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

May 15, 2025
Marco Chiarandini

AI505 – Optimization

Sheet 10, 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$^+$

Model and solve in MiniZinc the bin packing problem: you are given a set of items each with a size and a set of bins each with a capacity that limits the total size of the items that can be placed in it. Your task is to place the items in the minimum number of bins such that the capacity constraint is not exceeded. Focus on modelling the problem with both arrays of variables and arrays of set variables. Calculate a lower and upper bound on the number of bins before starting the solution process.
Document yourself about MiniZinc using the references listed in Unit 7.
Solve the two following problem instances:
Small:

```
int: cap = 10;
int: num_bins = 11;
array[1..n] int: size = [6, 6, 6, 5, 3, 3, 2, 2, 2, 2, 2];
```

Large:

```
int: cap = 100;
int: num_bins = 50;
array[1..n] int: size = [
  99,98,95,95,95,94,94,91,88,87,86,85,76,74,73,71,68,60,55,54,51,
  45,42,40,39,39,36,34,33,32,32,31,31,30,29,26,26,23,21,21,21,19,
  18,18,16,15,5,5,4,1];
```

An optimal solution for the small instance:

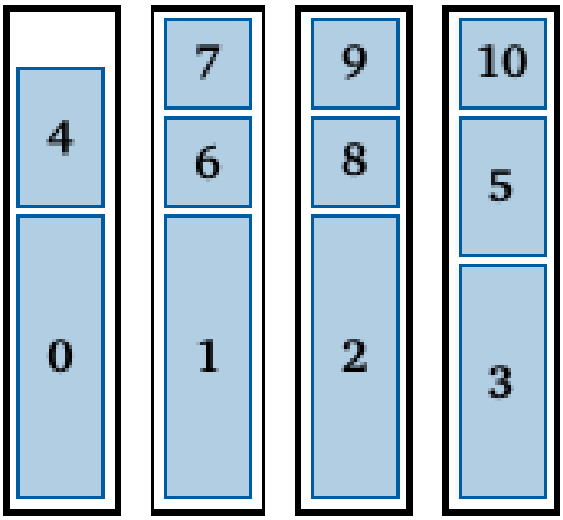
Solution:

Objects of different height must be packed into a finite number of bins or containers each of height H in a way that minimizes the number of bins used.
Model the problem and solve the following specific instance:

```
num_objs = 6;
objs = [360, 850, 630, 70, 700, 210]; % heights of objects
bin_capacity = 1440; % height of bins
```

Let m be the number of bins and n the number of items

Variables:
binary variables to represent for each bin whether the object is packed or not
$x_{ij} \in \mathbb{B}^{m \times n}$ for $i \in [1..m]$ and $j \in [1..n]$
Auxiliary variables to represent the load of a bin.




```
% binary variables
array[1..num_bins, 1..num_stuff] of var 0..1: bins;
% calculate how many things a bin takes
array[1..num_bins] of var 0..bin_capacity: bin_loads;


% number of loaded bins (which we will minimize)
var 0..num_bins: num_loaded_bins;

% minimize the number of loaded bins
% solve minimize num_loaded_bins;

% alternative solve statement
solve :: int_search(
         [bins[i,j] | i in 1..num_bins, j in 1..num_stuff], % ++ bin_loads
         input_order, % first_fail,
         indomain_max,
         complete)
         minimize num_loaded_bins;

constraint
    % sanity clause: No thing can be larger than capacity.
    % forall(s in 1..num_stuff) (
    %    stuff[s] <= bin_capacity
    % )
    % /\ % the total load in the bin cannot exceed bin_capacity
    forall(b in 1..num_bins) (
       bin_loads[b] = sum(s in 1..num_stuff) (size[s]*bins[b,s])
    )
    /\ % calculate the total load for a bin
    sum(s in 1..num_stuff) (size[s]) = sum(b in 1..num_bins) (bin_loads[b])
    /\ % a thing is packed just once
    forall(s in 1..num_stuff) (
       sum(b in 1..num_bins) (bins[b,s])  = 1
    )
    % /\ % symmetry breaking:
    %    % if bin_loads[i+1] is > 0 then bin_loads[i] must be > 0
    %  forall(b in 1..num_bins-1) (
    %    (bin_loads[b+1] > 0 -> bin_loads[b] > 0)
    %    % /\ % and should be filled in order of weight
    %    % bin_loads[b] >= bin_loads[b+1]
    %  )
    /\
  decreasing(bin_loads) :: domain
    /\ % another symmetry breaking: first bin must be loaded
    bin_loads[1] > 0
    /\ % calculate num_loaded_bins
    num_loaded_bins = sum(b in 1..num_bins) (bool2int(bin_loads[b] > 0))
```

