:

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

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

$$\begin{aligned} \max \ & c^\top x \\ \text{s.t.} \ & a^\top x \leq b \\ & x \in \mathbb{Z}^n_+ \end{aligned} \tag{9.2}$$

We can also consider an equality constrained version

Equality Constrained Integer Knapsack Problem:

NP-Hard

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

$$\max \ c^\top x \tag{9.3}$$

$$\text{s.t.} \ a^\top x = b \tag{9.4}$$

$$x \in \mathbb{Z}^n_+ \tag{9.5}$$

Example 9.1:

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

Variables:

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

Model

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

9.2 Capital Budgeting

Section 9.2. Capital Budgeting

The *capital budgeting* problem is a nice generalization of the knapsack problem. This problem as 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 it's expected return while satisfying it's weekly budget constriaints?

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 invesment 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:

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

Consider the example given in the following table.

Project	$\mathbb{E}$[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 "$\mathbb{E}$[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 explict model is given by
Model:

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

9.3 Set Covering

Section 9.3. Set Covering

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

Set Covering:

NP-Complete

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

The set cover problem can be modeled as

$$\begin{aligned} \min \quad & 1^\top x \\ \text{s.t.} \quad & \sum_{v \in V_i} x_v \geq 1 \text{ for all } i = 1, \dots, l \\ & x_v \in \{0,1\} \text{ for all } v \in V \end{aligned} \tag{9.1}$$

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.

Add flight crew scheduling example and images.

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 "$\mathbb{E}$[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 explict model is given by
Model:

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

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

Example: Vertex Cover:

NP-Complete

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

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

$$\begin{aligned}
\min \quad & 1^\top x\\
\text{s.t.} \quad & x_u + x_v \ge 1 \text{ for all } (u,v) \in E\\
& x_v \in \{0,1\} \text{ for all } v \in V.
\end{aligned} \tag{9.2}$$

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 constains a fire station. Figure 16.2 depicts the set of districts and an example placement of locations of fire stations. How can we minimize the 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. $V_1 = \{2,4,5\}$).

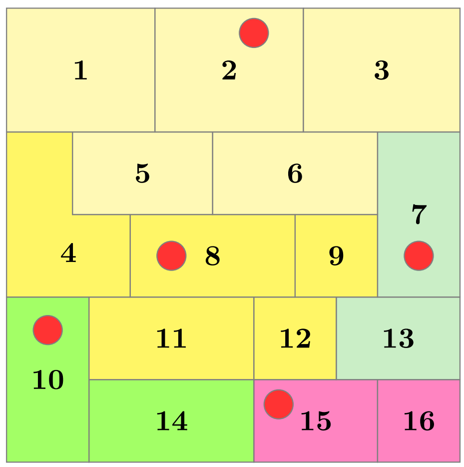
Variables:

- 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.pdf**[2]

Figure 9.2: Layout of districts and possible locations of fire stations.

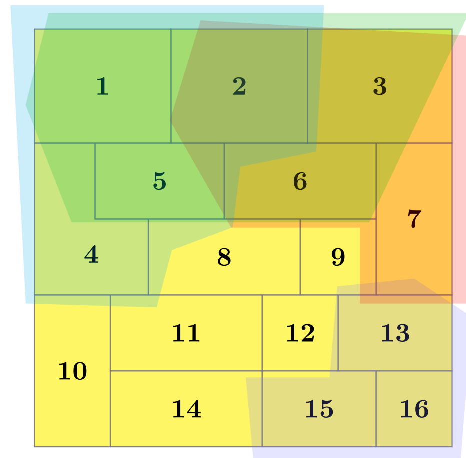
Set Covering - Matrix description:

NP-Complete

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

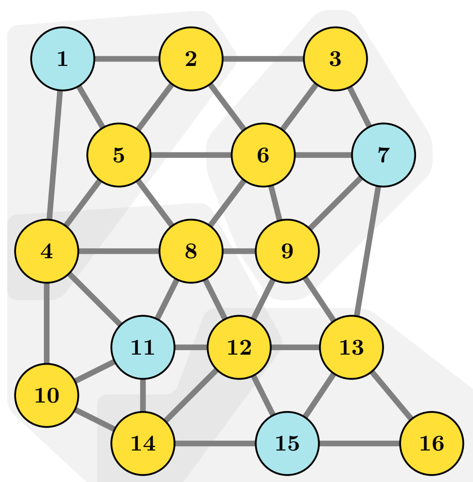
$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \geq 1 \\ & x \in \{0,1\}^n. \end{aligned} \tag{9.3}$$

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



© tikz/Illustration2.pdf[3]

Figure 9.3: 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.pdf[4]

Figure 9.4: Graph representation of fire station problem. Every node is connected to a chosen node by an edge

Example: Vertex Cover with matrix

An alternate way to solve **??** 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.

9.3.1. Covering (Generalizing Set Cover)

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

Covering:

NP-Complete

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

$$\begin{aligned} \max \ & c^\top x \\ \text{s.t..} \ & Ax \geq b \\ & x \in \mathbb{Z}_+^n. \end{aligned} \tag{9.4}$$

9.4 Assignment Problem

Section 9.4. 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:

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

9.5 Facility Location

Section 9.5. Facility Location

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

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

Add discussion on Facility Location Problems and pictures.

9.5.1. Capacitated Facility Location

Capacitated Facility Location:

NP-Complete

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

Sets:

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

Parameters:

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

Variables:

- Let
$$y_i = \begin{cases} 1 & \text{if we open facility } i, \\ 0 & \text{otherwise.} \end{cases}$$
- Let $x_{ij} \geq 0$ be the fraction of demand of customer j satisfied by facility i.

Model:

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

Alternative model!

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

9.5.2. 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

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}z_{ij} + \sum_{i=1}^{n} f_i x_i \\
\text{s.t.} \quad & \sum_{i=1}^{n} z_{ij} = 1 \text{ for all } j = 1,\dots,m \\
& \sum_{j=1}^{m} z_{ij} \leqslant M x_i \text{ for all } i = 1\dots,n \\
& z_{ij} \in \{0,1\} \text{ for all } i = 1,\dots,n \text{ and } j = 1,\dots,m \\
& x_i \in \{0,1\} \text{ for all } i = 1,\dots,n
\end{aligned}
\tag{9.1}
$$

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

Alternative model!

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij} z_{ij} + \sum_{i=1}^{n} f_i x_i \\
\text{s.t.} \quad & \sum_{i=1}^{n} z_{ij} = 1 \text{ for all } j = 1,\dots,m \\
& z_{ij} \leqslant x_i \text{ for all } i = 1\dots,n \text{ for all } j = 1,\dots,m \\
& z_{ij} \in \{0,1\} \text{ for all } i = 1,\dots,n \text{ and } j = 1,\dots,m \\
& x_i \in \{0,1\} \text{ for all } i = 1,\dots,n
\end{aligned}
\tag{9.2}
$$

9.6 Basic Modeling Tricks - Using Binary Variables

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

Binary variables can allow you to model many types of constraints. We discuss here varios 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.

$$x_i \leq x_j \tag{9.1}$$

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

$$x_1 + x_2 + \cdots + x_5 \leq 5 \cdot x_6 \tag{9.2}$$

   Alternatively!

$$x_i \leq x_6 \quad \text{for all } i = 1,\dots,5 \tag{9.3}$$

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

$$x_i \leq x_j \tag{9.4}$$

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

$$x_1 + x_2 + \cdots + x_i \leq i \cdot x_j \tag{9.5}$$

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

$$x_i \leq x_j \tag{9.6}$$

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

$$x_i + x_j = 1 \tag{9.7}$$

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

$$x_i + x_j \geq 1 \tag{9.8}$$

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

$$x_j \leq (1 - x_i) \tag{9.9}$$

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

$$x_i + x_j + x_k \leq 1 \tag{9.10}$$

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

$$x_i + x_j + x_k \leq 2 \tag{9.11}$$

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

$$x_i + x_j + x_k = 1 \tag{9.12}$$

These tricks can be connected to create different function values.

Example 9.2: 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 9.3: 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 ith unit. The inequality constraints impose that if we buy unit j, then we must by all units i with $i < j$.

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

9.6.1. Big M constraints - Activating/Deactivating Inequalities

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

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

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

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

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

Then we want to say

If $\delta = 0$, then

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

We can formulate this using a big-M constraint as

$$x_A + x_B + x_C + x_D + x_E \leq 3 + M\delta. \tag{9.13}$$

Notice the two case

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

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

We can speak about this technique more generally as

Big-M: If then:

We aim to model the relationship

$$\text{If} \quad \delta = 0, \quad \text{then} \quad a^\top x \leq b. \tag{9.14}$$

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

$$\begin{aligned} a^\top x - b &\leq M\delta \\ \delta &\in \{0,1\} \end{aligned} \tag{9.15}$$

9.6.2. Either Or Constraints

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

Either Or:

$$\text{Either} \quad a^\top x \leq b \quad \text{or} \quad c^\top x \leq d \quad \text{holds} \tag{9.16}$$

can be modeled as

$$\begin{aligned} a^\top x - b &\leq M_1 \delta \\ c^\top x - d &\leq M_2(1-\delta) \\ \delta &\in \{0,1\}, \end{aligned} \tag{9.17}$$

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

Example 9.4

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

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

Suppose that there are at most $M_1 = 5$ buses that could be rented an at most $M_2 = 20$ cars that could be available.
This constraint can be modeled as

$$\begin{aligned} x - 2 &\leq 5\delta \\ y - 10 &\leq 20(1-\delta) \\ \delta &\in \{0,1\}, \end{aligned} \tag{9.18}$$

9.6.3. If then implications - opposite direction

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

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

Thus, we want to model

If
$$\sum_{i \in I} x_i \leq 10$$
then
$$\delta = 1.$$

We can model this as
$$\sum_{i \in I} x_i \geq 10 + 1 + M\delta. \tag{9.19}$$

If inequality, then indicator:

W

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

Now
$$\text{If} \quad a^\top x \leq b \quad \text{then} \quad \delta = 1 \tag{9.20}$$
can be modeled as
$$a^\top x - b \geq \varepsilon(1 - \delta) + m\delta. \tag{9.21}$$

Proof. We not justify the statement above.

A simple way to understand this constraint is to consider the *contrapositive* of the if then statement that we want to model. The contrapositive says that
$$\text{If } \delta = 0, \text{ then } a^\top x - b > 0. \tag{9.22}$$

To show the contrapositive, we set $\delta = 0$. Then the inequality becomes
$$a^\top x - b \geq \varepsilon(1 - 0) + m0 = \varepsilon > 0.$$

Thus, the contrapositive holds.

If instead we wanted a direct proof:

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

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

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

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

Table 9.1: Short list: If/then models with a constraint and a binary variable. Here M and m are upper and lower bounds on $a^\top x - b$ and ε is a small number such that if $a^\top x > b$, then $a^\top x \ge b + \varepsilon$.

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

Table 9.2: Long list: If/then models with a constraint and a binary variable. Here M and m are upper and lower bounds on $a^\top x - b$ and ε is a small number such that if $a^\top x > b$, then $a^\top x \ge b + \varepsilon$.

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

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

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

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

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

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

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

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

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

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

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

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

9.6.4. Multi Term Disjunction with application to 2D packing

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

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

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

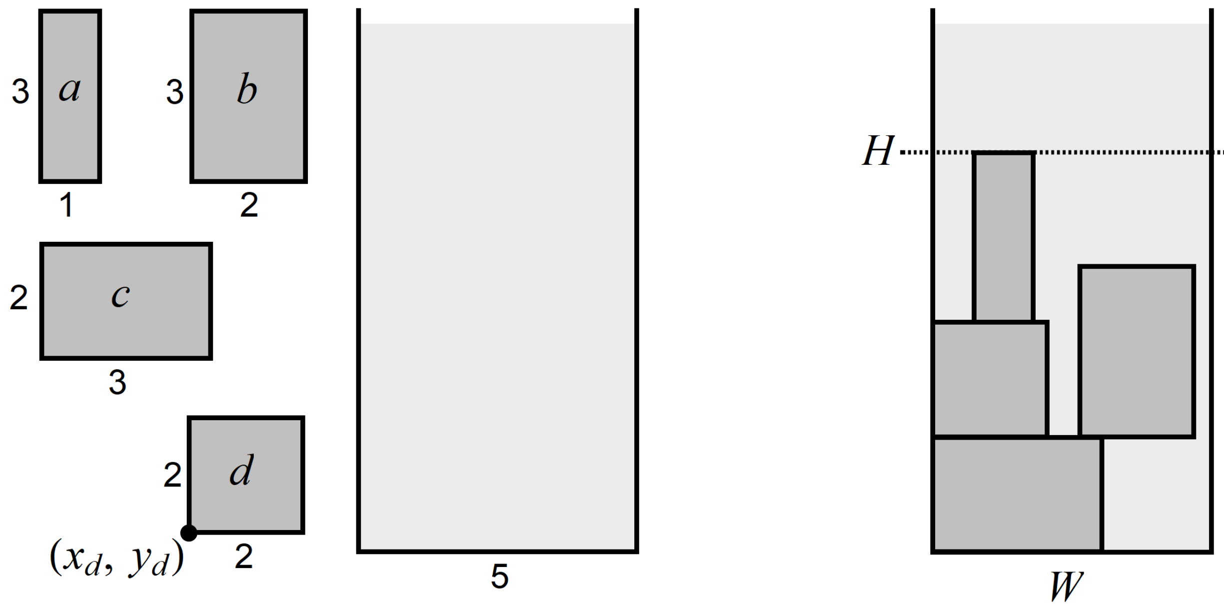
$$\begin{aligned} \mathbf{a}_1^\top \mathbf{x} &\le b_1 + M(1 - \delta_1) \\ \mathbf{a}_2^\top \mathbf{x} &\le b_2 + M(1 - \delta_2) \\ &\vdots \\ \mathbf{a}_n^\top \mathbf{x} &\le b_n + M(1 - \delta_1) \\ \sum_{i=1}^n \delta_i &\ge k \end{aligned}$$

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

9.6.4.1. Strip Packing Problem

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

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



(a) Pack the rectangles into the strip **(b) Minimize overall height H.**

Figure 9.5: An Example of the Strip Packing Problem

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

$$
\begin{aligned}
\text{Minimize} \quad & \max_{i \in I} \{y_i + h_i\} && (9.23a) \\
\text{s.t.} \quad & \max_{i \in I} \{x_i + w_i\} \leq W && (9.23b) \\
& \left.\begin{aligned} x_i + w_i &\leq x_j \\ x_j + w_j &\leq x_i \\ y_i + h_i &\leq y_j \\ y_j + h_j &\leq y_i \end{aligned}\right\} \begin{array}{l}\text{At least one of these} \\ \text{for every distinct pairs} \\ \text{of rectangles } i, j \in I\end{array} && (9.23c) \\
& x_i, y_i \geq 0 \qquad \forall\, i \in I && (9.23d)
\end{aligned}
$$

Constraint (**??**) is a set of four *disjunctive* constraints. This can be expressed linearly by introducing a new binary indicator variable δ_{ijk} for each of the disjunctive constraints in (**??**):

$$
\begin{array}{rlll}
\text{Minimize} & H & & \\
\text{s.t.} & y_i + h_i \le H & \forall\, i \in I & (\mathbf{??}) \\
& x_i + w_i \le W & \forall\, i \in I & (\mathbf{??}) \\
& x_i + w_i \le x_j + M(1-\delta_{ij1}) & \forall\, i,j \in I:\ i > j & \\
& x_j + w_j \le x_i + M(1-\delta_{ij2}) & \forall\, i,j \in I:\ i > j & \\
& y_i + h_i \le y_j + M(1-\delta_{ij3}) & \forall\, i,j \in I:\ i > j & (\mathbf{??}) \\
& y_j + h_j \le y_i + M(1-\delta_{ij4}) & \forall\, i,j \in I:\ i > j & \\
& \sum_{k=1}^{4} \delta_{ijk} \ge 1 & \forall\, i,j \in I:\ i > j & \\
& x_i, y_i \ge 0 & \forall\, i \in I & \\
& \delta_{ij} \in \{0,1\}^4 & \forall\, i,j \in I:\ i > j &
\end{array}
$$

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

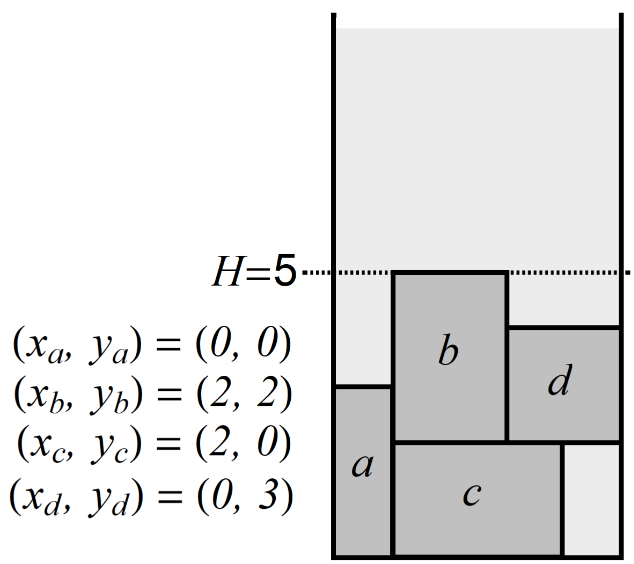


Figure 9.6: The Optimal Solution to the example problem

This problem is considered strongly NP-Hard.

9.6.5. SOS1 Constraints

Definition 9.5: Special Ordered Sets of Type 1 (SOS1)

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

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

Example: SOS1 Constraints Gurobipy

Solve the following optimization problem:

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

9.6.6. SOS2 Constraints

Definition 9.6: Special Ordered Sets of Type 2 (SOS2)

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

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

Example: SOS2 Gurobipy

Solve the following optimization problem:

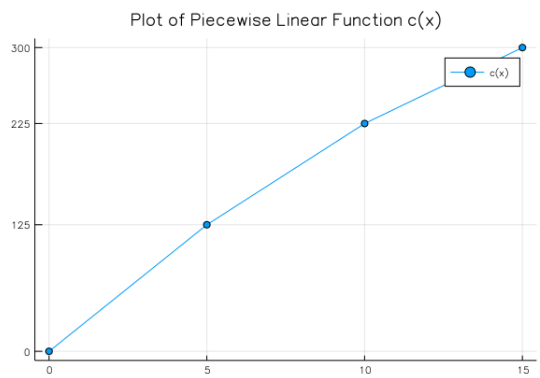
$$\begin{array}{rl} \text{maximize} & 3x_1 + 4x_2 + x_3 + 5x_4 \\ \text{subject to} & 0 \leq x_i \leq 5 \\ & \text{at most two of the } x_i \text{ can be nonzero} \\ & \text{and the nonzero } x_i \text{ must be consecutive} \end{array}$$

9.6.7. Piecewise linear functions with SOS2 constraint

Example: Piecewise Linear Function Gurobipy

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

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



© **pwl-plot.png**[5]

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

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

[5]**pwl-plot.png**, from **pwl-plot.png**. **pwl-plot.png**, **pwl-plot.png**.

