Department of Mathematics and Computer Science
University of Southern Denmark, Odense

May 1, 2025
Marco Chiarandini

AI505 – Optimization

Sheet 09, Spring 2025

Solution:

Included.

Exercises with the symbol $^+$ are to be done at home before the class. Exercises with the symbol * will be tackled in class. The remaining exercises are left for self training after the exercise class. Some exercises are from the text book and the number is reported. They have the solution at the end of the book.

Exercise 1^+ (19.2)

A Boolean satisfiability problem, often abbreviated SAT, requires determining whether a Boolean design exists that causes a Boolean-valued objective function to output true. SAT problems were the first to be proven to belong to the difficult class of NP-complete problems. This means that SAT is at least as difficult as all other problems whose solutions can be verified in polynomial time.
Consider the Boolean objective function:

$$f(\boldsymbol{x}) = x_1 \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$$

Formulate the problem as an integer linear program. Can any Boolean satisfiability problem be formulated as an integer linear program? Solve the problem with scipy.

Solution:

Enumeration tries all designs. Each component can either be true or false, thus resulting in 2^n possible designs in the worst case. This problem has $2^3 = 8$ possible designs.
The Boolean satisfiability problem merely seeks a valid solution. As such, we set c to zero. The constraints are more interesting. As with all integer linear programs, x is constrained to be nonnegative and integral. Furthermore, we let 1 correspond to true and 0 correspond to false and introduce the constraint $\boldsymbol{x} \leq \mathbf{1}$. This is equivalent to impose that $\boldsymbol{x} \in \mathbb{B}^n$.
...

Exercise 2^*

Consider $\max\{\boldsymbol{c}^T\boldsymbol{x} \mid A\boldsymbol{x} = \boldsymbol{b}, \boldsymbol{x} \in \mathbb{Z}^n\}$. Sometimes the solution of the linear relaxation is already integral. Can you find a sufficient condition for the matrix A for that to happen?

Solution:

From the simplex

$$A_B\boldsymbol{x}_B + A_N\boldsymbol{x}_N = \boldsymbol{b}$$

The variables not in basis are set to zero while for a feasibel basis the value of the variables in basis can be calculated as:

$$\boldsymbol{x}_B = A^{-1}\boldsymbol{b}$$

All coefficients in A and $\boldsymbol{b}$ are integer. Moreover,

$$\mathsf{A}^{-1} = \frac{1}{\det(\mathsf{A})}\operatorname{adj}(\mathsf{A}).$$

where adj is the adjugate of A, which is the transpose of the cofactor matrix C of A. The cofactor matrix of A is the $n \times n$ matrix C whose (i, j) entry is the (i, j) cofactor of A, which is the (i, j)-minor $M_{i,j}$ times a sign factor:

$$\mathsf{C} = \left((-1)^{i+j}\mathsf{M}_{ij}\right)_{1\leq i,j\leq n}.$$

The element M_{ij}, is the (i,j)-minor of A and corresponds to the determinant of the $(n-1)\times(n-1)$ matrix that results from deleting row i and column j of A.
Hence, the numerator of A^{-1} is a matrix whose absolute values of its elements are products and subtractions of integer numbers, the elements of A, and must therefore be integers.
Hence, if the denominator of A^{-1}, ie, $|A|$ is an integer number, the values of A^{-1} will be integer and x_B integer. This is the case if the matrix A_B has a determinant equal to ± 1.
Hence if all square submatrices of the matrix A have determinant equal to $0, +1, -1$ the value of the basis variables will be integer.

Exercise 3*

Consider the following problem:

$$\begin{aligned} &\min x_1 + x_2 \\ \text{subject to: } &\frac{2}{3}x_1 + \frac{1}{2}x_2 \geq 1 \\ &x_1, x_2 \geq 0 \\ &x_1, x_2 \in \mathbb{Z} \end{aligned}$$

Derive a Chvatal-Gomory cut.

Solution:
Relaxing the integrality constraint we have that the basis solutions are:

$$x_1 = 0 : \frac{1}{2}x_2 \geq 1 \implies x2_{\geq}2$$

$$x_2 = 0 : \frac{2}{3}x_1 \geq 1 \implies x_1 \geq 1.5$$

Thus, in LP relaxation, x1=1.5,x2=0 is feasible, but x1=1.5 is fractional — not acceptable for the ILP!