Exercise 2*

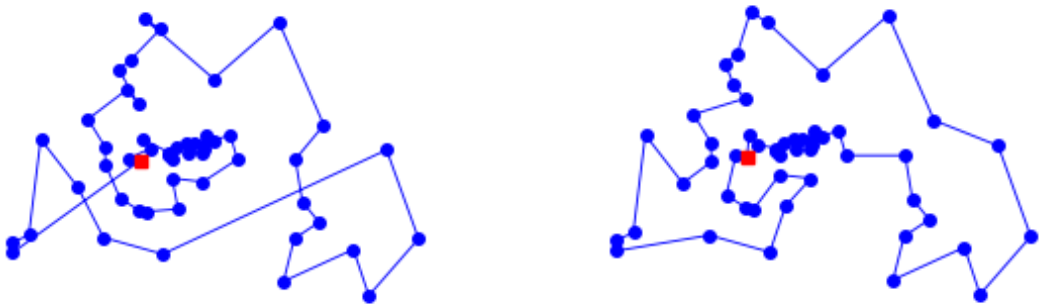
Build an heuristic solver for the Traveling Salesman Problem using the ROARNET-API specification. You can reuse the code from exercise 6 of Sheet 09 to generate an instance and to represent the problem and implement the API Specification on your own or you can use the one available from the course web

page.

Solution:

The full solution is available at link.
Restricting to the parts that were left to implement:

```python
#!/usr/bin/env python3
#
# SPDX-FileCopyrightText: 2025 Carlos M. Fonseca <cmfonsec@dei.uc.pt>
#
# SPDX-License-Identifier: Apache-2.0


from api.base import Neighbourhood, Move
import math
from api.utils import sparse_fisher_yates_iter


# ------------------------------ Neighbourhood ------------------------------

class TwoOptNeighbourhood(Neighbourhood):
   def moves(self, solution):
      assert self.problem == solution.problem
      # raise NotImplementedError("Not implemented yet")
      n = self.problem.n
      if solution.is_feasible():
         for ix in range(1, n-1):
            for jx in range(ix + 2, n + (ix != 1)):
               yield TwoOptMove(self, ix, jx)


   def random_moves_without_replacement(self, solution):
      assert self.problem == solution.problem
      # The Fisher-Yates shuffle algorithm is used to generate a random
      n = self.problem.n
      if solution.is_feasible():
         for x in sparse_fisher_yates_iter(n*(n-3)//2):
            jx = (1 + math.isqrt(1+8*x))//2
            ix = x - jx*(jx-1)//2 + 1
            jx += 2
            # Handle special case
            if ix == 1 and jx == n:
               ix = n - 2
            yield TwoOptMove(self, ix, jx)


# ------------------------------------ Move ------------------------------------

class TwoOptMove(Move):
   def __init__(self, neighbourhood, ix, jx):
      self.neighbourhood = neighbourhood
      # ix and jx are indices
      self.ix = ix
      self.jx = jx

   def __str__(self):
      return "%d:%d" % (self.i, self.j)

   def apply(self, solution):
      prob = solution.problem
      # Update tour length
      # Update solution
      # raise NotImplementedError("Not implemented yet")
      n, ix, jx = prob.n, self.ix, self.jx
```



```
        # Update tour length
        t = solution.tour
        solution.lb -= prob.dist[t[ix-1]][t[ix]] + prob.dist[t[jx-1]][t[jx % n]]
        solution.lb += prob.dist[t[ix-1]][t[jx-1]] + prob.dist[t[ix]][t[jx % n]]
        # Update solution
        solution.tour[ix:jx] = solution.tour[ix:jx][::-1]

    def objective_value_increment(self, solution):
        prob = solution.problem
        # Tour length increment
        #raise NotImplementedError("Not implemented yet")
        n, ix, jx = prob.n, self.ix, self.jx
        # Tour length increment
        t = solution.tour
        incr = prob.dist[t[ix-1]][t[jx-1]] + prob.dist[t[ix]][t[jx % n]]
        incr -= prob.dist[t[ix-1]][t[ix]] + prob.dist[t[jx-1]][t[jx % n]]
        return incr
```