Example: Piecewise Linear Function Application Gurobipy

Consider the following optimization problem where the objective function includes the term $c(x)$, where $c(x)$ is the piecewise linear function described in Example 18:

$$\max z = 12x_{11} + 12x_{21} + 14x_{12} + 14x_{22} - c(x) \tag{9.24}$$

$$\text{s.t.} x_{11} + x_{12} \leq x + 5 \tag{9.25}$$

$$x_{21} + x_{22} \leq 10 \tag{9.26}$$

$$0.5x_{11} - 0.5x_{21} \geq 0 \tag{9.27}$$

$$0.4x_{12} - 0.6x_{22} \geq 0 \tag{9.28}$$

$$x_{ij} \geq 0 \tag{9.29}$$

$$0 \leq x \leq 15 \tag{9.30}$$

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

$$\begin{array}{rrll}
\max & 12X_{1,1} + 12X_{2,1} + 14X_{1,2} + 14X_{2,2} - y & & \\
\text{Subject to} & x - 5z_2 - 10z_3 - 15z_4 & = 0 & \\
& y - 125z_2 - 225z_3 - 300z_4 & = 0 & \\
& z_1 + z_2 + z_3 + z_4 & = 1 & \\
& X_{1,1} + X_{1,2} - x & \leq 5 & \\
& X_{2,1} + X_{2,2} \leq 10 & & \\
& 0.5X_{1,1} - 0.5X_{2,1} & \geq 0 & \\
& 0.4X_{1,2} - 0.6X_{2,2} & \geq 0 & \\
& SOS2 : \{z_1, z_2, z_3, z_4\} & & \\
& X_{i,j} & \geq 0 & \forall i \in \{1,2\}, j \in \{1,2\} \\
& 0 \leq z_i & \leq 1 & \forall i \in \{1,2,3,4\} \\
& 0 \leq x & \leq 15 & \\
& y & \text{free} &
\end{array} \tag{9.31}$$

9.6.7.1. SOS2 with binary variables

Modeling Piecewise linear function

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

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

9.6.8. Maximizing a minimum

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

Consider the problem

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

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

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

9.6.9. Relaxing (nonlinear) equality constraints

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

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

9.6.10. Connecting to continuous variables

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

1. If $x_i > 0$, then $y_i = 1$.
$$x_i \leq M y_i \tag{9.32}$$
where M is a suffiently 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
$$x_i - 0.0000001 \geq y_i - 1. \tag{9.33}$$
3. If $y_i = 1$, then $x_i \geq 5$
$$5 y_i \leq x_i. \tag{9.34}$$

9.6.11. 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.

9.6.11.1. 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:

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

9.6.11.2. Maximum

The exact equality $t = \max\{x_1, \ldots, x_n\}$ can be expressed by introducing a sequence of mutually exclusive indicator variables $z_1, \ldots, z_n$, 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:

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

9.7 Network Flow

Section 9.7. Network Flow

Fix up this section

9.7.1. 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 $K_1, K_2, \ldots, K_k$, 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.

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

(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.

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

9.7.2. 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 $\sum_{i=1}^{k} d_i$

9.7.3. 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

9.7.4. Usage

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

9.8 Transportation Problem

Section 9.8. Transportation Problem

Add discussion of transportation problem and picture.

Youtube! - TRANSPORTATION PROBLEM with PuLP in PYTHON

Notebook: Solution with Pyomo

9.9 Job Shop Scheduling

Section 9.9. Job Shop Scheduling

Pyhton MIP example

Fill in model and discussion and add code example. Need to create gnat chart code for nice visulizations.

9.10 Jobshop Scheduling: Makespan Minimization

Add discussion of some makespan minimization problems.

Wikipedia: Jobshop Scheduling

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

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

$$\text{min:} \tag{9.1}$$
$$C \tag{9.2}$$
$$\text{s.t.:} \tag{9.3}$$
$$x_{o_r^j j} \geq x_{o_{r-1}^j j} + p_{o_{r-1}^j j} \ \forall r \in \{2,..,m\}, j \in J \tag{9.4}$$
$$x_{ij} \geq x_{ik} + p_{ik} - M \cdot y_{ijk} \ \forall j,k \in J, j \neq k, i \in M \tag{9.5}$$
$$x_{ik} \geq x_{ij} + p_{ij} - M \cdot (1 - y_{ijk}) \ \forall j,k \in J, j \neq k, i \in I \tag{9.6}$$
$$C \geq x_{o_m^j j} + p_{o_m^j j} \ \forall j \in J \tag{9.7}$$
$$x_{ij} \geq 0 \ \forall i \in J, i \in I \tag{9.8}$$
$$y_{ijk} \in \{0,1\} \ \forall j,k \in J, i \in I \tag{9.9}$$
$$C \geq 0 \tag{9.10}$$

9.11 Quadratic Assignment Problem (QAP)

Resources

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

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

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

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

Quadratic Assignment Problem:

NP-Complete

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

Sets:

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

Parameters:

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

Variables:

- Let

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

Model:

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

9.12 Genaralized Assignment Problem (GAP)

Fix up this section

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

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

9.12.1. In special cases

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

9.12.2. Explanation of definition

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

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

$$\text{maximize} \sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij}x_{ij}. \tag{9.1}$$

$$\text{subject to} \sum_{j=1}^{n} w_{ij}x_{ij} \leq t_i \qquad i = 1,\ldots,m; \tag{9.2}$$

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad j = 1,\ldots,n; \tag{9.3}$$

$$x_{ij} \in \{0,1\} \qquad i = 1,\ldots,m, \quad j = 1,\ldots,n; \tag{9.4}$$

9.13 Other material

9.13.1. Binary reformulation of integer variables

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

Full reformulation:

u many binary variables

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

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \tag{9.1}$$

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

$$\begin{aligned} x &= \sum_{i=1}^{u} i z_i = z_1 + 2z_2 + \cdots + u z_u \\ 1 &\geq \sum_{i=1}^{u} z_i = z_1 + z_2 + \cdots + z_u \\ z_i &\in \{0,1\} \quad \text{for } i = 1, \dots, u \end{aligned} \tag{9.2}$$

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

Binary reformulation:

$O(\log u)$ many binary variables

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

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \tag{9.3}$$

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

$$\begin{aligned} x &= \sum_{i=0}^{\lfloor \log(u) \rfloor} 2^i z_i = z_0 + 2z_1 + 4z_2 + 8z_3 + \cdots + 2^{\lfloor \log(u) \rfloor} z_{\lfloor \log(u) \rfloor} \\ z_i &\in \{0,1\} \quad \text{for } i = 1, \dots, \lfloor \log(u) \rfloor \end{aligned} \tag{9.4}$$

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

9.14 Literature and Resources

Resources

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

Knapsack Problem

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

Set Cover

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

Facility Location

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

Other examples

- *Sudoku*
- *AIMMS - Employee Training*
- *AIMMS - Media Selection*
- *AIMMS - Diet Problem*
- *AIMMS - Farm Planning Problem*
- *AIMMS - Pooling Probem*
- *INFORMS - Impact*
- *INFORMS - Success Story - Bus Routing*

Notes from AIMMS modeling book.

- *AIMMS - Practical guidelines for solving difficult MILPs*
- *AIMMS - Linear Programming Tricks*
- *AIMMS - Formulating Optimization Models*
- *AIMMS - Practical guidelines for solving difficult linear programs*

Modeling Tricks

- *JuMP tips and tricks*
- *Mosek Modeling Cookbook*

Further Topics

- *Precedence Constraints*

10. Integer Programming Formulations

Outcomes

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

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.

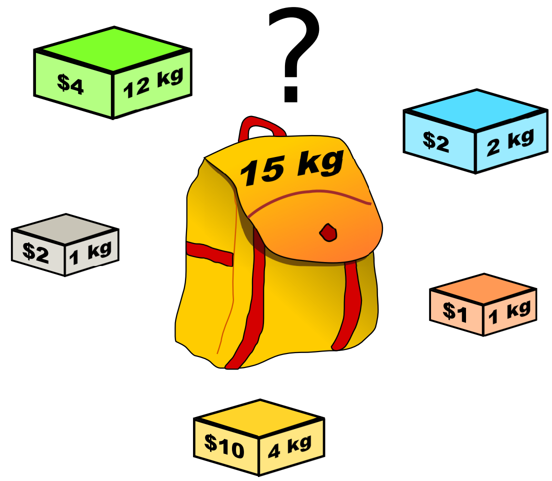
10.1 Knapsack Problem

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 16.1), but has applications in many contexts.

Binary Knapsack Problem:

NP-Complete

[1]Video! - Michel Belaire (EPFL) teaching knapsack problem
[2]**wiki/File/knapsack**, from **wiki/File/knapsack**. **wiki/File/knapsack**, **wiki/File/knapsack**.



© wiki/File/knapsack[2]

Figure 10.1: Knapsack Problem: which items should we choose take in the knapsack that maximizes the value while respecting the 15kg weight limit?

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

$$\begin{aligned} \max \ & c^\top x \\ \text{s.t.} \ & a^\top x \leq b \\ & x \in \{0,1\}^n \end{aligned} \tag{10.1}$$

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 16.1.

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:

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

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

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & a^\top x \leq b \\ & x \in \mathbb{Z}^n_+ \end{aligned} \tag{10.2}$$

We can also consider an equality constrained version

Equality Constrained Integer Knapsack Problem:

NP-Hard

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

$$\max \quad c^\top x \tag{10.3}$$

$$\text{s.t.} \quad a^\top x = b \tag{10.4}$$

$$x \in \mathbb{Z}^n_+ \tag{10.5}$$

Example 10.1:

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

Variables:

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

Model

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

10.2 Capital Budgeting

The *capital budgeting* problem is a nice generalization of the knapsack problem. This problem as 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 it's expected return while satisfying it's weekly budget constriaints?

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 invesment 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:

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

Consider the example given in the following table.

Project	$\mathbb{E}$[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 "$\mathbb{E}$[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 explict model is given by
Model:

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

10.3 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

$$\begin{aligned} \min \quad & \mathbb{1}^\top x \\ \text{s.t.} \quad & \sum_{v \in V_i} x_v \geq 1 \text{ for all } i = 1,\dots,l \\ & x_v \in \{0,1\} \text{ for all } v \in V \end{aligned} \tag{10.1}$$

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:

$$\begin{aligned} \min \quad & 1^\top x \\ \text{s.t.} \quad & x_u + x_v \geq 1 \text{ for all } (u,v) \in E \\ & x_v \in \{0,1\} \text{ for all } v \in V. \end{aligned} \tag{10.2}$$

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 constains a fire station. Figure 16.2 depicts the set of districts and an example placement of locations of fire stations. How can we minimize the 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.

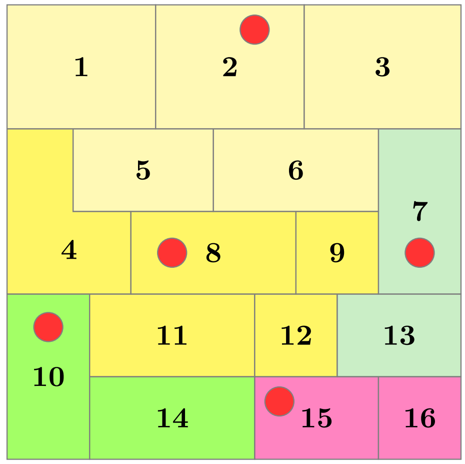
Model:

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

Set Covering - Matrix description:

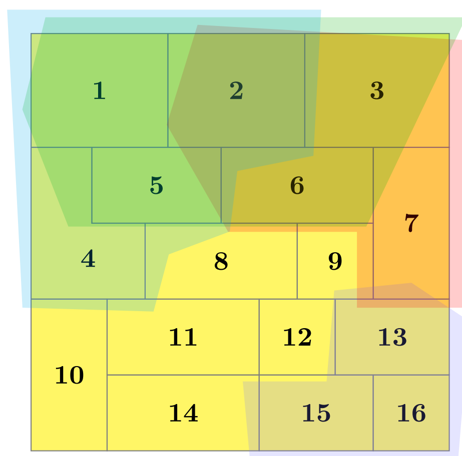
NP-Complete

[3]**tikz/Illustration1.pdf**, from **tikz/Illustration1.pdf**. **tikz/Illustration1.pdf**, **tikz/Illustration1.pdf**.
[4]**tikz/Illustration2.pdf**, from **tikz/Illustration2.pdf**. **tikz/Illustration2.pdf**, **tikz/Illustration2.pdf**.
[5]**tikz/Illustration3.pdf**, from **tikz/Illustration3.pdf**. **tikz/Illustration3.pdf**, **tikz/Illustration3.pdf**.



© tikz/Illustration1.pdf[3]

Figure 10.2: Layout of districts and possible locations of fire stations.

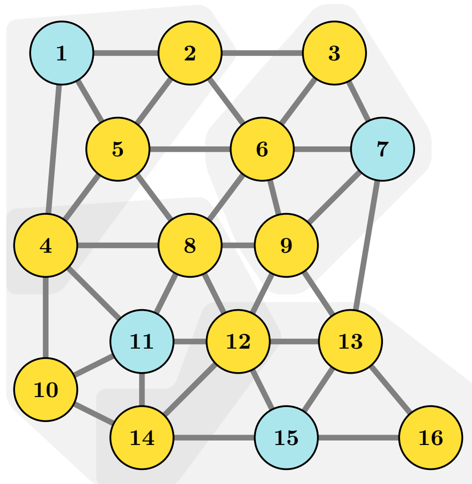


© tikz/Illustration2.pdf[4]

Figure 10.3: Set cover representation of fire station problem. For example, choosing district 16 to have a fire station covers districts 13, 15, and 16.

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

$$\begin{aligned} \max \;\; & c^\top x \\ \text{s.t.. } \;\; & Ax \geq 1 \\ & x \in \{0,1\}^n. \end{aligned} \tag{10.3}$$



© tikz/Illustration3.pdf[5]

Figure 10.4: Graph representation of fire station problem. Every node is connected to a chosen node by an edge

Example: Vertex Cover with matrix

An alternate way to solve **??** 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.

10.3.1. 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

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t..} \quad & Ax \geq b \\ & x \in \mathbb{Z}_+^n. \end{aligned} \tag{10.4}$$

10.4 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:

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

10.5 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*.

10.5.1. 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:

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

10.5.2. 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

$$\begin{aligned} \min \quad & \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij} z_{ij} + \sum_{i=1}^{n} f_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} z_{ij} = 1 \text{ for all } j = 1,\dots,m \\ & \sum_{j=1}^{m} z_{ij} \leqslant M x_i \text{ for all } i = 1\dots,n \\ & z_{ij} \in \{0,1\} \text{ for all } i = 1,\dots,n \text{ and } j = 1,\dots,m \\ & x_i \in \{0,1\} \text{ for all } i = 1,\dots,n \end{aligned} \tag{10.1}$$

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

10.6 Basic Modeling Tricks - Using Binary Variables

Binary variables can allow you to model many types of constraints. We discuss here varios 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.

$$x_i \leq x_j \tag{10.1}$$

   (a) If any of items $1,\dots,i$ are selected, then item j is selected.

$$x_1 + x_2 + \cdots + x_i \leq i \cdot x_j \tag{10.2}$$

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

$$x_i \leq x_j \tag{10.3}$$

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

$$x_1 + x_2 + \cdots + x_i \leq i \cdot x_j \tag{10.4}$$

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

$$x_i \leq x_j \tag{10.5}$$

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

$$x_i + x_j = 1 \tag{10.6}$$

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

$$x_i + x_j \geq 1 \tag{10.7}$$

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

$$x_j \leq (1 - x_i) \tag{10.8}$$

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

$$x_i + x_j + x_k \leq 1 \tag{10.9}$$

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

$$x_i + x_j + x_k \leq 2 \tag{10.10}$$

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

$$x_i + x_j + x_k = 1 \tag{10.11}$$

These tricks can be connected to create different function values.

Example 10.2: 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 10.3: 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 ith unit. The inequality constraints impose that if we buy unit j, then we must by all units i with $i < j$.

10.7 Network Flow

Fix up this section

10.7.1. 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 $K_1, K_2, \ldots, K_k$, 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.

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

(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.

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

10.7.2. 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 $\sum_{i=1}^{k} d_i$

10.7.3. 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]

10.7.4. Usage

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

10.8 Transportation Problem

Add discussion of transportation problem and picture.

Youtube! - TRANSPORTATION PROBLEM with PuLP in PYTHON

Notebook: Solution with Pyomo

10.9 Jobshop Scheduling: Makespan Minimization

Add discussion of some makespan minimization problems.

Wikipedia: Jobshop Scheduling

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

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

6

$$\text{min:} \tag{10.1}$$
$$C \tag{10.2}$$
$$\text{s.t.:} \tag{10.3}$$
$$x_{o_r^j j} \geq x_{o_{r-1}^j j} + p_{o_{r-1}^j j} \ \forall r \in \{2,..,m\}, j \in J \tag{10.4}$$
$$x_{ij} \geq x_{ik} + p_{ik} - M \cdot y_{ijk} \ \forall j,k \in J, j \neq k, i \in M \tag{10.5}$$
$$x_{ik} \geq x_{ij} + p_{ij} - M \cdot (1 - y_{ijk}) \ \forall j,k \in J, j \neq k, i \in I \tag{10.6}$$
$$C \geq x_{o_m^j j} + p_{o_m^j j} \ \forall j \in J \tag{10.7}$$
$$x_{ij} \geq 0 \ \forall i \in J, i \in I \tag{10.8}$$
$$y_{ijk} \in \{0,1\} \ \forall j,k \in J, i \in I \tag{10.9}$$
$$C \geq 0 \tag{10.10}$$

10.10 Quadratic Assignment Problem (QAP)

Resources

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

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

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

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

Quadratic Assignment Problem:

NP-Complete

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

Sets:

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

Parameters:

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

Variables:

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

Model:

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

10.11 Genaralized Assignment Problem (GAP)

Fix up this section

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

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

10.11.1. In special cases

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

10.11.2. Explanation of definition

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

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

$$\text{maximize} \sum_{i=1}^{m}\sum_{j=1}^{n} p_{ij}x_{ij}. \tag{10.1}$$

$$\text{subject to} \sum_{j=1}^{n} w_{ij}x_{ij} \le t_i \qquad i = 1,\ldots,m; \tag{10.2}$$

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad j = 1,\ldots,n; \tag{10.3}$$

$$x_{ij} \in \{0,1\} \qquad i = 1,\ldots,m, \quad j = 1,\ldots,n; \tag{10.4}$$

10.12 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

10.13 Modeling Tricks

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.

10.13.1. Either Or Constraints

"At least one of these constraints holds" is what we would like to model. Equivalently, we can phrase this as an *inclusive or* constraint.