Exercise 4

Apply the linear programming branch and bound algorithm to the following instance of the 0-1 knapsack problem: values $v = [9, 4, 2, 3, 5, 3]$, weights $w = [7, 8, 4, 5, 9, 4]$ and capacity $W = 20$.

Solution:
The relaxed knapsack problem can be efficiently solved with a greedy approach. Items are added one at a time by selecting the next item with the greatest ratio of value to weight. If there is enough remaining capacity, the item is fully assigned with $x_i = 1$. If not, a fractional value is assigned such that the remaining capacity is saturated and all remaining items have $x_i = 0$.

item:	6	4	5	3	2
ratio:	3/4	3/5	5/9	2/4	4/8
	0.75	0.6	0.556	0.5	0.5

Exercise 5

Consider the problem of selecting students for a swimming medley relay team. In Table 1 we show times for each swimming style of five students.
We need to choose a student for each of the four swimming styles such that the total relay time is minimized. Formulate the problem as a MILP and solve it in Python.

Exercise 6

In this exercise, you have to implement the dynamic programming algorithm for TSP with the help of the code below. How far can you go solving the TSP to optimality by enumeration (implemented below) and with dynamic programming?
Some typing preliminaries. Cities are points in the 2D plane represented by complex numbers, a builtin type for a two-dimensional data.

Student	backstroke	breaststroke	butterfly	freestyle
A	43.5	47.1	48.4	38.2
B	45.5	42.1	49.6	36.8
C	43.4	39.1	42.1	43.2
D	46.5	44.1	44.5	41.2
E	46.3	47.8	50.4	37.2

Table 1:

```
import functools
import itertools
import pathlib
import random
import time
import math
import re
import matplotlib.pyplot as plt
from collections import Counter, defaultdict, namedtuple
from statistics import mean, median, stdev
from typing import Set, List, Tuple, Iterable, Dict


# Basic concepts
City = complex # e.g. City(300, 100)
Cities = frozenset # A set of cities
Tour = list # A list of cities visited, in order
TSP = callable # A TSP algorithm is a callable function
Link = Tuple[City, City] # A city-city link

def distance(A: City, B: City) -> float:
    "Distance between two cities"
    return abs(A - B)

def shortest(tours: Iterable[Tour]) -> Tour:
    "The tour with the smallest tour length."
    return min(tours, key=tour_length)

def tour_length(tour: Tour) -> float:
    "The total distances of each link in the tour, including the link from last back to
        first."
    return sum(distance(tour[i], tour[i - 1]) for i in range(len(tour)))

def valid_tour(tour: Tour, cities: Cities) -> bool:
    "Does ‘tour‘ visit every city in ‘cities‘ exactly once?"
    return Counter(tour) == Counter(cities)
```

Then we can randomly generate a problem instance as follows.

```


# generate instance
def random_cities(n, seed=1234, width=9999, height=6666) -> Cities:
    "Make a set of n cities, sampled uniformly from a (width x height) rectangle."
    random.seed(seed)
    return Cities(City(random.randrange(width), random.randrange(height))
                  for c in range(n))
```

We can find optimal solutions by exhaustive search, ie, enumeration of all permutations of points.

