EITN95: Simulation
Take Home Exam - Report

Niklas Vogel
980610T516

June 13, 2022

https://github.com/nikvogel/SimulationExam
(public after submission deadline)

Contents

1 Optimization - Product Mix Problem

1.1 Problem formulation

The problem of this task consists of finding the optimal product mix between five different cameras in order to maximize the profit considering production and demand constraints. Note: The assignment states that the profit should be maximized. As only the sales prices are given and not the costs I assume that sales price refers to the margin.
The problem shortly described above can be formulated as a Linear Program (LP). I is the set of camera types and x_i states the sold cameras of type $i \in I$.

$$
\begin{array}{llr}
\textbf{max} & 4x_1 + 3x_2 + 2x_3 + 2x_4 + x_5 & \\
 & 2x_1 & \leq 36000 \\
 & 2x_2 + 2x_3 + 3x_4 + x_5 & \leq 216000 \\
 & 0.2x_1 + x_2 + 0.5x_4 & \leq 18000 \\
\textbf{s.t.} & x_1 & \leq 16000 \\
 & x_3 & \leq 2000 \\
 & x_1 + x_2 + x_3 & \leq 34000 \\
 & x_4 + x_5 & \leq 28000 \\
 & x_i \geq 0 \quad \forall i \in I &
\end{array}
$$

The objective function maximizes the profit. Constraint 1 ensures that the macro lenses needed to sell the cameras do not surpass the stock level. Constraints 2 and 3 follow the same logic for prime and wide lenses. Constraint 4 and 5 each assures that the planned sales for cameras PI and PIII respectively are lower or equal to the demand of the products. Constraint 6 makes sure that the total demand of PI, PII and PIII combined do not surpass their aggregated demand. Constraint 7 ensures the same for cameras of type PIV and PV.

1.2 Optimal Solution

The optimal solution of the LP described can be found in table 1.1.

Table 1.1: Optimal solution for Linear Program

PI	**PII**	**PIII**	**PIV**	**PV**	**Profit**
16,000	14,800	2,000	0	28,000	1,404,000,000

1.3 General Sensitivity Analysis

A sensitivity analysis was performed using Excel to find the shadow prices and their sensitivity. The results can be found in table 1.2.
The shadow prices state how much one would be willing to pay for an additional resource in order to maximize the objective function. In this case the firm would be willing to pay 30,000 SEK for an additional wide lens.
In addition to that the sensitivity analysis calculates the increase and decrease as well. Those numbers describe how much the right hand side of the constraint can change till the shadow price changes. Note that for a non-binding constraint, which shadow price is always 0, the increase is ∞.

Table 1.2: Sensitivity Analysis Results

Constraint	**Value**	**Shadow Price**	**R.H. Side**	**Increase**	**Decrease**
Macro Lens	32,000	0	36,000	1E+30	4,000
Prime Lens	61,600	0	216,000	1E+30	154,400
Wide Lens	18,000	30,000	18,000	1,200	14,800
Demand Constraint 1	16,000	34,000	16,000	1,500	16,000
Demand Constraint 2	2,000	20,000	2,000	1,200	2,000
Demand Constraint 3	32,800	0	34,000	1E+30	1,200
Demand Constraint 4	28,000	10,000	28,000	154,400	28,000

1.4 Extended Sensitivity Analysis Demand PIII

This subsections further analyses the impact of a change in demand of product PIII. Therefore the the maximum demand for PIII was incremented with a step size of 600 pieces and the LP was executed for every step.
The profit increases by 20,000 SEK for every additional demand of PIII (shadow price) till reaching a demand of 3,200. Note that this is consistent with the results of the sensitivity analysis in the subsection above (value of 2,000 + increase of 1,200). From >3,200

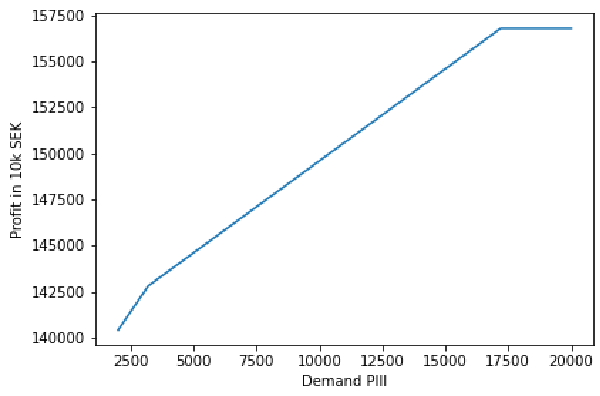


Figure 1.1: Impact of increasing demand of PIII on profit

till 17,200 the shadow price is 10,000 SEK. For a demand over 17,200 the shadow price is constant at 0 and the profit stagnates at 1,568,000,000 SEK.
The plot of the results is presented in figure 1.1. We can observe that the slope of the curve changes twice what corresponds to the number of times the shadow price changes.

1.5 Extended Sensitivity Analysis Price PI

It is not needed to implement the suggested simulation ure as the Excel Sensitivity Report gives us the numbers we need. For all variable cells Excel calculates the allowable increase and decrease for the objective coefficient (price in this case). Excel gives us an allowable decrease of $34,000$ and an allowable increase of 10^30. As the demand constraint for camera PI is binding, we can see straight away that the price can rise to ∞ and the optimal number of PI can not change due to that constraint.
As a result the value α equals 0.6 and β equals ∞ resulting in an allowable range for the price of $[0.6, \infty)$ without an impact on the value of PI sold.