Either Or:

$$\text{Either} \quad a^\top x \leq b \quad \text{or} \quad c^\top x \leq d \quad \text{holds} \tag{10.1}$$

can be modeled as

$$\begin{aligned} a^\top x - b &\leq M_1 \delta \\ c^\top x - d &\leq M_2(1-\delta) \\ \delta &\in \{0,1\}, \end{aligned} \tag{10.2}$$

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

Example 10.4

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

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

Suppose that there are at most $M_1 = 5$ buses that could be rented an at most $M_2 = 20$ cars that could be available.
This constraint can be modeled as

$$\begin{aligned} x - 2 &\leq 5\delta \\ y - 10 &\leq 20(1-\delta) \\ \delta &\in \{0,1\}, \end{aligned} \tag{10.3}$$

10.13.2. If then implications

If then implications are extremely useful in models. For instance, if we have more than 5 passengers, then we need to take two cars. Most if then statements can be modeled with by a constraint and an on/off flag. For example

$$\text{If} \quad \delta = 1, \text{then} \quad a^\top x \le b. \tag{10.4}$$

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

$$\begin{aligned} a^\top x - b &\le M(1-\delta) \\ \delta &\in \{0,1\} \end{aligned} \tag{10.5}$$

On the other hand, if we want to model the reverse implication, we have to be slightly more careful. We let m be a lower bound on the quantity $a^\top x - b$ and we let ε be a tiny number that is an error bound in verifying if an inequality is violated. **If the data a, b are integer and x is an integer, then we can take $\varepsilon = 1$.**

Now

$$\text{If} \quad a^\top x \le b \ \text{ then } \ \delta = 1 \tag{10.6}$$

can be modeled as

$$a^\top x - b \ge \varepsilon(1-\delta) + m\delta. \tag{10.7}$$

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

$$\text{If } \delta = 0, \text{ then } a^\top x - b > 0. \tag{10.8}$$

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

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

Thus, the contrapositive holds.

If instead we wanted a direct proof:

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

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

Therefore

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

After rearranging

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

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

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

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

Table 10.1: Short list: If/then models with a constraint and a binary variable. Here M and m are upper and lower bounds on $a^\top x - b$ and ε is a small number such that if $a^\top x > b$, then $a^\top x \geq b + \varepsilon$.

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

Table 10.2: Long list: If/then models with a constraint and a binary variable. Here M and m are upper and lower bounds on $a^\top x - b$ and ε is a small number such that if $a^\top x > b$, then $a^\top x \geq b + \varepsilon$.

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

—-

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

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

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

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

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

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

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

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

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

10.13.3. Binary reformulation of integer variables

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

Full reformulation:

u many binary variables

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

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \tag{10.9}$$

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

$$\begin{aligned} x &= \sum_{i=1}^{u} i z_i = z_1 + 2z_2 + \cdots + u z_u \\ 1 &\geq \sum_{i=1}^{u} z_i = z_1 + z_2 + \cdots + z_u \\ z_i &\in \{0,1\} \quad \text{for } i = 1, \ldots, u \end{aligned} \tag{10.10}$$

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

Binary reformulation:

$O(\log u)$ many binary variables

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

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \tag{10.11}$$

this can be reformulated with u binary variables $z_1, \ldots, z_{\log(\lfloor u \rfloor)+1}$ as

$$\begin{aligned} x &= \sum_{i=0}^{\log(\lfloor u \rfloor)+1} 2^i z_i = z_0 + 2z_1 + 4z_2 + 8z_3 + \cdots + 2^{\log(\lfloor u \rfloor)+1} z_{\log(\lfloor u \rfloor)+1} \\ z_i &\in \{0,1\} \quad \text{for } i = 1, \ldots, \log(\lfloor u \rfloor) + 1 \end{aligned} \tag{10.12}$$

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

10.13.4. SOS1 Constraints

Definition 10.5: Special Ordered Sets of Type 1 (SOS1)

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

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

Example: SOS1 Constraints Gurobipy

Solve the following optimization problem:

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

10.13.5. SOS2 Constraints

Definition 10.6: Special Ordered Sets of Type 2 (SOS2)

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

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

Example: SOS2 Gurobipy

Solve the following optimization problem:

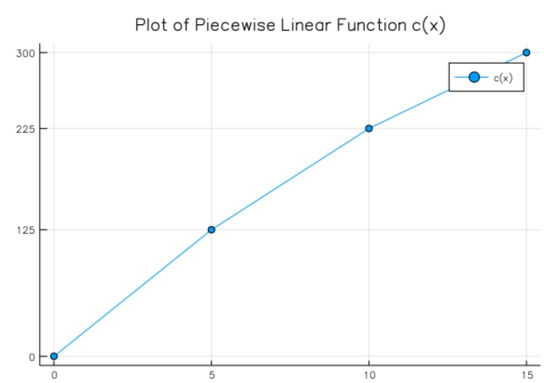
$$\begin{array}{rl} \text{maximize} & 3x_1 + 4x_2 + x_3 + 5x_4 \\ \text{subject to} & 0 \le x_i \le 5 \\ & \text{at most two of the } x_i \text{ can be nonzero} \\ & \text{and the nonzero } x_i \text{ must be consecutive} \end{array}$$

10.13.6. Piecewise linear functions with SOS2 constraint

Example: Piecewise Linear Function Gurobipy

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

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



© **pwl-plot.png**[7]

We will use integer programming to describe this function. We will fix $x = a$ and then the integer program

[7] **pwl-plot.png**, from **pwl-plot.png**. **pwl-plot.png**, **pwl-plot.png**.

will set the value y to $c(a)$.

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

Example: Piecewise Linear Function Application Gurobipy

Consider the following optimization problem where the objective function includes the term $c(x)$, where $c(x)$ is the piecewise linear function described in Example 18:

$$\max z = 12x_{11} + 12x_{21} + 14x_{12} + 14x_{22} - c(x) \tag{10.13}$$
$$\text{s.t.} x_{11} + x_{12} \leq x + 5 \tag{10.14}$$
$$x_{21} + x_{22} \leq 10 \tag{10.15}$$
$$0.5x_{11} - 0.5x_{21} \geq 0 \tag{10.16}$$
$$0.4x_{12} - 0.6x_{22} \geq 0 \tag{10.17}$$
$$x_{ij} \geq 0 \tag{10.18}$$
$$0 \leq x \leq 15 \tag{10.19}$$

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

$$\begin{array}{rrll}
\max & 12X_{1,1} + 12X_{2,1} + 14X_{1,2} + 14X_{2,2} - y & & \\
\text{Subject to} & x - 5z_2 - 10z_3 - 15z_4 & = 0 & \\
& y - 125z_2 - 225z_3 - 300z_4 & = 0 & \\
& z_1 + z_2 + z_3 + z_4 & = 1 & \\
& X_{1,1} + X_{1,2} - x & \leq 5 & \\
& X_{2,1} + X_{2,2} \leq 10 & & \\
& 0.5X_{1,1} - 0.5X_{2,1} & \geq 0 & \\
& 0.4X_{1,2} - 0.6X_{2,2} & \geq 0 & \\
& SOS2 : \{z_1, z_2, z_3, z_4\} & & \\
& X_{i,j} & \geq 0 & \forall i \in \{1,2\}, j \in \{1,2\} \\
& 0 \leq z_i & \leq 1 & \forall i \in \{1,2,3,4\} \\
& 0 \leq x & \leq 15 & \\
& y & \text{free} &
\end{array} \tag{10.20}$$

10.13.6.1. SOS2 with binary variables

Modeling Piecewise linear function

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

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

10.13.7. Maximizing a minimum

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

Consider the problem

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

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

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

10.13.8. Relaxing (nonlinear) equality constraints

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

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

10.14 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

10.14.1. Further Topics

- Precedence Constraints

10.15 MIP Solvers and Modeling Tools

- AMPL
- GAMS
- AIMMS
- Python-MIP

- Pyomo
- PuLP
- JuMP
- GUROBI
- CPLEX (IBM)
- Express
- SAS
- Coin-OR (CBC, CLP, IPOPT)
- SCIP

11. Algorithms and Complexity

Chapter 11. Algorithms and Complexity
60% complete. Goal 80% completion date: August 20
Notes:

Outcomes

1. *Describe asymptotic growth of functions using Big-O notation.*
2. *Analyze algorithms for the asymptotic runtime.*
3. *Classify problem types with respect to notions of how difficult they are to solve.*

How long will an algorithm take to run? How difficult might it be to solve a certain problem? Is the knapsack problem easier to solve than the traveling salesman problem? Or the matching problem? How can we compare the difficulty to solve these problems?

We will understand these questions through complexity theory. We will first use "Big-O" notation to simplify asymptotic analysis of the runtime of algorithms and the size of the input data of an algorithm.

We will then classify problem types as being either easy (in the class P) or probably very hard (in the class NP Hard). We will also learn about the problem classes NP, and NP-Complete.

To begin, watch these videos (Video 1, Video 2)about sorting algorithms. Notice how a different algorithm can produce a much different number of steps needed to solve the problem. The first video explains bubble sort and quick sort. The second video explains insertion sort, and then described the analysis of the algorithms (how many comparisons they make as the number of balls to sort grows. Pay attention to this analysis as this is very crucial in this module.

This video is a great introduction to the basic idea of Big-O notation. We will go over the more formal definition.

Here are two great videos about P versus NP (Video 1, Video 2).

11.1 Big-O Notation

We begin with some defintions 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 11.1: 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

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \tag{11.1}$$

This is because as n gets large, $g(n) >> f(n)$. 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 11.2: 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 $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist!
For this, we use the $\limsup$ notation.That is, we notice that

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty. \tag{11.2}$$

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 11.3: 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

$$0 \leq f(n) \leq c\, g(n) \quad \text{for all} \quad n \geq n_0. \tag{11.3}$$

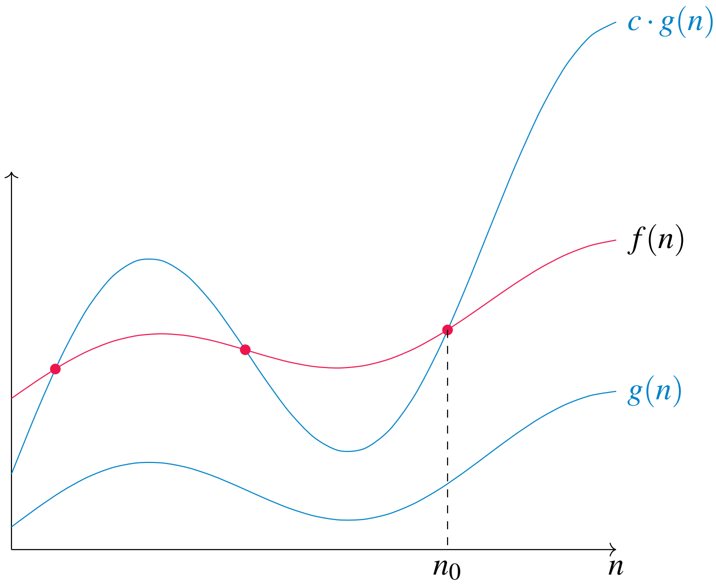


Figure 11.1: Example of Big-O notation: $f(n) = O(g(n))$. We see that for all $n \geq n_0$, we have $c \cdot g(n) \geq f(n)$.

Example 11.4:

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 17.3 is satisfied.
Note first that we always have

1. $n^2 \leq n^2$ and therefore $5n^2 \leq 5n^2$

Note that if $n \geq 1$, then

2. $n \leq n^2$ and therefore $10n \leq 10n^2$

3. $1 \leq n^2$ 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

$$5n^2 + 10n + 7 \leq 5n^2 + 10n^2 + 7n^2 \qquad \text{for all } n \geq 1, \tag{11.4}$$

$$\Rightarrow \quad 5n^2 + 10n + 7 \leq 22n^2 \qquad \text{for all } n \geq 1. \tag{11.5}$$

Hence, we have shown that Equation 17.3 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,	100010000
g(n)	200,	20000,	2000000,	200000000
h(n) .	2000,	2000000,	2000000000,	2000000000000

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

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
	n	10	20
Exponential	$n^{\log n}$	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-algorithms**[1]

Figure 11.2: time-of-algorithms

11.2 Algorithms - Example with Bubble Sort

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

Definition 11.5: 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.

11.2.1. Sorting

The problem of sorting a list is a simple problem to think about that has many algorithms to consider. We will describe one such algorithm: Bubble Sort.

Sorting Problem:

[1] **time-of-algorithms**, from **time-of-algorithms**. **time-of-algorithms**, **time-of-algorithms**.

Polynomial time (P)

Given a list of numbers $(x_1, \dots, x_n)$ sort them into increasing order.

Example 11.6: 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:

1. Compare numbers in position 1 and 2. If numbers are out of order, then swap them.
2. Next, compare numbers in position 2 and 3. If numbers are out of order, then swap them.
3. Continue this process of comparing subsequent numbers until you get to the end of the list (and compare numbers in position $n-1$ and n.

   Now the largest number should be in last position!
4. If no swaps had to be made, then the whole list is sorted!
5. Otherwise, if any swaps were needed, then set the last number aside, and start over from the beginning and sort the remaining list.

Example: Bubble Sort

Let try using Bubble Sort to sort this list.

First pass through the list.
Step 1:

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

Step 2:

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

Step 3:

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

Step 4:

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

Step 5:

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

Now 35 is in the last spot!

Second pass through the list

Step 1:

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

Step 2:

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

Step 3:

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

Step 4:

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

Now 22 is in the correct spot!

Third pass through the list

Step 1:

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

Step 2:

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

Step 3:

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

Fourth pass through the list

Step 1:

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

Step 2:

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

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 comparisons
- In the third pass, we needed 3 comparisons
- 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 $(n/2+1)$ pairst that add up to n, plus one $n/2$ left over.

Hence, the sum is

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

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.

[2] **bubble-sort-computational-example**, from **bubble-sort-computational-example**. **bubble-sort-computational-example**, **bubble-sort-computational-example**.

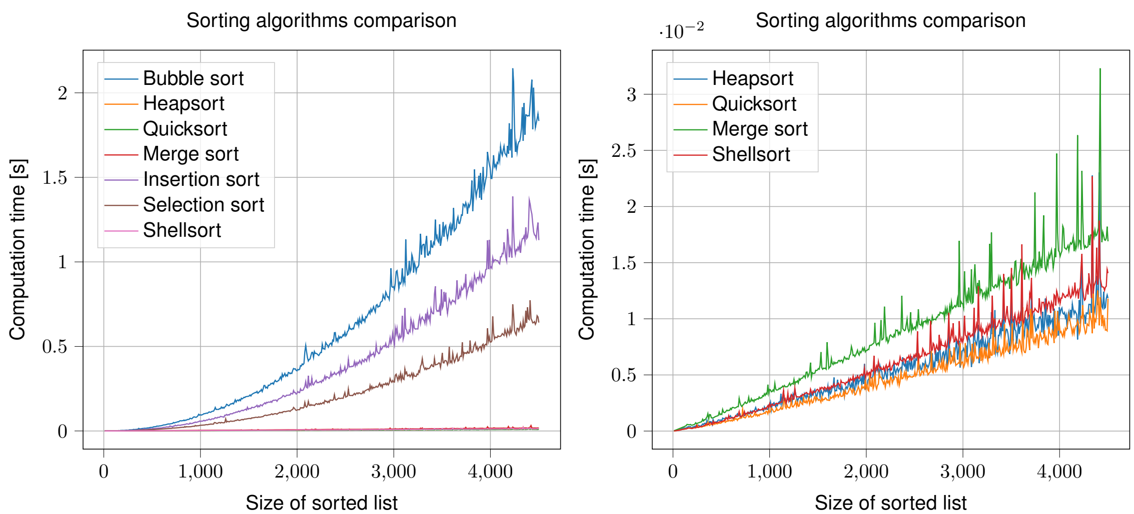
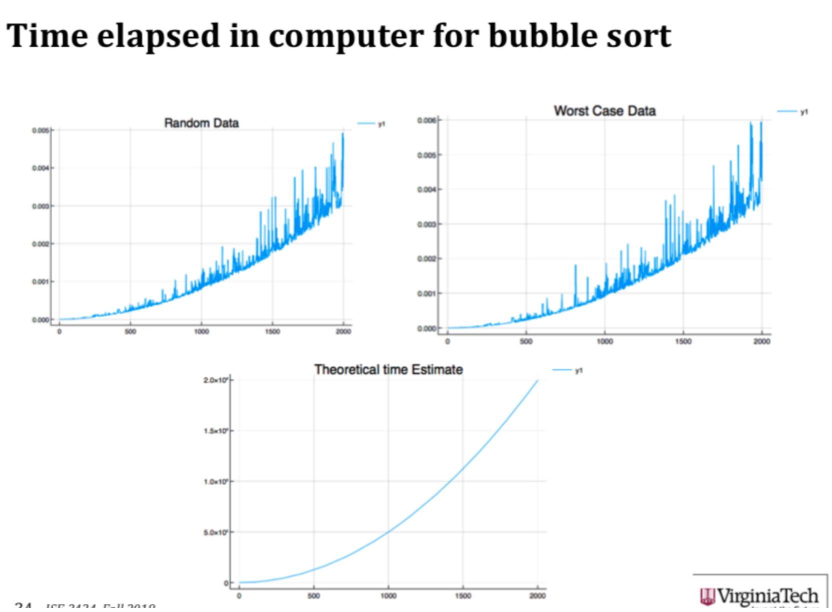


Figure 11.3: Comparison of runtimes of sorting algorithms.



© **bubble-sort-computational-example**[2]

Figure 11.4: bubble-sort-computational-example

There are some other relations that hold:

Theorem 11.7: Summations

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.
- $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$.

There are other formulas, but they get more complicated. In general, we know that

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

11.3 Problem, instance, size

11.3.1. Problem, instance

Definition 11.8: 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 11.9: Instance

An instance *is a specific case of a problem. For example, for the problem of sorting, an instance we saw already is* $(4, 9, 10, 15, 22, 35)$.

11.3.2. 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.

11.3.3. 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 11.10: 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 meerly comparing these numbers could take a long time.
So to be more precise, the size of the problem is

# of bits to encode the problem

which can be upper bounded by

$$n\phi_{\max}$$

where $\phi_{\max}$ is the maximum encoding size of a number given in the data.
For the sake of simplicity, we will typically ignore the size $\phi_{\max}$ in our complexity discussion. A more rigirous discussion of complexity will be given in later (advanced) parts of the book.

Example 11.11: 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 deceibed as $|V| + |E|$, that is, in terms of the number of nodes and the number of edges in the graph.

11.4 Complexity Classes

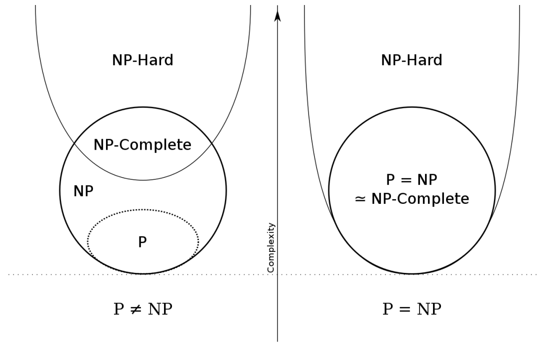
In this subsection we will discuss the complexity classes P, NP, NP-Complete, and NP-Hard. These classes help measure how difficult a problem is. **Informally,** these classes can be thought of as

- 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 17.5 .