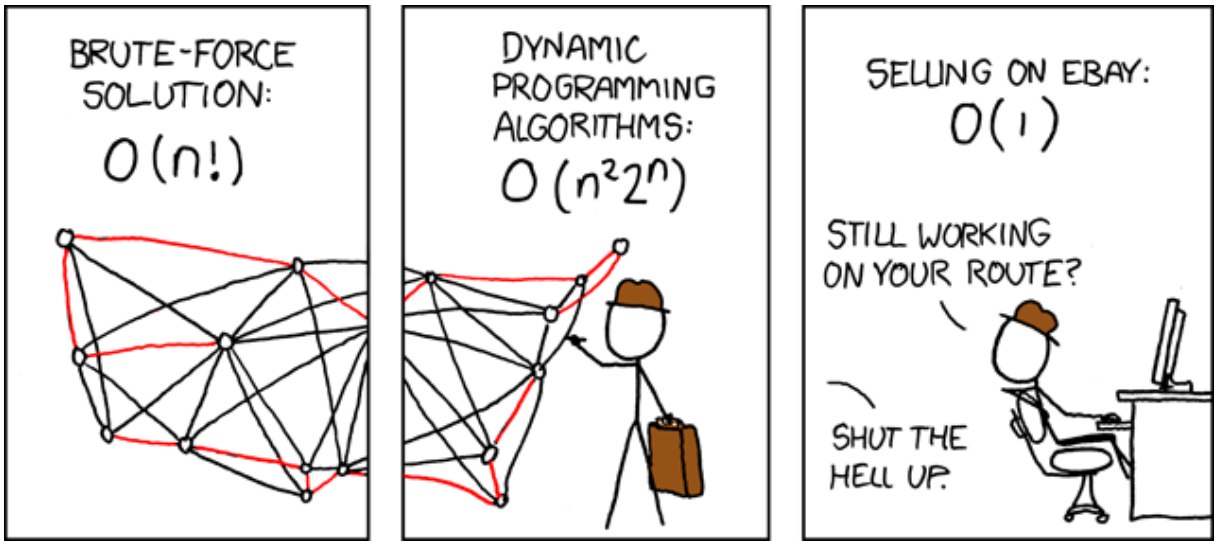

Figure 1: https://xkcd.com/399/

```
def exhaustive_tsp(cities) -> Tour:
    "Generate all possible tours of the cities and choose the shortest one."
    return shortest(itertools.permutations(cities))


instance = random_cities(8, seed=42, width=100, height=100)
#optimal_tour = exhaustive_tsp(instance)
#print("Optimal tour:", optimal_tour)
#print("Tour length:", tour_length(optimal_tour))
```

Finally, we can visualize the tour.

```
Segment = list # A portion of a tour; it does not loop back to the start.

def plot_tour(tour: Tour, style='bo-', hilite='rs', title=''):
    "Plot every city and link in the tour, and highlight the start city."
    scale = 1 + len(tour) ** 0.5 // 10
    plt.figure(figsize=((3 * scale, 2 * scale)))
    start = tour[0]
    plot_segment([*tour, start], style)
    plot_segment([start], hilite)
    plt.title(title)

def Xs(cities) -> List[float]: "X coordinates"; return [c.real for c in cities]
def Ys(cities) -> List[float]: "Y coordinates"; return [c.imag for c in cities]

def plot_segment(segment: Segment, style='bo:'):
    "Plot every city and link in the segment."
    plt.plot(Xs(segment), Ys(segment), style, linewidth=2/3, markersize=4, clip_on=False)
    plt.axis('scaled'); plt.axis('off')


#plot_tour(optimal_tour)
```

Solution:

```
cache = functools.cache

def first(iterable):
```

```
    "Return the first item in the iterable."
    return next(iter(iterable))

def held_karp_tsp(cities) -> Tour:
    """The Held-Karp shortest tour of this set of cities.
    For each end city C, find the shortest segment from A (the start) to C.
    Out of all these shortest segments, pick the one that is the shortest tour."""
    A = first(cities)
    shortest_segment.cache_clear() # Clear cache for a new problem
    return shortest(shortest_segment(A, cities - {A, C}, C)
                    for C in cities - {A})

@cache
def shortest_segment(A, Bs, C) -> Segment:
    "The shortest segment starting at A, going through all Bs, and ending at C."
    if not Bs:
        return [A, C]
    else:
        return min((shortest_segment(A, Bs - {B}, B) + [C] for B in Bs),
                   key=segment_length)

def segment_length(segment):
    "The total of distances between each pair of consecutive cities in the segment."
    # Same as tour_length, but without distance(tour[0], tour[-1])
    return sum(distance(segment[i], segment[i-1])
               for i in range(1, len(segment)))



def run(tsp: callable, cities: Cities):
    """Run a TSP algorithm on a set of cities and plot/print results."""
    t0 = time.perf_counter()
    tour = tsp(cities)
    t1 = time.perf_counter()
    L = tour_length(tour)
    print(f"length {round(L):,d} tour of {len(cities)} cities in {t1 - t0:.3f} secs")
    plot_tour(tour)
    plt.savefig("tsp_run.png")

# Run the TSP algorithms
run(held_karp_tsp, instance)
run(exhaustive_tsp, instance)
```