We will now discuss these classes more formally.

[3]**wiki/File/complexity-classes.png**, from **wiki/File/complexity-classes.png**. **wiki/File/complexity-classes.png**, **wiki/File/complexity-classes.png**.



© wiki/File/complexity-classes.png[3]

Figure 11.5: Complexity class possibilities. Most academics agree that the case $P \neq NP$ is more likely.

11.4.1. P

Definition 11.12: 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 11.13: Sorting

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

Example 11.14: Complexity Minimum Spanning Tree

The minimum size spanning tree problem is in P. It can be solved, for instance, by Prim's algorithm, which by 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 11.15: 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.

11.4.2. NP

In this section, we will be more specific about the types of problems we want to consider. In particular, we will consider *decisions problems*. These are problems where we only request an answer of "yes" or "no".

We can rephrase maximization problems as problems that ask "does there exists a solution with objective value greater than some number?"

Example 11.16: Maximum Matching as a decisions problem

Input: *A graph $G = (V,E)$ with weights w_e for $e \in E$ and an objective goal W.*
Does there exists a matching with objective value greater than W?
Output: *Either "yes" or "no".*

We can now define the class of NP.

Definition 11.17: 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 $\mathscr{P}$
- Integer programming
- TSP
- Binary knapsack
- Maximum independent set
- Subset sum
- Partition
- SAT, k-SAT
- Clique

Thus, to show that a problem is in NP, you must do the following:

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

Example 11.18

Integer Linear Programming is in NP. More explicitly, the feasibility question of
"Does there exists an integer vector x that satisfies $Ax \leq b$"

is in NP.
Although it turns out to be difficult to find such an x or even prove that one exists, this problem is in NP for the following reason: if you are given a particular x and someone claims to you that it is a feasible solution to the problem, then you can easily check if they are correct. In this case, the vector x that you were given is called a certificate.
Note that it is easy to verify if x is a solution to the problem because you just have to

1. *Check if x is integer.*
2. *Use matrix multiplication to check if $Ax \leq b$ holds.*

11.4.3. 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 11.19: Reduction

Given two problems $\mathscr{A}, \mathscr{B}$, we say $\mathscr{A}$ is reduced to *$\mathscr{B}$ (and we write $\mathscr{A} \leq \mathscr{B}$ when we can assert that if we can solve $\mathscr{B}$ in polynomial time, then we can also solve $\mathscr{A}$ in polynomial time.*

11.4.4. 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.

11.4.5. 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

1. showing that it can be used to solve binary integer programming feasibility,
2. showing that the problem is in NP.

The first problem proven to be NP-Complete is called *3-SAT* []. 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

$$\texttt{true} \wedge \texttt{true} = \texttt{true} \tag{11.1}$$
$$\texttt{true} \wedge \texttt{false} = \texttt{false} \tag{11.2}$$
$$\texttt{false} \wedge \texttt{false} = \texttt{false} \tag{11.3}$$
$$\texttt{false} \wedge \texttt{true} = \texttt{false} \tag{11.4}$$

$$\texttt{true} \vee \texttt{true} = \texttt{true} \tag{11.5}$$
$$\texttt{true} \vee \texttt{false} = \texttt{true} \tag{11.6}$$
$$\texttt{false} \vee \texttt{false} = \texttt{false} \tag{11.7}$$
$$\texttt{false} \vee \texttt{true} = \texttt{true} \tag{11.8}$$

$$\neg\, \texttt{true} = \texttt{false} \tag{11.9}$$
$$\neg\, \texttt{false} = \texttt{true} \tag{11.10}$$

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

$$(X_1 \wedge \neg X_2 \vee X_3) \wedge (X_1 \vee \neg X_3) \vee (X_1 \wedge X_2 \wedge X_3). \tag{11.11}$$

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

$$X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4. \tag{11.12}$$

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

$$(X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4) \wedge (X_1 \vee X_2 \vee X_3) \wedge (X_2 \vee \neg X_3 \vee \neg X_4 \vee X_5). \tag{11.13}$$

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.

11.5 Problems and Algorithms

We will discuss the following concepts:

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

11.5.1. Matching Problem

Definition 11.20: Matching

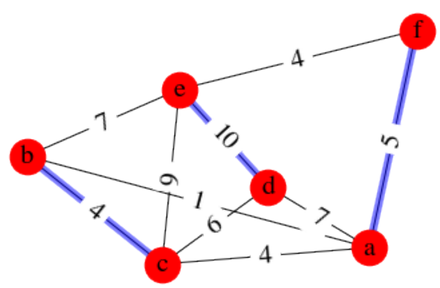
Given a graph $G = (V,E)$, a matching *is a subset $E' \subseteq E$ such that no vertex $v \in V$ is contained in more than one edge in E'.*
A perfect matching *is a matching were every vertex is connected to an edges in E'.*
A maximal matching *is a matching E' such that there is no matching E'' that strictly contains it.*

INCLUDE PICTURES OF MATCHINGS

Figure 11.6: 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 11.21: 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-maximal**[4]

Figure 11.7: graph-for-matching-maximal

[4]**graph-for-matching-maximal**, from **graph-for-matching-maximal**. **graph-for-matching-maximal**, **graph-for-matching-maximal**.

11.5.1.1. Greedy Algorithm for Maximal Matching

The greedy algorithm iteratively adds the edge with largest weight that is feasible to add.

Greedy Algorithm for Maximal Matching:

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

1. Begin with an empty graph $(M = \emptyset)$
2. Label the edges in the graph such that $w(e_1) \geq w(e_2) \geq \cdots \geq w(e_m)$
3. For $i = 1, \ldots, m$
   If $M \cup \{e_i\}$ is a valid matching (i.e., no vertex is incident with two edges), then set $M \leftarrow M \cup \{e_i\}$ (i.e., add edge e_i to the graph M)
4. Return M

Theorem 11.22: Greedy algorithm gives a 2-approximation [[Avis83]]

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

11.5.1.2. Other algorithms to look at

1. Improved Algorithm [**DRAKE2003211**]
2. Blossom Algorithm Wikipedia

11.5.2. Minimum Spanning Tree

Definition 11.23: Spanning Tree

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

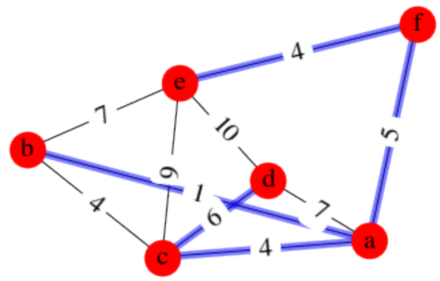
© **spanning-tree**[5]

Figure 11.8: spanning-tree

[5]**spanning-tree**, from **spanning-tree**. **spanning-tree**, **spanning-tree**.

Definition 11.24: 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-MST[6]

Figure 11.9: spanning-tree-MST

Lemma 11.25: Edges and Spanning Trees

Let G be a connected graph with n vertices.

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

See Section 19.2 for integer programming formulations of this problem.

11.5.3. Kruskal's algorithm

Kruskal - for Minimum Spanning tree:

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

1. Begin with an empty tree $(T = \emptyset)$
2. Label the edges in the graph such that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
3. For $i = 1, \ldots, m$
   If $T \cup \{e_i\}$ is acyclic, then set $T \leftarrow T \cup \{e_i\}$
4. Return T

11.5.3.1. Prim's Algorithm

[6] **spanning-tree-MST**, from **spanning-tree-MST**. **spanning-tree-MST**, **spanning-tree-MST**.

11.5.4. Traveling Salesman Problem

See Section 19.3 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 **??**.

11.5.4.1. Nearest Neighbor - Construction Heuristic

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

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

Nearest Neighbor:

Complexity: $O(n^2)$

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

11.5.4.2. 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 17.10

Definition 11.26: Triangle Inequality on a Graph

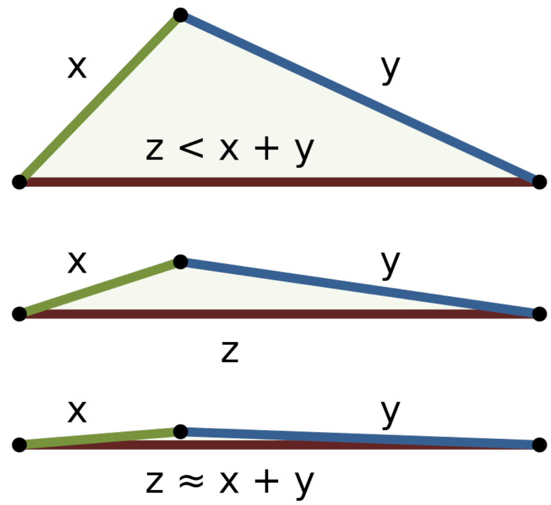
A complete, weighted graph G (i.e., a graph that has all possible edges and a weight assigned to each edge) satisfies the triangle inequality *provided that for ever triple of vertices a,b,c and edges e_{ab}, e_{bc}, e_{ac}, we have that*

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

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

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

[7] **wiki/File/triangle_inequality.png**, from **wiki/File/triangle_inequality.png**. **wiki/File/triangle_inequality.png**, **wiki/File/triangle_inequality.png**.



© wiki/File/triangle_inequality.png[7]

Figure 11.10: wiki/File/triangle_inequality.png

Algorithm 1 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.

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

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

Lastly, due to the triangle inequality we know that

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

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

Putting this together, we have

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

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

11.5.4.3. 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.

11.6 Resources

Resources

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

Resources

Bubble Sort

- *Wikipedia*

Resources

Kruskal Wikipedia

Resources

Prim's Algorithm

- *Wikipedia*
- *TeXAmple - Figure for min spannig tree*

Resources

Nearest Neighbor for TSP Wikipedia

12. Introduction to computational complexity

Chapter 12. Introduction to computational complexity
Move this section to mode advanced version of the book.

12.1 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

12.2 Problem, instance, size

12.2.1. Problem, instance

Definition 12.1: 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 12.2: Instance

Is a specific case of a problem. In other words, we can say "Instance∈Problem".

12.2.2. 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.

12.2.3. Size of an instance

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

Definition 12.3: 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-egde matrix represetation).

12.3 Algorithms, running time, Big-O notation

12.3.1. Basics

Definition 12.4: Algorithm

List of instructions to solve a problem.

Definition 12.5: 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.

12.3.2. 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 $f : \mathbb{Z}_+ \to \mathbb{Z}_+$.
- Given σ, the function f is defined as follows:

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

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

12.3.3. Big-O notation

A function $f : \mathbb{Z}_+ \to \mathbb{Z}_+$ belongs to the class of functions $\mathrm{O}(g(n))$ (that is, $f \in \mathrm{O}(g(n))$) 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 $\mathrm{O}(g(n))$" or "f is order $\mathrm{O}(g(n))$".

12.3.4. Examples

- Basic examples of Big-O notation: $\mathrm{O}(1)$, $\mathrm{O}(n^k)$, $\mathrm{O}(c^n)$, $\mathrm{O}(\log(n))$, 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 $\log(b)$.

12.4 Basics

Definition 12.6: Polynomial time algorithm

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

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

Definition 12.7: 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.

12.5 Complexity classes

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

12.5.1. Polynomial time problems

Definition 12.8: The class $\mathscr{P}$

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) $P \in \mathscr{P}$, (2) $P \notin \mathscr{P}$, and $P\ ?\ \mathscr{P}$ (i.e., we don't know).

Examples:

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

12.5.2. Non-deterministic polynomial time problems

Definition 12.9: 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 $\mathscr{P}$
- Integer programming
- TSP
- Binary knapsack
- Maximum independent set
- Subset sum
- Partition
- SAT, k-SAT
- Clique

12.5.3. Complements of problems in NP

Definition 12.10: 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 $\mathscr{P}$
- Every "complement" of an NP problem
- PRIMES

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

12.6 Relationship between the classes

12.6.1. A basic result

Theorem 12.11

The following relationship holds:

$$\mathscr{P} \subseteq NP \cap coNP.$$

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

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

12.7 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 12.12: Polynomial time reductions

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

$$f : P_1 \to P_2$$

such that

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

Remarks:

1. 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.
2. Therefore, the notation $P_1 \leq_{\mathscr{P}} P_2$ 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.

12.8 Comparing problems, Polynomial time reductions

12.8.1. 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 12.13: Polynomial time reductions

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

$$f : P_1 \to P_2$$

such that

1. *For all $w \in P_1$ the answer to w is YES if and only if the answer to $f(w)$ is YES.*
2. *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:

1. 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.
2. Therefore, the notation $P_1 \leq_{\mathscr{P}} P_2$ 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.

12.8.2. Basic properties

Proposition 12.14

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

Proposition 12.15

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

Proposition 12.16

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

12.9 NP-Completeness

12.9.1. The basics

Definition 12.17: NP-Completeness

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

1. $P \in NP$
2. $Q \leq_{\mathscr{P}} P$ *for all* $Q \in NP$ *(that is, every problem Q in NP can be polynomially reduced to P).*

Proposition 12.18

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

12.9.2. Do NP-complete problems exist?

Theorem 12.19: S. Cook, 1971

SAT is NP-complete.

12.10 NP-Hardness

Definition 12.20: 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 $P \in \mathscr{P}$ then $\mathscr{P} = \mathrm{NP}$.

12.11 Exercises

1. Let P, Q be decision problems such that every instance of Q is an instance of P (that is $\{\text{instances in } Q\} \subseteq \{\text{instances in } P\}$).

   (a) Give an example of P, Q such that $Q \in \mathscr{P}$ and $P \in \text{NP} - \text{complete}$.

   (b) 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:

(a)
- 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, $a_1, \ldots, a_n = 1$. The following algorithm decides this special case:

  `ALGORITHM:`

  1. List the objects $1, \ldots, n$ in decreasing order according to $c_1, \ldots, c_n$.
  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, $Q \in \mathscr{P}$.

(b)
- Let P be *SAT*.
- Let Q be *3-SAT*.

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

2. 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?

   (a) Given that *Hamiltonian Cycle* is NP-complete, prove that *Hamiltonian Path* is NP-complete.

   (b) Given that *Hamiltonian Cycle* is NP-complete, prove that the optimization version of the TSP problem is NP-hard.

Solution:

(a) **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: *Hamiltonian Cycle* $\leq_P$ *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

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

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

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

in G'.

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

Conclusion: *Hamiltonian Path* is NP-complete.

(b) 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.

3. Given that the *node packing problem* is $\mathrm{NP}-$complete, show that the following problems are also $\mathrm{NP}-$complete:

(a) *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?

(b) *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:

(a) **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.

(b) ***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.

13. Exponential Size Formulations

Chapter 13. Exponential Size Formulations
60% complete. Goal 80% completion date: August 20
Notes:

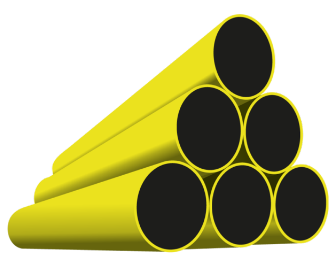
Although typically models need to be a reasonable size in order for us to code them and send them to a solver, there are some ways that we can allow having models of exponential size. The first example here is the cutting stock problem, where we will model with exponentially many variables. The second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

13.1 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 of 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.


1

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 13.1: Cutting stock with pipes

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

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

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

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

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

Then we have the following model

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{N} z_j \\
\text{s.t.} \quad & \sum_{i=1}^{k} L_i x_{ij} \le L z_j \quad \text{for } j = 1,\dots,N \\
& \sum_{j=1}^{N} x_{ij} \ge d_i \quad \text{for } i = 1,\dots,k \\
& z_j \in \{0,1\} \text{ for } j = 1,\dots,N \\
& x_{ij} \in \mathbb{Z}_+ \text{ for } i = 1,\dots,k,\ j = 1,\dots,N
\end{aligned}
\tag{13.1}
$$

Exercise 13.2: Show Bound

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

Example: Gurobipy

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

Table 13.1: Table comparing computaional time in the two models. We stopped computations at 600 seconds. Notice that the pattern model does not care how large the demand is - it still solves in the same amount of time! The demand multiplier k here means that we multiply k times the demand vector used in the example. This grows the number of variables in the Board based model, but doesn't change much in the pattern based model.

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

$$\begin{aligned}
\min \quad & \sum_{j=1}^{16} z_j \\
\text{s.t.} \quad & 4x_{1j} + 5x_{2j} + 6x_{3j} \leq 19 z_j \\
& \sum_{j=1}^{16} x_{1j} \geq 12 \\
& \sum_{j=1}^{16} x_{2j} \geq 15 \\
& \sum_{j=1}^{16} x_{3j} \geq 22 \\
& z_j \in \{0,1\} \text{ for } j = 1, \dots, 16 \\
& x_{ij} \in \mathbb{Z}_+ \text{ for } i = 1, \dots, 3,\ j = 1, \dots, 16
\end{aligned} \tag{13.2}$$

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

$$z_1 \geq z_2 \geq z_3 \geq \cdots \geq z_N. \tag{13.3}$$

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

13.1.1. 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

$$\begin{aligned} \sum_{i=1}^{k} L_i a_i &\le L \\ a_i &\in \mathbb{Z}_+ \text{ for all } i = 1, \dots, k \end{aligned} \tag{13.4}$$

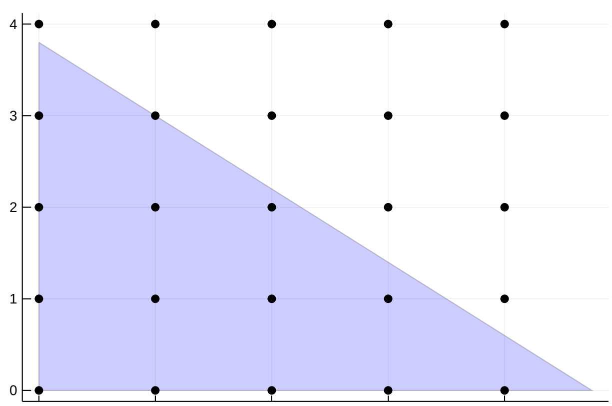
In our running example, we have

$$\begin{aligned} 4a_1 + 5a_2 + 6a_3 &\le 19 \\ a_i &\in \mathbb{Z}_+ \text{ for all } i = 1, \dots, 3 \end{aligned} \tag{13.5}$$

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

$$P = \{(a_1, a_2) : 4a_1 + 5a_2 \le 19, a_1 \ge 0, a_2 \ge 0\} \tag{13.6}$$

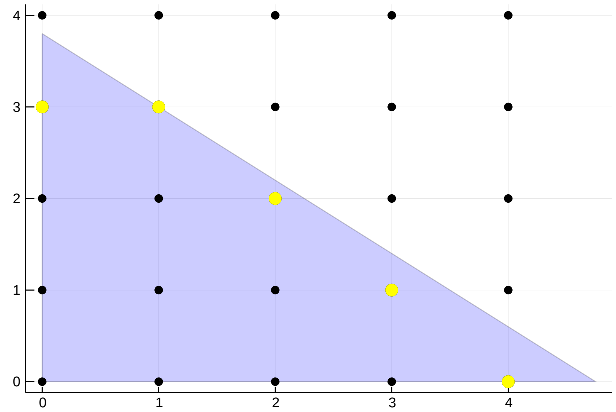
which we can see here:



© **knapsack_fig.pdf**[2]

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.

[2]**knapsack_fig.pdf**, from **knapsack_fig.pdf**. **knapsack_fig.pdf**, **knapsack_fig.pdf**.



© **knapsack_fig_maximal.pdf**[3]

For example, the pattern $[3,0,0]$ is not needed (only cut 3 of length 4m) since we could also use the patten $[4,0,0]$ (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 13.3: 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.

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 2 & 1 & 2 & 3 & 4 & 1 \\ 0 & 1 & 0 & 2 & 1 & 2 & 2 & 1 & 0 & 3 \\ 3 & 2 & 2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{13.7}$$

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

$$\min \sum_{i=1}^{p} x_i \tag{13.8}$$

$$\text{such that } Ax \geq \begin{bmatrix} 12 \\ 15 \\ 22 \end{bmatrix} \tag{13.9}$$

$$x \in \mathbb{Z}_+^p \tag{13.10}$$

[3]**knapsack_fig_maximal.pdf**, from **knapsack_fig_maximal.pdf**. **knapsack_fig_maximal.pdf**, **knapsack_fig_maximal.pdf**.

13.1.2. Column Generation

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

Thus we can write it as

$$\begin{aligned} \min \quad & (c_N - c_B A_B^{-1} A_N) x_N + c_B A_B^{-1} b \\ \text{s.t.} \quad & x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ & x \geq 0 \end{aligned} \tag{13.11}$$

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

$$\begin{aligned} \min \quad & (1_N - 1_B A_B^{-1} N) x_N + 1_B A_B^{-1} b \\ \text{s.t.} \quad & x_B + A_B^{-1} A_N x_N = A_B^{-1} b \\ & x \geq 0 \end{aligned} \tag{13.12}$$

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

$$\left(1 - 1_B A_B^{-1} A_N^i\right) x_i \tag{13.13}$$

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

$$1 - 1_B A_B^{-1} a < 0 \quad \Rightarrow \quad 1 < 1_B A_B^{-1} a \tag{13.14}$$

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.

$$\begin{aligned} \max \quad & 1_B A_B^{-1} a \\ \text{s.t.} \quad & \sum_{i=1}^{k} L_i a_i \leq L \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \ldots, k \end{aligned} \tag{13.15}$$

Example 13.4: 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

$$A = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{pmatrix} \tag{13.16}$$

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.

$$A = \begin{pmatrix} 4 & 0 & 0 & a_1 \\ 0 & 3 & 0 & a_2 \\ 0 & 0 & 3 & a_3 \end{pmatrix} \tag{13.17}$$

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

$$\begin{aligned} \max \quad & \tfrac{1}{4}a_1 + \tfrac{1}{3}a_2 + \tfrac{1}{3}a_3 \\ s.t. \quad & 4a_1 + 5a_2 + 6a_3 \leq 19 \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k \end{aligned} \tag{13.18}$$

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

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

13.1.3. Cutting Stock - Multiple widths

Resources

Gurobi has as 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

13.2 Spanning Trees

Resources

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

13.3 Traveling Salesman Problem

Resources

See math.waterloo.ca for excellent material on the TSP.
See also this chapter A Practical Guide to Discrete Optimization.
Also, watch this excellent talk by Bill Cook "Postcards from the Edge of Possibility": Youtube!

Google Maps!

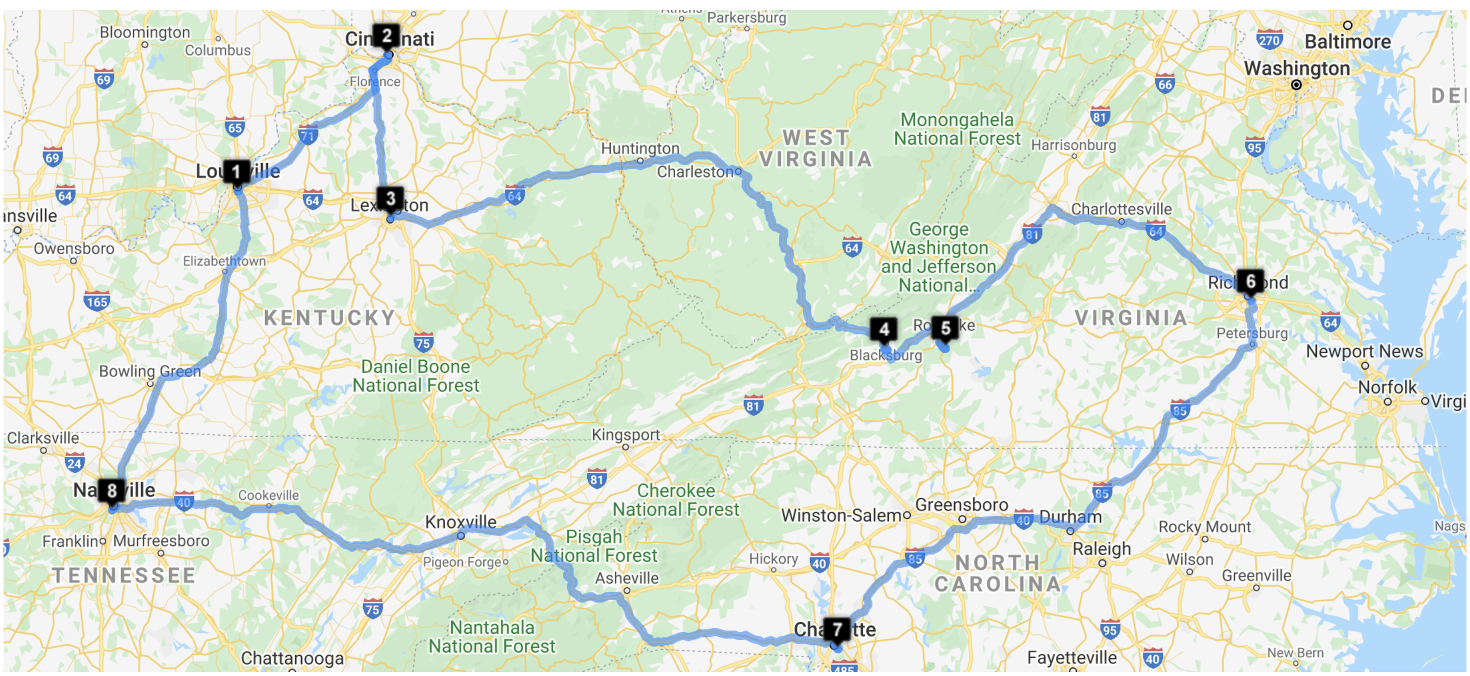


Figure 13.1: Optimal tour through 8 cities. Generated by Gebweb - Optimap. See it also on Google Maps!.

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 $i \to j$.

Figure 13.2: wiki/File/William_Rowan_Hamilton_painting.jpg

A *tour*, or Hamiltonian cycle (see Figure 19.2), 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$i \to$ city$j$.

[4] **wiki/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**.

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

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

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

$$\sum_{j \in N} x_{ij} = 1 \qquad \text{for all } i \in N \quad \text{[outgoing arc]} \tag{13.1}$$

$$\sum_{i \in N} x_{ij} = 1 \qquad \text{for all } j \in N \quad \text{[incoming arc]} \tag{13.2}$$

$$x_{ii} = 0 \qquad \text{for all } i \in N \quad \text{[no self loops]} \tag{13.3}$$

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

ADD SUBTOURS FIGURE

13.3.1. Miller Tucker Zemlin (MTZ) Model

The Miller-Tucker-Zemlin (MTZ) model for the TSP uses varibles 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 (19.1)-(19.3) 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

$$\text{If } x_{ij} = 1, \text{ then } u_i + 1 \leq u_j. \tag{13.4}$$

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

$$u_i + 1 \leq u_j + n(1 - x_{ij}) \tag{13.5}$$

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

Consider a subtour now $2 \to 5 \to 3 \to 2$. Thus, $x_{25} = x_{53} = x_{32} = 1$. Then using the constraints from (19.5), we have that

$$u_2 < u_5 < u_3 < u_2, \tag{13.6}$$

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:

$$
\begin{aligned}
\min \sum_{i,j \in N} & c_{ij} x_{ij} && && (13.7)\\
\sum_{j \in N} x_{ij} &= 1 && \text{for all } i \in N \quad \text{[outgoing arc]} && (13.8)\\
\sum_{i \in N} x_{ij} &= 1 && \text{for all } j \in N \quad \text{[incoming arc]} && (13.9)\\
x_{ii} &= 0 && \text{for all } i \in N \quad \text{[no self loops]} && (13.10)\\
u_i + 1 &\le u_j + n(1 - x_{ij}) && \text{for all } i, j \in N, i, j \neq 1 \quad \text{[prevents subtours]} && (13.11)\\
u_1 &= 1 && && (13.12)\\
2 \le & u_i \le n && \text{for all } i \in N, i \neq 1 && (13.13)\\
u_i &\in \mathbb{Z} && \text{for all } i \in N && (13.14)\\
x_{ij} &\in \{0, 1\} && \text{for all } i, j \in N && (13.15)
\end{aligned}
$$

Example 13.5: 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 13.6: 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

$$\begin{aligned}
& x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} = 1 && \textit{outgoing from node 1} \\
& x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} = 1 && \textit{outgoing from node 2} \\
& x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} = 1 && \textit{outgoing from node 3} \\
& x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} = 1 && \textit{outgoing from node 4} \\
\\
& x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} = 1 && \textit{incoming to node 1} \\
& x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} = 1 && \textit{incoming to node 2} \\
& x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} = 1 && \textit{incoming to node 3} \\
& x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} = 1 && \textit{incoming to node 4} \\
\\
& x_{1,1} = 0 && \textit{No self loop with node 1} \\
& x_{2,2} = 0 && \textit{No self loop with node 2} \\
& x_{3,3} = 0 && \textit{No self loop with node 3} \\
& x_{4,4} = 0 && \textit{No self loop with node 4} \\
\\
& u_1 = 1 && \textit{Start at node 1} \\
& 2 \leq u_i \leq 4, \quad \forall i \in \{2,3,4\} \\
& u_2 + 1 \leq u_3 + 4(1 - x_{2,3}) \\
& u_2 + 1 \leq u_4 + 4(1 - x_{2,4}) \leq 3 \\
& u_3 + 1 \leq u_2 + 4(1 - x_{3,2}) \leq 3 \\
& u_3 + 1 \leq u_4 + 4(1 - x_{3,4}) \leq 3 \\
& u_4 + 1 \leq u_2 + 4(1 - x_{4,2}) \leq 3 \\
& u_4 + 1 \leq u_3 + 4(1 - x_{4,3}) \leq 3 \\
& x_{i,j} \in \{0,1\} \quad \forall i \in \{1,2,3,4\}, j \in \{1,2,3,4\} \\
& u_i \in \mathbb{Z}, \quad \forall i \in \{1,2,3,4\}
\end{aligned}$$

Example 13.7: MTZ model for TSP with 5 nodes

$$
\begin{aligned}
\min \quad & x_{1,2}+2x_{1,3}+3x_{1,4}+4x_{1,5}+x_{2,1}+x_{2,3}+2x_{2,4}+2x_{2,5}+2x_{3,1}+ \\
& x_{3,2}+4x_{3,4}+x_{3,5}+3x_{4,1}+2x_{4,2}+4x_{4,3}+2x_{4,5}+ \\
& 4x_{5,1}+2x_{5,2}+x_{5,3}+2x_{5,4} \\
\textit{Subject to} \quad & x_{1,1}+x_{1,2}+x_{1,3}+x_{1,4}+x_{1,5}=1 \\
& x_{2,1}+x_{2,2}+x_{2,3}+x_{2,4}+x_{2,5}=1 \\
& x_{3,1}+x_{3,2}+x_{3,3}+x_{3,4}+x_{3,5}=1 \\
& x_{4,1}+x_{4,2}+x_{4,3}+x_{4,4}+x_{4,5}=1 \\
& x_{5,1}+x_{5,2}+x_{5,3}+x_{5,4}+x_{5,5}=1 \\
& \\
& x_{1,1}+x_{2,1}+x_{3,1}+x_{4,1}+x_{5,1}=1 \\
& x_{1,2}+x_{2,2}+x_{3,2}+x_{4,2}+x_{5,2}=1 \\
& x_{1,3}+x_{2,3}+x_{3,3}+x_{4,3}+x_{5,3}=1 \\
& x_{1,4}+x_{2,4}+x_{3,4}+x_{4,4}+x_{5,4}=1 \\
& x_{1,5}+x_{2,5}+x_{3,5}+x_{4,5}+x_{5,5}=1 \\
& \\
& x_{1,1}=0 \\
& x_{2,2}=0 \\
& x_{3,3}=0 \\
& x_{4,4}=0 \\
& x_{5,5}=0 \\
& \\
& u_1=1 \\
& 2\le u_i\le 5 \quad \forall i\in\{1,2,3,4,5\} \\
& u_2+1\le u_3+5(1-x_{2,3}) \\
& u_2+1\le u_4+5(1-x_{2,4}) \\
& u_2+1\le u_5+5(1-x_{2,5}) \\
& u_3+1\le u_2+5(1-x_{3,2}) \\
& u_3+1\le u_4+5(1-x_{3,4}) \\
& u_4+1\le u_2+5(1-x_{4,2}) \\
& u_4+1\le u_3+5(1-x_{4,3}) \\
& u_3+1\le u_5+5(1-x_{3,5}) \\
& u_4+1\le u_5+5(1-x_{4,5}) \\
& u_5+1\le u_2+5(1-x_{5,2}) \\
& u_5+1\le u_3+5(1-x_{5,3}) \\
& u_5+1\le u_4+5(1-x_{5,4}) \\
& x_{i,j}\in\{0,1\} \quad \forall i\in\{1,2,3,4,5\}, j\in\{1,2,3,4,5\} \\
& u_i\in\mathbb{Z}, \quad \forall i\in\{1,2,3,4,5\}
\end{aligned}
$$

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 13.8: Subtour elimation constraints via MTZ model

Consider the subtour $2 \to 4 \to 5 \to 2$.
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:

$$u_2 + 1 \le u_4 + 5(1 - x_{2,4})$$
$$u_4 + 1 \le u_5 + 5(1 - x_{4,5})$$
$$u_5 + 1 \le u_2 + 5(1 - x_{5,2}).$$

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

$$u_2 + 1 \le u_5$$
$$u_4 + 1 \le u_5$$
$$u_5 + 1 \le u_2.$$

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

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

which reduces to

$$3 \le 0.$$

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

Example 13.9: Weak Model

Consider again the same tour in the last example, that is, the subtour $2 \to 4 \to 5 \to 2$. 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

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

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

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

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

13.3.2. 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

$$2 \to 5 \to 3 \to 2. \tag{13.16}$$

We can prevent this subtour by adding the constraint

$$x_{25} + x_{53} + x_{32} \leq 2 \tag{13.17}$$

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*

$$\sum_{(i,j)\in S} x_{ij} \leq |S| - 1 \qquad \text{Subtour Elimination Constraint.} \tag{13.18}$$

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:

$$\min \sum_{i,j\in N} c_{ij}x_{ij} \tag{13.19}$$

$$\sum_{j\in N} x_{ij} = 1 \qquad \text{for all } i \in N \quad \text{[outgoing arc]} \tag{13.20}$$

$$\sum_{i\in N} x_{ij} = 1 \qquad \text{for all } j \in N \quad \text{[incoming arc]} \tag{13.21}$$

$$x_{ii} = 0 \qquad \text{for all } i \in N \quad \text{[no self loops]} \tag{13.22}$$

$$\sum_{(i,j)\in S} x_{ij} \leq |S| - 1 \qquad \text{for all subtours } S \quad \text{[prevents subtours]} \tag{13.23}$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i,j \in N \tag{13.24}$$

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 13.10: 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

$$\begin{aligned}
& x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} = 1 && \textit{outgoing from node 1} \\
& x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} = 1 && \textit{outgoing from node 2} \\
& x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} = 1 && \textit{outgoing from node 3} \\
& x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} = 1 && \textit{outgoing from node 4} \\
\\
& x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} = 1 && \textit{incoming to node 1} \\
& x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} = 1 && \textit{incoming to node 2} \\
& x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} = 1 && \textit{incoming to node 3} \\
& x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} = 1 && \textit{incoming to node 4} \\
\\
& x_{1,1} = 0 && \textit{No self loop with node 1} \\
& x_{2,2} = 0 && \textit{No self loop with node 2} \\
& x_{3,3} = 0 && \textit{No self loop with node 3} \\
& x_{4,4} = 0 && \textit{No self loop with node 4} \\
\\
& x_{1,2} + x_{2,1} \le 1 && S = [(1,2),(2,1)] \\
& x_{1,3} + x_{3,1} \le 1 && S = [(1,3),(3,1)] \\
& x_{1,4} + x_{4,1} \le 1 && S = [(1,4),(4,1)] \\
& x_{2,3} + x_{3,2} \le 1 && S = [(2,3),(3,2)] \\
& x_{2,4} + x_{4,2} \le 1 && S = [(2,4),(4,2)] \\
& x_{3,4} + x_{4,3} \le 1 && S = [(3,4),(4,3)] \\
& x_{2,1} + x_{1,3} + x_{3,2} \le 2 && S = [(2,1),(1,3),(3,2)] \\
& x_{1,2} + x_{2,3} + x_{3,1} \le 2 && S = [(1,2),(2,3),(3,1)] \\
& x_{3,1} + x_{1,4} + x_{4,3} \le 2 && S = [(3,1),(1,4),(4,3))] \\
& x_{1,3} + x_{3,4} + x_{4,1} \le 2 && S = [(1,3),(3,4),(4,1)] \\
& x_{2,1} + x_{1,4} + x_{4,2} \le 2 && S = [(2,1),(1,4),(4,2)] \\
& x_{1,2} + x_{2,4} + x_{4,1} \le 2 && S = [(1,2),(2,4),(4,1)] \\
& x_{3,2} + x_{2,4} + x_{4,3} \le 2 && S = [(3,2),(2,4),(4,3)] \\
& x_{2,3} + x_{3,4} + x_{4,2} \le 2 && S = [(2,3),(3,4),(4,2)] \\
\\
& x_{i,j} \in \{0,1\} \quad \forall i \in \{1,2,3,4\}, j \in \{1,2,3,4\}
\end{aligned}$$

Example 13.11

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* .

13.3.3. Traveling Salesman Problem - Branching Solution

We will see in the next section

1. That the constraint (19.1)-(19.3) always produce integer solutions as solutions to the linear relaxation.

2. A way to use branch and bound (the topic of the next section) in order to avoid subtours.

13.3.4. Traveling Salesman Problem Variants

13.3.4.1. Many salespersons (m-TSP)

m-Traveling Salesman Problem - DFJ Model:

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \tag{13.25}$$

$$\sum_{j \in N} x_{ij} = 1 \qquad \text{for all } i \in N \quad \text{[outgoing arc]} \tag{13.26}$$

$$\sum_{i \in N} x_{ij} = 1 \qquad \text{for all } j \in N \quad \text{[incoming arc]} \tag{13.27}$$

$$\sum_{j \in N} x_{Dj} = m \qquad \text{for all } i \in N \quad \text{[outgoing arc]} \tag{13.28}$$

$$\sum_{i \in N} x_{iD} = m \qquad \text{for all } j \in N \quad \text{[incoming arc]} \tag{13.29}$$

$$x_{ii} = 0 \qquad \text{for all } i \in N \cup \{D\} \quad \text{[no self loops]} \tag{13.30}$$

$$\sum_{(i,j) \in S} x_{ij} \le |S| - 1 \qquad \text{for all subtours } S \subseteq N \ \text{[prevents subtours]} \tag{13.31}$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i,j \in N \cup \{D\} \tag{13.32}$$

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 $u_i \le T$. 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

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \tag{13.33}$$

$$\sum_{j \in N} x_{ij} = 1 \qquad \text{for all } i \in N \quad \text{[outgoing arc]} \tag{13.34}$$

$$\sum_{i \in N} x_{ij} = 1 \qquad \text{for all } j \in N \quad \text{[incoming arc]} \tag{13.35}$$

$$\sum_{j \in N} x_{Dj} = m \qquad \text{for all } i \in N \quad \text{[outgoing arc]} \tag{13.36}$$

$$\sum_{i \in N} x_{iD} = m \qquad \text{for all } j \in N \quad \text{[incoming arc]} \tag{13.37}$$

$$x_{ii} = 0 \qquad \text{for all } i \in N \cup \{D\} \quad \text{[no self loops]} \tag{13.38}$$

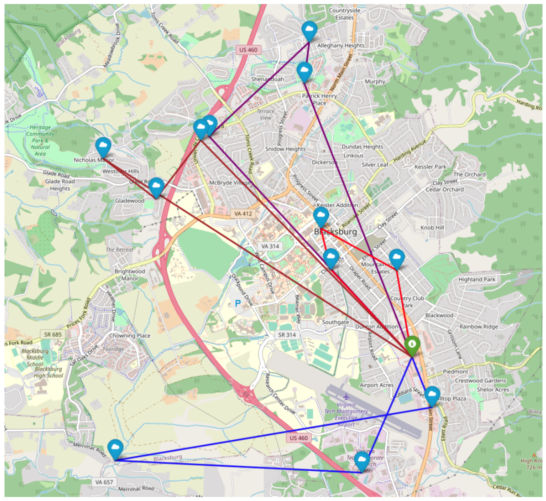
$$u_i + 1 \le u_j + n(1 - x_{ij}) \qquad \text{for all } i,j \in N, i,j \ne 1 \ \text{[prevents subtours]} \tag{13.39}$$

$$u_1 = 1 \tag{13.40}$$

$$2 \le u_i \le T \qquad \text{for all } i \in N, i \ne 1 \tag{13.41}$$

$$u_i \in \mathbb{Z} \qquad \text{for all } i \in N \tag{13.42}$$

$$x_{ij} \in \{0,1\} \qquad \text{for all } i,j \in N \cup \{D\} \tag{13.43}$$



© **m-tsp_solution**[5]

Image html

13.3.4.2. 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

$$u_2 + 1 = u_3.$$

13.4 Vehicle Routing Problem (VRP)

Section 13.4. Vehicle Routing Problem (VRP)
10% complete.
Add discussion and examples of solving VRP using Google OR tools. `https://www.youtube.com/watch?v=AJ6LeiMe_PQ&t=757s&ab_channel=MixedIntegerProgramming`
Add description and link to code for Clark-Wright Algorithm
Discuss that there are many many variations of this problem and it is somewhat endless to work on.

The VRP is a generlization 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.

[5] **m-tsp_solution**, from **m-tsp_solution**. **m-tsp_solution**, **m-tsp_solution**.

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.

13.4.1. Case Study: Bus Routing in Boston

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

13.4.2. An Integer Programming Model

ILP

$$\begin{aligned} c_{ij} = &\text{ cost of travel } i \text{ to } j \\ x_{ijk} = &\, 1 \text{ iff vehicle } k \text{ travels} \\ &\text{directly from } i \text{ to } j \end{aligned}$$

subject to

$$\begin{array}{ccc} \min \sum_{i,j} c_{ij} \sum_k x_{ijk} & & \\ \sum_i \sum_k x_{ijk} = 1 & \forall j \neq \text{ depot} & \text{Exactly one vehicle in} \\ \sum_j \sum_k^k x_{ijk} = 1 & \forall i \neq \text{ depot} & \text{Exactly one vehicle out} \\ \sum_i \sum_k x_{ihk} - \sum_j \sum_k x_{hjk} = 0 & \forall k, h & \text{It's the same vehicle} \\ \sum_i q_i \sum_j x_{ijk} \leq Q_k \quad \forall k & \text{Capacity constraint} & \\ \sum_{ijk} x_{ijk} = |S| - 1 \forall S \subseteq P(N), 0 \notin S & \text{Subtour elimination} & \\ x_{ijk} \in \{0,1\} & & \end{array}$$

13.4.3. Clark Wright Algorithm

Borrowed from https://www.researchgate.net/publication/285833854_Chapter_4_Heuristics_for_the_Vehicle_Routing_Problem

The Clarke and Wright Savings Heuristic The Clarke and Wright heuristic [12] initially constructs back and forth routes $(0,i,0)$ for $(i = 1,\dots,n)$ and gradually merges them by applying a saving criterion. More specifically, merging the two routes $(0,\dots,i,0)$ and $(0, j,\dots,0)$ into a single route $(0,\dots,i, j,\dots,0)$ generates a saving $s_{ij} = c_{i0} + c_{0j} - c_{ij}$. Since the savings remain the same throughout the algorithm, they can be computed a priori. In the so-called parallel version of the algorithm which appears to be the best (see Laporte and Semet [46]), the feasible route merger yielding the largest saving is implemented at each iteration, until no more merger is feasible. This simple algorithm possesses the advantages of being intuitive, easy to implement, and fast. It is often used to generate an initial solution in more sophisticated algorithms. Several enhancements and acceleration procedures have been proposed for this algorithm (see, e.g., Nelson et al. [59] and Paessens [62]), but given the speed of today's computers and the robustness of the latest metaheuristics, these no longer seem justified.

[12] G. CLARKE AND J. W. WRIGHT, Scheduling of vehicles from a central depot to a number of delivery points, Operations Research, 12 (1964), pp. 568-581.

Resources

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.youtube.com/watch?v=LFeeaNP_rbY
Fantastic talk - Very thorough
https://www.opendoorlogistics.com/tutorials/tutorial-v-vehicle-routing-scheduling/

13.5 Steiner Tree Problem

Model 1

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

such that

$$x_t = 1 \quad \forall t \in T$$

$$2x_{uv} - x_u - x_v \leq 0 \quad \forall (u,v) \in E$$

$$x_v - \sum_{(u,v)\in E} x_{uv} \leq 0 \forall v \in V$$

$$\sum_{(u,v)\in\delta(S)} x_{uv} \geq x_w \forall S \subseteq V, \forall w \in S$$

Model 2

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

$$x_t = 1 \quad \forall t \in T$$

$$2x_{uv} - x_u - x_v \leq 0 \quad \forall (u,v) \in E$$

$$x_v - \sum_{(u,v)\in E} x_{uv} \leq 0 \quad \forall v \in V$$

$$x_{uv} + x_{vu} \leq 1$$

$$\sum_{v\in V} x_v - \sum_{(u,v)\in E} x_{uv} = 1$$

$$n x_{uv} + l_v - l_u \geq 1 - n(1 - x_{vu}) \quad \forall (u,v) \in E$$

$$n x_{vu} + l_u - l_v \geq 1 - n(1 - x_{uv}) \quad \forall (u,v) \in E$$

```
Optimizing the shop footprint:
Step 1: Machine learning model predicts effect of opening or closing shops
Step 2: ILP Breaks down shop into manageable slusters
Step 3:  ILP for optimal footprint planning
```

13.6 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

13.6.1. Google maps data

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

13.6.2. TSP In Excel

TSP with excel solver

14. Algorithms to Solve Integer Programs

Chapter 14. Algorithms to Solve Integer Programs
50% complete. Goal 80% completion date: September 20
Notes:

Outcomes

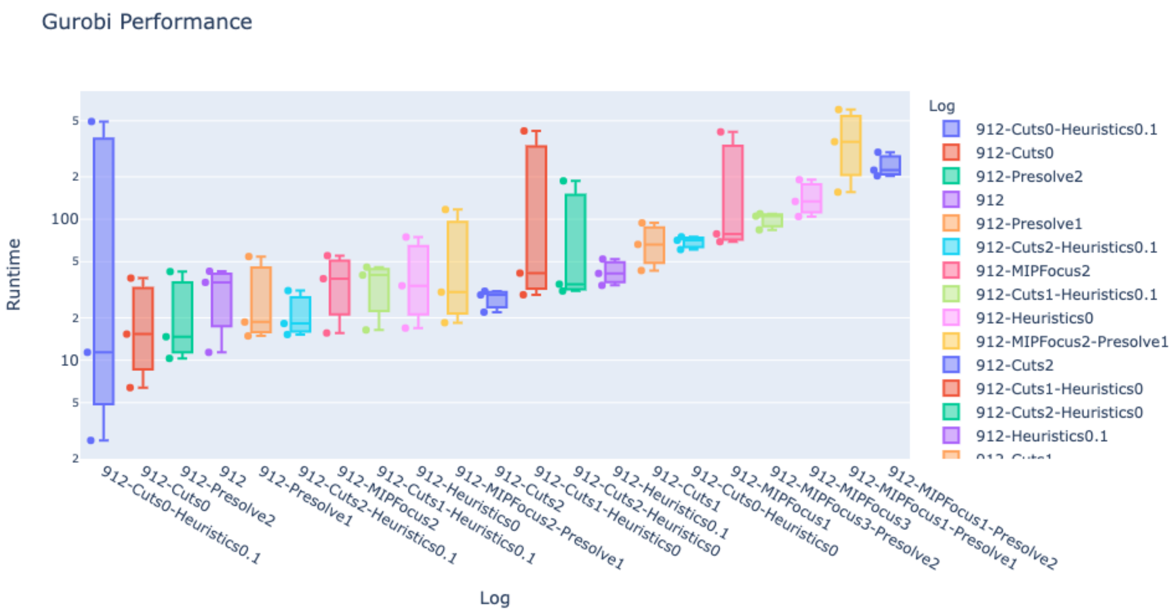
1. *Understand misconceptions in difficulty of integer programs*
2. *Learn basic concepts of algorithms used in solvers*
3. *Practice these basic concepts at an elementary level*
4. *Apply these concepts to understanding output from a solver*

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

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

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

[1] **gurobi_performance**, from **gurobi_performance**. **gurobi_performance**, **gurobi_performance**.



© gurobi_performance[1]

Figure 14.1: GUROBI Performance on a set of problems while varying different possible settings. This plot show the wild variablity of performance of different approaches. Thus, it is very unclear which is the "best" method. Furthermore, this plot can look quite different depending on the problem set one is working with. Altough we will not emphasize determining optimal settings in this text, we want to make clear that the techniques used in solvers are quite complicated and are tuned very carefully. We will study some elementary versions of techniques used in these solvers.

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

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

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

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

14.1 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: Linear Relaxation:

Example 14.2

Consider the problem

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

14.1.1. Rounding LP Solution can be bad!

Consider the two variable knapsack problem

$$\max 3x_1 + 100x_2 \tag{14.3}$$

$$x_1 + 100x_2 \leq 100 \tag{14.4}$$

$$x_i \in \{0,1\} \text{ for } i = 1,2. \tag{14.5}$$

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

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

Suppose that we rounded the LP solution.

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

How can we avoid this issue?

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

14.1.2. Rounding LP solution can be infeasible!

Now only could it produce a poor solution, it is not always clear how to round to a feasible solution.

14.1.3. Fractional Knapsack

The fractional knapsack problem has an exact greedy algorithm.

14.2 Branch and Bound

14.2.1. Algorithm

Algorithm 2 Branch and Bound - Maximization

Require: Integer Linear Problem with max objective

Ensure: Exact Optimal Solution x^*

1: Set $LB = -\infty$.
2: Solve LP relaxation.
  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.

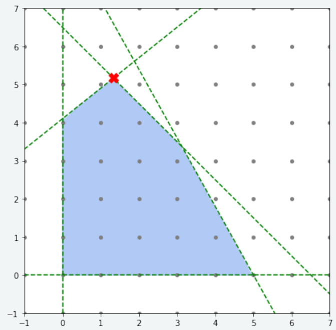
Here is an example of branching on general integer variables.

Example 14.3

Consider the two variable example with

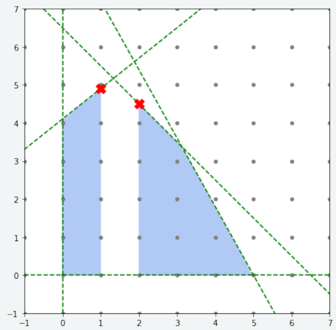
$$
\begin{aligned}
\max -3x_1 + 4x_2 & \\
2x_1 + 2x_2 &\leq 13 \\
-8x_1 + 10x_2 &\leq 41 \\
9x_1 + 5x_2 &\leq 45 \\
0 \leq x_1 &\leq 10, \textit{ integer} \\
0 \leq x_2 &\leq 10, \textit{ integer}
\end{aligned}
$$

$x = [1.33, 5.167] obj = 16.664$



© **branch-and-bound1**[2]

$x = [1, 4.9] obj = 16.5998$
$x = [2, 4.5] obj = 12.0$

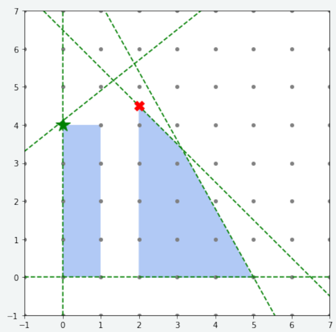


© **branch-and-bound2**[3]

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**.

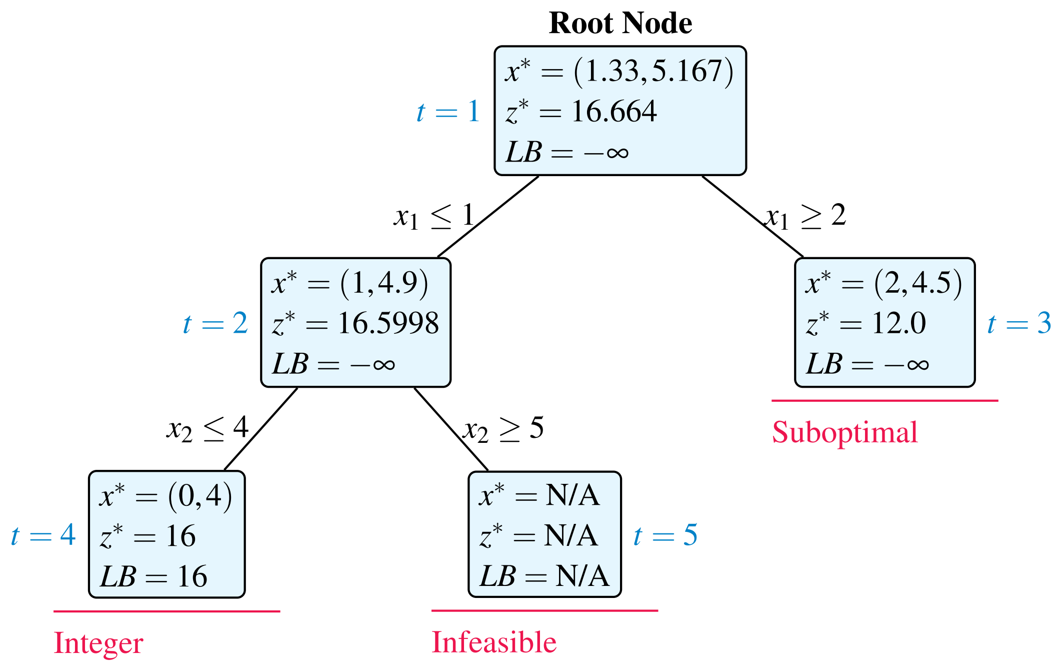
Example 14.4: Example continued

Infeasible Region
$x = [0, 4] obj = 16.0$
$x = [2, 4.5] obj = 12.0$



© **branch-and-bound3**[4]

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



14.2.2. Knapsack Problem and 0/1 branching

Consider the problem

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

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

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

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

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

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

Consider the problem

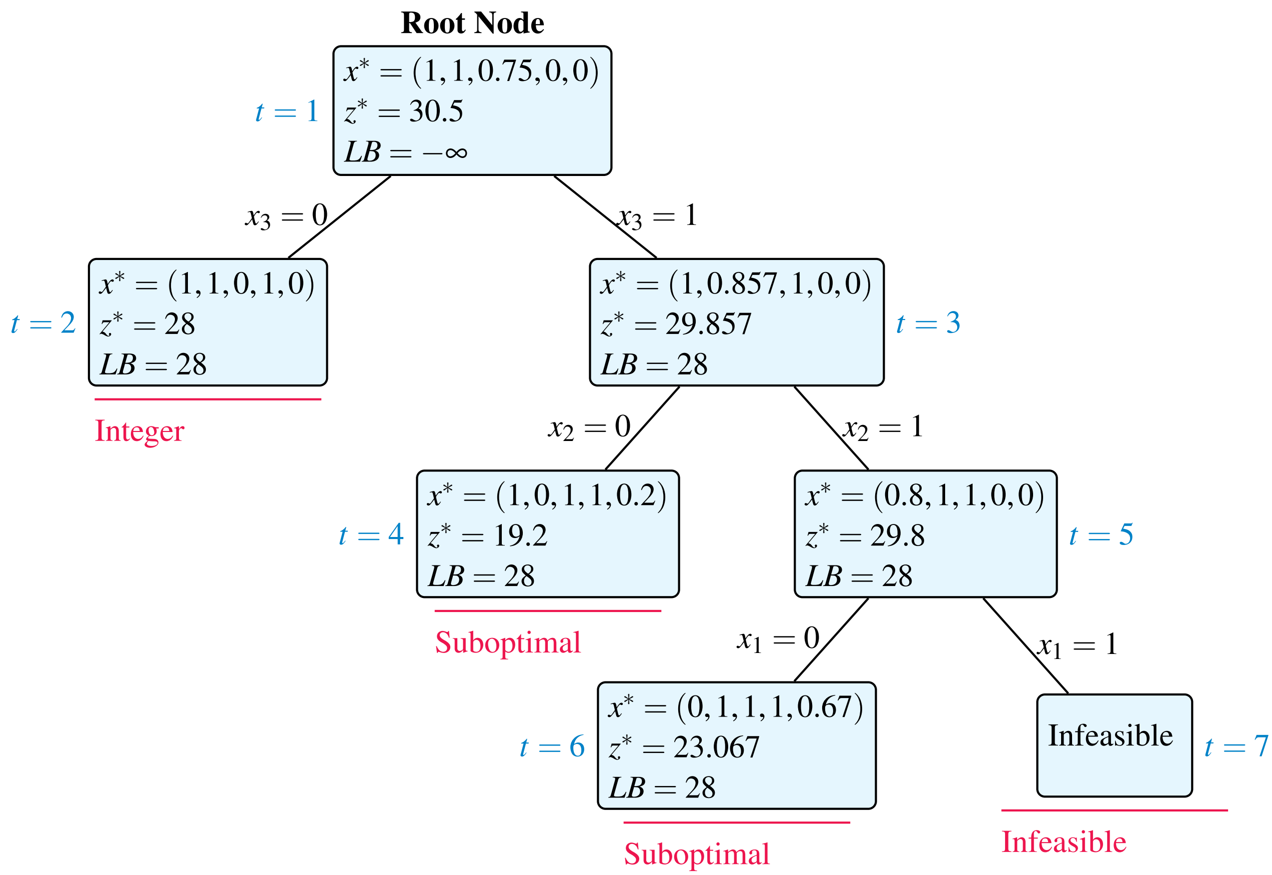
$$\begin{aligned} \max \quad & 16x_1 + 22x_2 + 12x_3 + 8x_4 \\ \text{s.t.} \quad & 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \\ & 0 \leq x_i \leq 1 \ \ i = 1,2,3,4 \\ & x_i \in \{0,1\} \ \ i = 1,2,3,4 \end{aligned}$$

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.

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



14.2.3. Traveling Salesman Problem solution via Branching

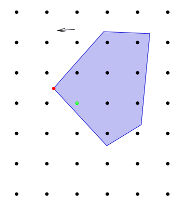
Describe solving TSP via a generalized branching method that removes subtours (instead of adding constraints).

14.3 Cutting Planes

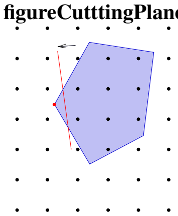
Cutting planes are inequalities $\pi^\top x \leq \pi_0$ that are valid for the feasible integer solutions that the cut off part of the LP relaxation. Cutting planes can create a tighter description of the feasible region that allows for the optimal solution to be obtained by simply solving a strengthened linear relaxation.

The cutting plane procedure, as demonstrated in Figure 20.2. The procedure is as follows:

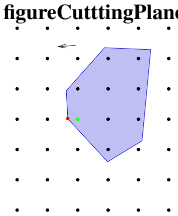
1. Solve the current LP relaxation.
2. If solution is integral, then return that solution. STOP
3. Add a cutting plane (or many cutting planes) that cut off the LP-optimal solution.
4. Return to Step 1.



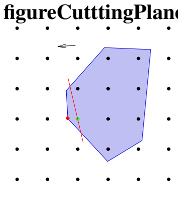
© figureCutttingPlane1[5]



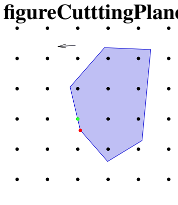
© figureCutttingPlane2[6]



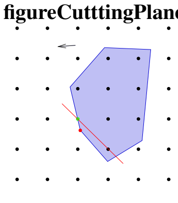
© figureCutttingPlane3[7]



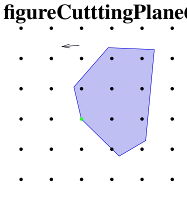
© figureCutttingPlane4[8]



© figureCutttingPlane5[9]



© figureCutttingPlane6[10]



© figureCutttingPlane7[11]

Figure 14.2: The cutting plane procedure.

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

14.3.1. Chvátal Cuts

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

Chvátal Cuts:

Suppose

$$a_1x_1 + \cdots + a_nx_n \leq d \tag{14.1}$$

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

$$\lfloor a_1 \rfloor x_1 + \cdots + \lfloor a_n \rfloor x_n \leq \lfloor d \rfloor \tag{14.2}$$

is valid for the integer points in P, that is, it is valid for the set $P \cap \mathbb{Z}^n$. Equation (20.2) is called a Chvátal Cut.

We will illustrate this idea with an example.

Example 14.5

Recall example **??**. *The model was*
Model

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

From the equality constraint we can derive several inequalities.

1. *Divide by 25 and round down both sides:*

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

2. *Divide by 10 and round down both sides:*

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

3. *Divide by 5 and round down both sides:*

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

4. *Multiply by 0.12 and round down both sides:*

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

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

New Model

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

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

14.3.2. Gomory Cuts

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

$$x_i + \sum_{i \in N} \tilde{a}_i x_i = \tilde{b}_i \tag{14.3}$$

is an equation in the optimal simplex tableau.

Gomory Cut:

The Gomory cut corresponding to the tableau row (20.3) is

$$\sum_{i \in N} (\tilde{a}_i - \lfloor \tilde{a}_i \rfloor) x_i \geq \tilde{b}_i - \lfloor \tilde{b}_i \rfloor \tag{14.4}$$

We will solve the following problem using only Gomory Cuts.

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

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

$$
\begin{array}{lrcl}
\min & x_1 - 2x_2 & & \\
\text{s.t.} & -4x_1 + 6x_2 + s_1 & = & 9 \\
& x_1 + x_2 + s_2 & = & 4 \\
& x \geq 0 & , & x_1, x_2 \in \mathbb{Z}
\end{array}
\tag{14.5}
$$

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

Initial Basis

Basis	RHS	x_1	x_2	s_1	s_2
z	0.0	1.0	-2.0	0.0	0.0
s_1	9.0	-4.0	6.0	1.0	0.0
s_2	4.0	1.0	1.0	0.0	1.0

⋮ ⋮

Optimal Basis

Basis	RHS	x_1	x_2	s_1	s_2
z	-3.5	0.0	0.0	0.3	0.2
x_1	1.5	1.0	0.0	-0.1	0.6
x_2	2.5	0.0	1.0	0.1	0.4

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

Example 14.6

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

$$x_1 - 0.1s_1 + -0.6s_2 = 1.5. \tag{14.6}$$

Applying the Gomory Cut (20.4), we have the inequality

$$0.9s_1 + 0.4s_2 \geq 0.5. \tag{14.7}$$

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 14.7: (Gomory Cut in Original Space)

The Gomory Cut (20.7) can be rewritten in the original variables using the equations from (20.5). That is, we can use the equations

$$
\begin{array}{rcl}
s_1 & = & 9 + 4x_1 - 6x_2 \\
s_2 & = & 4 - x_1 - x_2,
\end{array}
\tag{14.8}
$$

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

$$-3.2x_1 + 5.8x_2 \leq 9.2. \tag{14.9}$$

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

Example 14.8: (Gomory cuts plus new tableau)

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

$$0.9s_1 + 0.4s_2 - s_3 = 0.5. \tag{14.10}$$

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

14.3.3. Cover Inequalities

Consider the binary knapsack problem

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

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*

$$\sum_{i \in S} x_i \leq |S| - 1 \quad \Rightarrow \quad x_1 + x_2 + x_3 + x_4 \leq 4 - 1 = 3. \tag{14.11}$$

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

$$x_1 + x_2 \leq 2 - 1 = 1 \tag{14.12}$$

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

$$S = \{1,4\} \quad \Rightarrow \quad x_1 + x_4 \leq 1 \tag{14.13}$$

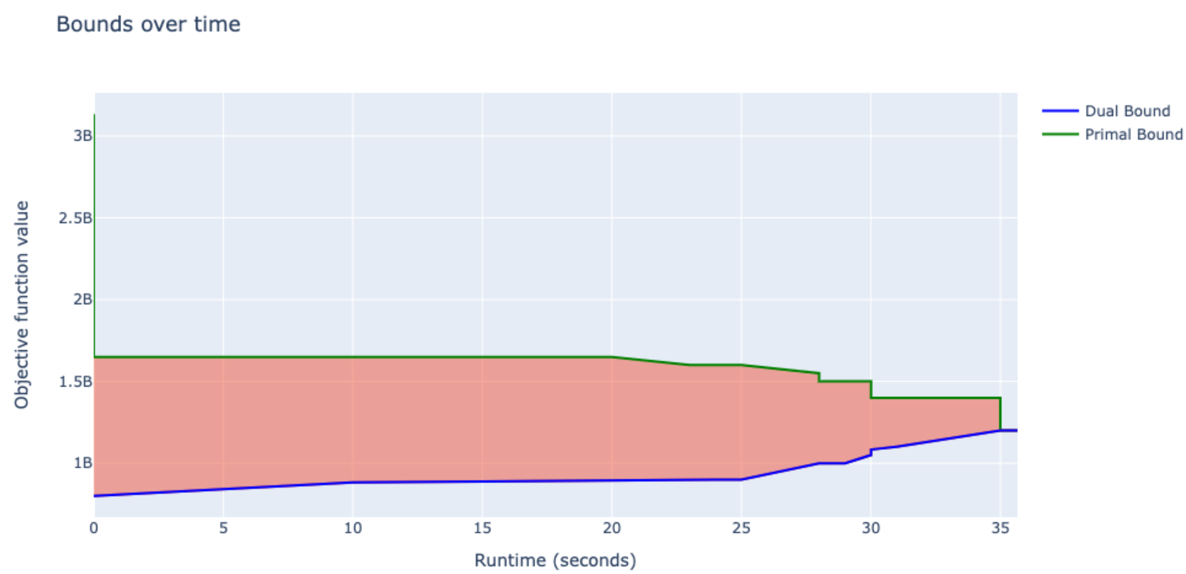
and

$$S = \{2,3,4\} \quad \Rightarrow \quad x_2 + x_3 + x_4 \leq 2 \tag{14.14}$$

♠

14.4 Interpreting Output Information and Progress

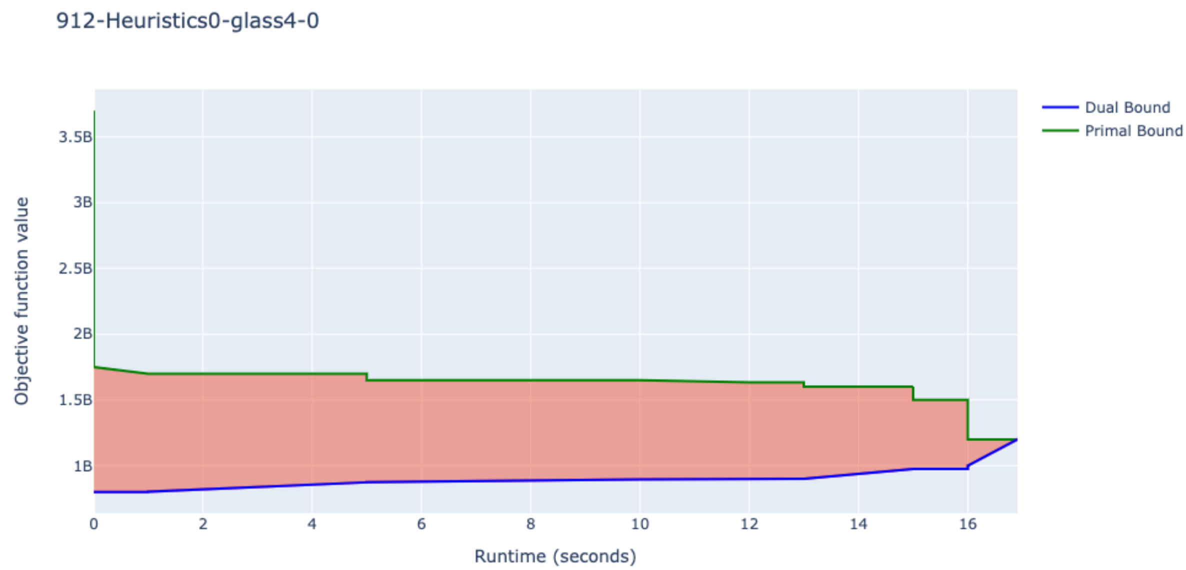
Section 14.4. Interpreting Output Information and Progress
Write this section. Include screenshot of a solver log



© **solve_progress1**[15]

Figure 14.3: This shows the progress of the solver over time.

[15]**solve_progress1**, from **solve_progress1**. **solve_progress1**, **solve_progress1**.
[16]**solve_progress2**, from **solve_progress2**. **solve_progress2**, **solve_progress2**.



© **solve_progress2**[16]

Figure 14.4: This shows the progress of the solver over time.

14.5 Branching Rules

These are advanced techniques that are not necessary at this point.

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

- Strong Branching
- Pseudo-cost Branching

14.6 Lagrangian Relaxation for Branch and Bound

This is an advanced technique that is not necessary to learn at this point.

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

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

14.7 Benders Decomposition

This is an advanced technique that is not necessary to learn at this point.

14.8 Literature and Resources

Resources

LP Rounding

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

Fractional Knapsack problem

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

Branch and Bound

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

Gomory Cuts

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

Benders Decomposition

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

14.9 Other material for Integer Linear Programming

Recall the problem on lemonade and lemon juice from Chapter 8.2:

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:

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

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

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

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 14.9

Let (MP) denote the following mixed-integer linear programming problem:

$$\begin{array}{rrcrcrcl} \min & x_1 & & & + & x_3 & & \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 & \geq & 1 \\ & -x_1 & - & x_2 & + & 2x_3 & \geq & 0 \\ & -x_1 & + & 5x_2 & - & x_3 & = & 3 \\ & x_1 & , & x_2 & , & x_3 & \geq & 0 \\ & & & & & x_3 & \in & \mathbb{Z}. \end{array}$$

The linear programming relaxation of (MP) is:

$$\begin{array}{rrcrcrcl} \min & x_1 & & & + & x_3 & & \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 & \geq & 1 \\ & -x_1 & - & x_2 & + & 2x_3 & \geq & 0 \\ & -x_1 & + & 5x_2 & - & x_3 & = & 3 \\ & x_1 & , & x_2 & , & x_3 & \geq & 0. \end{array}$$

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

$$\begin{array}{rrcrcrcl} \min & x_1 & & & + & x_3 & & \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 & \geq & 1 \\ & -x_1 & - & x_2 & + & 2x_3 & \geq & 0 \\ & -x_1 & + & 5x_2 & - & x_3 & = & 3 \\ & & & & & x_3 & \leq & 0 \\ & x_1 & , & x_2 & , & x_3 & \geq & 0, \end{array}$$

and (P3) is the problem

$$\begin{array}{rrcrcrcl} \min & x_1 & & & + & x_3 & & \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 & \geq & 1 \\ & -x_1 & - & x_2 & + & 2x_3 & \geq & 0 \\ & -x_1 & + & 5x_2 & - & x_3 & = & 3 \\ & & & & & x_3 & \geq & 1 \\ & x_1 & , & x_2 & , & x_3 & \geq & 0. \end{array}$$

Note that any feasible solution to (MP) must be a feasible solution to either (P2) or (P3). Using the help of a solver, one sees that (P2) is infeasible. The problem (P3) has an optimal solution at $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$, which is also feasible to (MP). Hence, $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$ is an optimal solution to (MP).

We now give a description of the method for a general mixed-integer linear programming problem (MIP). Suppose that (MIP) is a minimization problem and has n variables $x_1, \ldots, x_n$. Let $\mathscr{I} \subseteq \{1, \ldots, n\}$ denote the set of indices i such that x_i is required to be an integer in (MIP).

Branch-and-bound method

Input: The problem (MIP).

Steps:

1. Set `bestbound` $:= \infty$, $\mathbf{x}^*_{\text{best}} :=$ `N/A`, `activeproblems` $:= \{(LP)\}$ where (LP) denotes the linear programming relaxation of (MIP).

2. If there is no problem in `activeproblems`, then stop; if $\mathbf{x}^*_{\text{best}} \neq$ `N/A`, then $\mathbf{x}^*_{\text{best}}$ is an optimal solution; otherwise, (MIP) has no optimal solution.

3. Select a problem P from `activeproblems` and remove it from `activeproblems`.

4. Solve P.

- If P is unbounded, then stop and conclude that (MIP) does not have an optimal solution.

- If P is infeasible, go to step 2.

- If P has an optimal solution $\mathbf{x}^*$, then let z denote the objective function value of $\mathbf{x}^*$.

5. If $z \geq$ `bestbound`, go to step 2.

6. If x_i^* is not an integer for some $i \in \mathscr{I}$, then create two subproblems P_1 and P_2 such that P_1 is the problem obtained from P by adding the constraint $x_i \leq \lfloor x_i^* \rfloor$ and P_2 is the problem obtained from P by adding the constraint $x_i \geq \lceil x_i^* \rceil$. Add the problems P_1 and P_2 to `activeproblems` and go to step 2.

7. Set $\mathbf{x}^*_{\text{best}} = \mathbf{x}^*$, `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:
$$\begin{array}{rrcrcrcl} \min & x_1 & & & & & & \\ \text{s.t.} & x_1 & + & 2x_2 & - & 2x_3 & = & 1 \\ & x_1 & , & x_2 & , & x_3 & \geq & 0 \\ & x_1 & , & x_2 & , & x_3 & \in & \mathbb{Z}. \end{array}$$
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 $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{2}{3} \\ \frac{1}{3} \end{bmatrix}$	(P2): $x_3 \leq 0$, (P3): $x_3 \geq 1$	(P2), (P3)	N/A	∞
2	(P2)	infeasible	—	(P3)	N/A	∞
3	(P3)	optimal $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$	—	—	$\begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$	1

Exercises

1. Suppose that (MP) in Example 20.9 above has x_2 required to be an integer as well. Continue with the computations and determine an optimal solution to the modified problem.

2. With the help of a solver, determine the optimal value of

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

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

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

   a. Suppose that $\mathbf{d} \in \mathbb{Q}^m$ satisfies $\mathbf{d} \geq 0$ and $\mathbf{d}^\mathsf{T}\mathbf{A} \in \mathbb{Z}^n$. Prove that every $\mathbf{x}$ satisfying S also satisfies $\mathbf{d}^T\mathbf{Ax} \geq \lceil \mathbf{d}^\mathsf{T}\mathbf{b} \rceil$. (This inequality is known as a **Chvátal-Gomory cutting plane.**)

   b. Suppose that $\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 5 & 3 \\ 7 & 6 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} 2 \\ 1 \\ 8 \end{bmatrix}$. Show that every $\mathbf{x}$ satisfying S also satisfies $x_1 + x_2 \geq 2$.

Solutions

1. An optimal solution to the modified problem is given by $x^* = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

2. An optimal solution is $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$. Thus, the optimal value is 6.

3.
   a. Since $\mathbf{d} \geq 0$ and $\mathbf{Ax} \geq \mathbf{b}$, we have $\mathbf{d}^T\mathbf{Ax} \geq \mathbf{d}^\mathsf{T}\mathbf{b}$. If $\mathbf{d}^\mathsf{T}\mathbf{b}$ is an integer, the result follows immediately. Otherwise, note that $\mathbf{d}^\mathsf{T}\mathbf{A} \in \mathbb{Z}^n$ and $\mathbf{x} \in \mathbb{Z}^n$ imply that $\mathbf{d}^T\mathbf{Ax}$ is an integer. Thus, $\mathbf{d}^T\mathbf{Ax}$ must be greater than or equal to the least integer greater than $\mathbf{d}^\mathsf{T}\mathbf{b}$.

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

14.9.1. Other discrete problems

14.9.2. Assignment Problem and the Hungarian Algorithm

Assignment Problem:

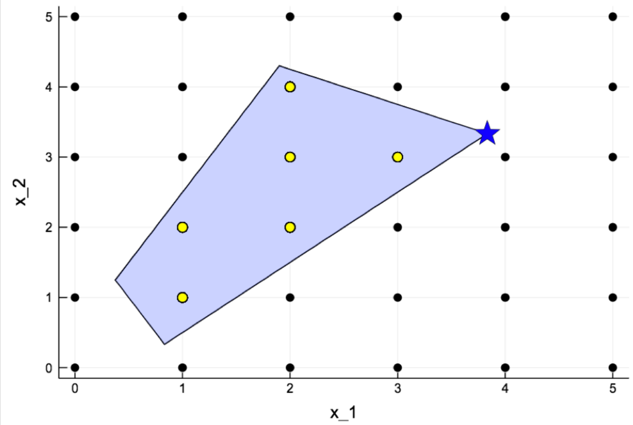
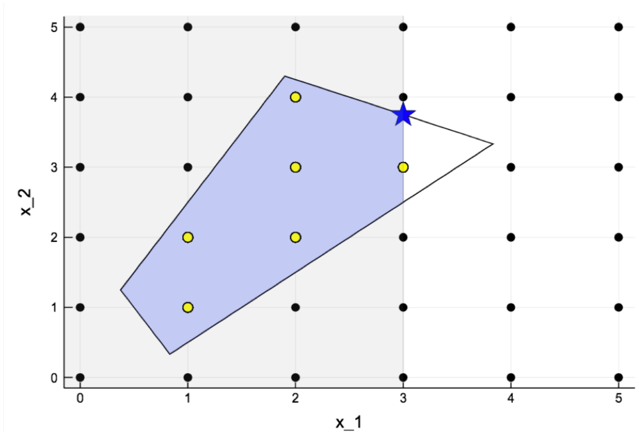
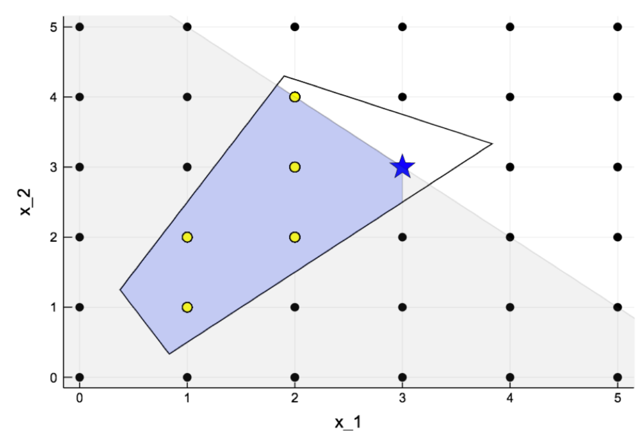
Polynomial time (P)

$$\begin{aligned} & \min \langle C, X \rangle \\ \text{s.t. } & \sum_i X_{ij} = 1 \quad \text{for all } j \\ & \sum_j X_{ij} = 1 \quad \text{for all } i \\ & X_{ij} \in \{0,1\} \text{ for all } i = 1, \dots n, j = 1, \dots, m \end{aligned} \tag{14.1}$$

This problem is efficiently solved by the Hungarian Algorithm.

14.9.3. History of Computation in Combinatorial Optimization

Book: Computing in Combinatorial Optimization by William Cook, 2019

Model	LP Solution
$\begin{array}{lrl} \max & x_1 + x_2 & \\ \text{subject to} & -2x_1 + x_2 & \leq 0.5 \\ & x_1 + 2x_2 & \leq 10.5 \\ & x_1 - x_2 & \leq 0.5 \\ & -2x_1 - x_2 & \leq -2 \end{array}$	 © **cutting-plane-1-picture**[12]
$\begin{array}{lrl} \max & x_1 + x_2 & \\ \text{subject to} & -2x_1 + x_2 & \leq 0.5 \\ & x_1 + 2x_2 & \leq 10.5 \\ & x_1 - x_2 & \leq 0.5 \\ & -2x_1 - x_2 & \leq -2 \\ & x_1 & \leq 3 \end{array}$	 © **cutting-plane-2-picture**[13]
$\begin{array}{lrl} \max & x_1 + x_2 & \\ \text{subject to} & -2x_1 + x_2 & \leq 0.5 \\ & x_1 + 2x_2 & \leq 10.5 \\ & x_1 - x_2 & \leq 0.5 \\ & -2x_1 - x_2 & \leq -2 \\ & x_1 & \leq 3 \\ & x_1 + x_2 & \leq 6 \end{array}$	 © **cutting-plane-3-picture**[14]

15. Heuristics for TSP

Chapter 15. Heuristics for TSP
50% complete. Goal 80% completion date: October 20
Notes:

Add code from GUROBI webinar with models and heuristic examples and show plots of improvements

Add links to resources from TSP video

Create graphics using `https://www.manim.community/` and `https://github.com/nipunramk/Reducible`.

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.

15.1 Construction Heuristics

15.1.1. 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 $i = 1, \ldots, n$, randomly choose a node not yet in the tour and place it at the end of the tour.

15.1.2. Nearest Neighbor

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

Nearest Neighbor:

Complexity: $O(n^2)$

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

15.1.3. Insertion Method

Insertion Method:

Complexity: $O(n^2)$

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

15.2 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.

15.2.1. 2-Opt (Subtour Reversal)

We will assume that all tours start and end with then node 1.

2-Opt (Subtour reversal):

Input a tour $1 \to \cdots \to 1$.

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

$$1 \to \cdots \to s \to i \to x_1 \to \cdots \to x_k \to j \to t \to \cdots \to 1.$$

3. Consider the subtour reversal

$$1 \to \cdots \to s \to j \to x_k \to \cdots \to x_1 \to i \to t \to \cdots \to 1.$$

   Thus, we reverse the order of $i \to x_1 \to \cdots \to x_k \to j$.

4. In this process, we
   - deleted the edges (s,i) and (j,t)
   - added the edges (s,j) and (i,t)

Pictorially, this looks like the following

Figure 21.1

Computationally, we need to consider the costs on the edges of a graph.....

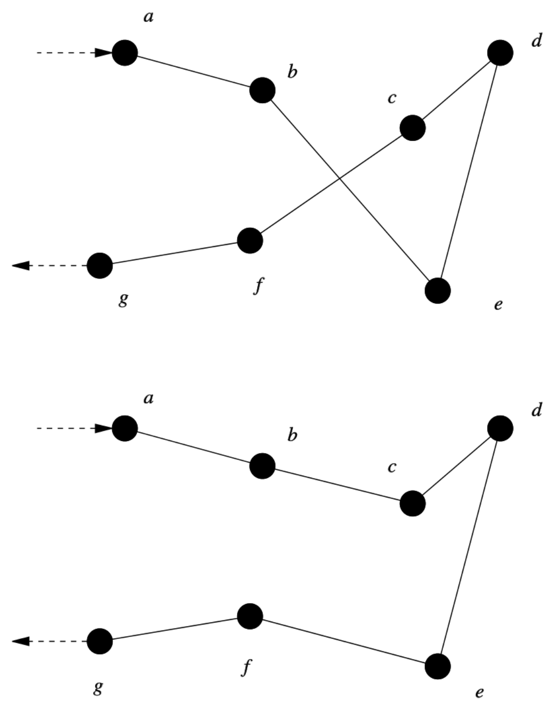
See [**Englert2014**] for an analysis of performance of this improvement.

15.2.2. 3-Opt

15.2.3. k-Opt

This is a generalization of 2-Opt and 3-Opt.

[1] **wiki/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.png[1]
Figure 15.1: wiki/File/2-opt_wiki.png

15.3 Meta-Heuristics

15.3.1. 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.

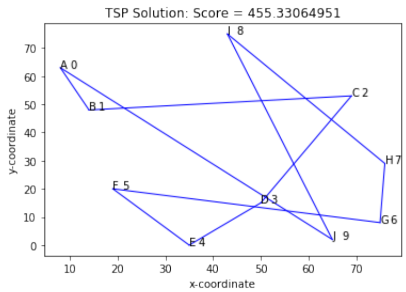
In the context of TSP, we will consider 2-Opt improving moves and the Hill Climbing algorithm for TSP in this case is referred to as the 2-Opt algorithm (also known as the Subtour Reversal Algorithm).

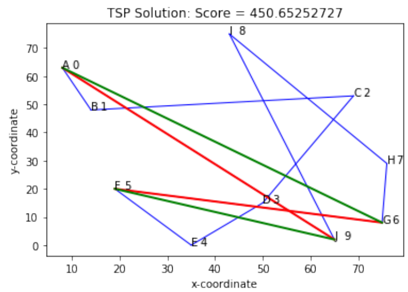
Hill Climbing:

1. Start with an initial feasible solution, label it as the current solution.
2. List all neighbors of the current solution.
3. If no neighbor has a better solution, then stop.
4. Otherwise, move to the best neighbor and go to Step 2.

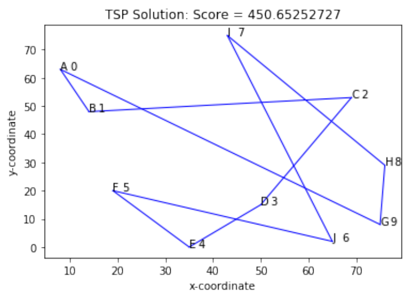
Here is an example on the TSP problem with 2-Opt swaps:

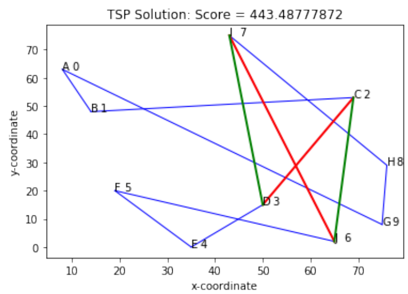
Current solution	Improvement Step	Reversal



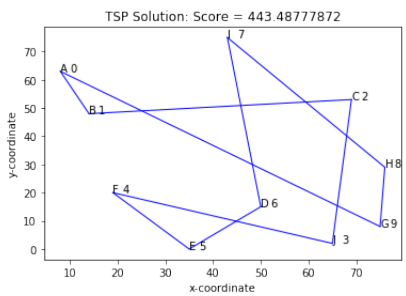


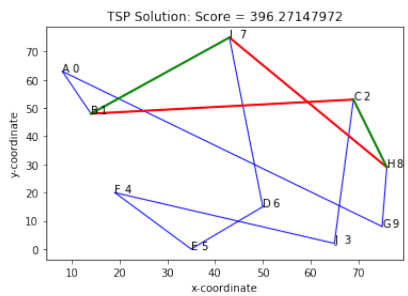
[A, B, C, D, E, F, **G, H, I, J**]



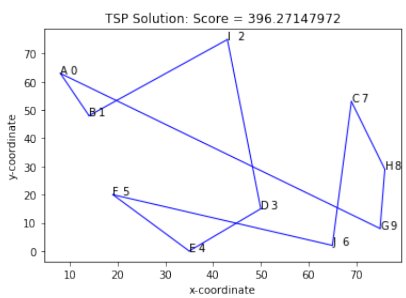


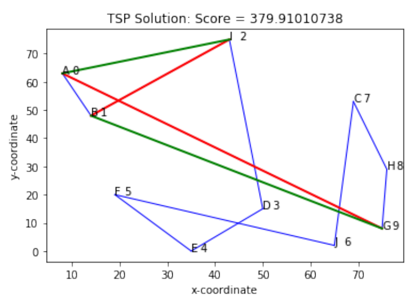
[A, B, C, **D, E, F, J**, I, H, G]



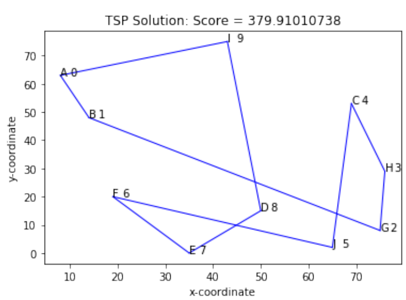


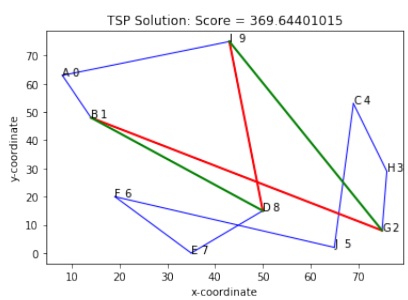
[A, B, **C, J, F, E, D, I**, H, G]



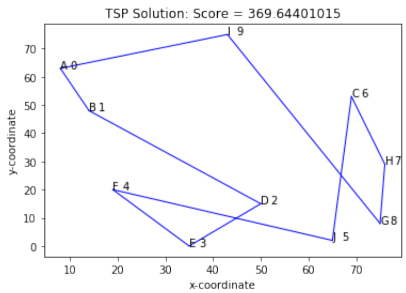


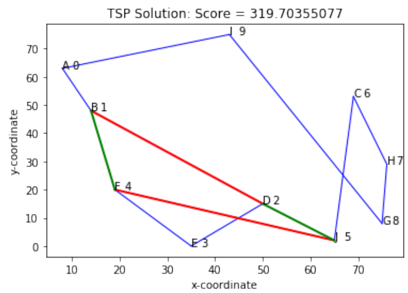
[A, B, **I, D, E, F, J, C, H, G**]



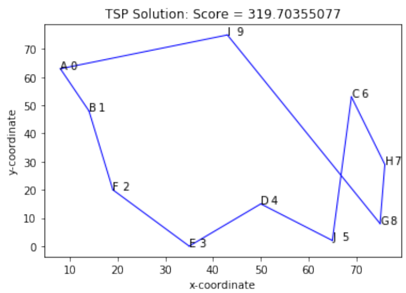


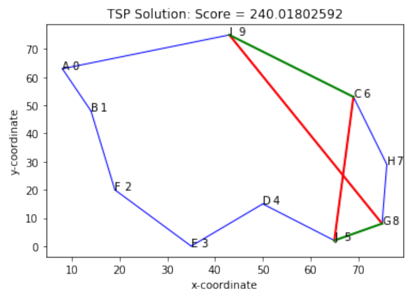
[A, B, **G, H, C, J, F, E, D**, I]



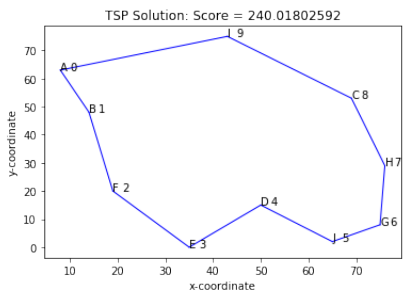


[A, B, **D, E, F**, J, C, H, G, I]





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



No Improvement [A, B, F, E, D, J, G, H, C, I]

15.3.2. Simulated Annealing

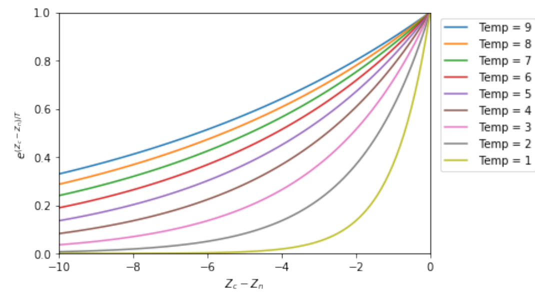
Here is a great python package for TSP Simulated annealing: `https://pypi.org/project/satsp/`.

Simulated annealing is a randomized heuristic that randomly decides when to accept non-improving moves. For this, we use what is referred to as a *temperature schedule*. The temperature schedule guides a parameter T that goes into deciding the probability of accepting a non-improving move.

A typically temperature schedule starts at a value $T = 0.2 \times Z_c$, where Z_c is the objective value of an initial feasible solution. Then the temperature is decreased over time to smaller values.

Temperature schedule example:

- $T_1 = 0.2Z_c$
- $T_2 = 0.8T_1$
- $T_3 = 0.8T_2$
- $T_4 = 0.8T_3$
- $T_5 = 0.8T_4$



© **simulated_annealing_temperatures**[2]

Figure 15.2: simulated_annealing_temperatures

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]

1. Start with an initial feasible solution, label it as the current solution, and compute its object value Z_c.
2. Select a neighbor of the current solution and compute its objective value Z_n.
3. Decide whether or not to move to the neighbor:
   (a) If the neighbor is an improvement ($Z_n < Z_c$), **accept the move** and set the neighbor as the current solution.

   (b) 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.
4. While still iterations left in the schedule, update the temperature T and go to Step 2.
5. Return the best found solution during the algorithm.

Figure 21.2

[2] **simulated_annealing_temperatures**, from **simulated_annealing_temperatures**. **simulated_annealing_temperatures**, **simulated_annealing_temperatures**.

15.3.3. Tabu Search

Connect to code example for general tabu search

Tabu Search Outline:

[minimization version]

1. Initialize a *Tabu List* as an empty set: Tabu $= \{\}$.
2. Start with an initial feasible solution, label it as the current solution.
3. List all neighbors of the current solution.
4. Choose the best neighbor that is not tabu to move too (the move should not be restricted by the set Tabu.)
5. Add moves to the Tabu List.
6. If the Tabu List is longer than its designated maximal size S, then remove old moves in the list until it reaches the size S.
7. If no object improvement has been seen for K steps, then Stop.
8. Otherwise, Go to Step 3 and continue.

15.3.4. 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

15.3.5. 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..

15.3.6. Ant Colony Optimization

Wikipedia - Ant Colony Optimization

15.4 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 Annleaning
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 Annleaning
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 Annleaning
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)
```

15.4.1. VRP - Clark Wright Algorithm

Include discussion of Clark Wright algorihtm, or link to earlier section on Algorithms

Resources and References

Resources

- *Amazing video covering all TSP and topics in this section*
- *Interactive tutorial of TSP algorithms*
- *TSP Simulated Annealing Video with fun music.*
- *VRP Heuristic Approach Lecture by Flipkart Delivery Hub*
- `https://github.com/Gurobi/pres-mipheur` - *Gurobi coded heuristics for TSP with comparison.*