2 Optimization - Integer Problem

2.1 Integer Problem

The following Integer Problem (IP) should be solved:

$$\begin{aligned}
\textbf{max} \quad & x_1 + 5x_2 \\
& 2x_1 - \ \ x_2 \leq 4 \\
\textbf{s.t.} \quad & -x_1 + \ \ x_2 \leq 1 \\
& x_1 + 4x_2 \leq 12 \\
& x_1, x_2 \in Z^+
\end{aligned}$$

Implemented and solved MATLAB delivers the results presented in table 2.1.

Table 2.1: Optimal solution for Integer Program

x_1^*	x_2^*	z^*
3	2	13

2.2 Relaxed Problem

Relaxing the IP to a LP ($x_1, x_2 \in R$) the solution changes according to table 2.2.

Table 2.2: Optimal solution for Integer Program

x_1^*	x_2^*	z^*
1.6	2.6	14.6

Rounding the x-values of the optimal solution of the LP gives four pairs of x-values. Only two of them are feasible and both pairs lead to nonoptimal solutions as the objective values of 11 and 12 are lower than the optimal objective value of the IP of 13. The results are summarized in table 2.3.

Table 2.3: Optimal solution for Integer Program

x_1^*	x_2^*	**feasible?**	z^*
1	2	yes	11
1	3	no, violating constraints 2 & 3	-
2	2	yes	12
2	3	no, violating constraint 3	-

2.3 Branch & Bound

Performing a optimization of the IP using the branch and bound algorithm, the following steps were executed:

1. Calculate relaxed problem (LP), x_1 and x_2 non integers, initial upper bound $= 14.6$
2. Lower bound obtained by rounding (see subsection above) $= 12$
3. Branch x_1: A: $x_1 \leq 1$, B: $x_2 \geq 2$
4. Solve LP with additional constraint $x_1 \leq 1$ (Branch A), x_1 are x_2 integers, terminate branch as objective value $<$ lower bound
5. Solve LP with additional constraint $x_1 \geq 2$ (Branch B), x_2 non integer
6. Branch branch B: C: $x_2 \leq 2$, $x_2 \geq 3$
7. Solve LP with additional constrains $x_1 \geq 2$ and $x_2 \leq 2$ (Branch C), x_1 are x_2 integers, new lower bound $= 13$
8. Solve LP with additional constrains $x_1 \geq 2$ and $x_2 \geq 3$ (Branch D, terminate branch problem infeasible
9. Only final nodes left, examining both branches (A and B) again, the maximum value of all final nodes is 13, set this to upper bound and terminate as this equals the lower bound

The solution obtained by the branch and bound algorithm equals the optimal solution stated in section 2.1. The steps are summarized in a branch and bound tree presented in 2.1.

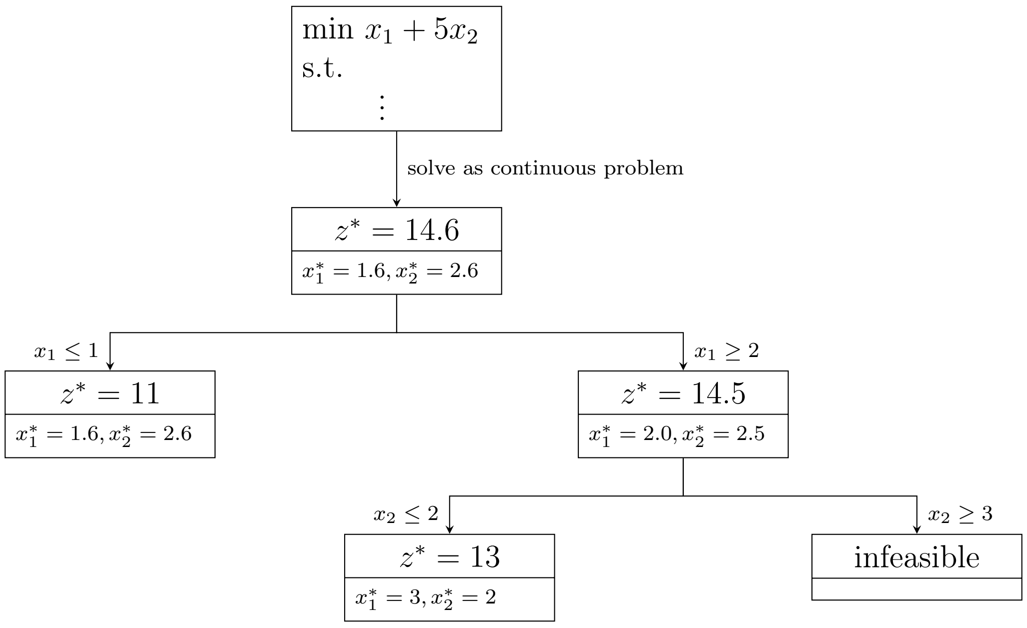


Figure 2.1: Branch and Bound Tree of IP

3 Simulation - Discrete Event Simulation

In this section the following queuing system is analyzed:

- There are two servers
- The service times are uniformly distributed between 0 and 2 seconds
- The time between the arrivals are exponentially distributed with mean 1.2 seconds

3.1 Implementation

3.1.1 Event Scheduling

To model the queuing system an event scheduling approach has been chosen as the system is fairly simple and only the two event arrival and departure exist. The rules for the events are described in procedures 1 and 2.

Procedure 1 RuleArrival

```
$N \leftarrow N + 1$
if $N <= 2$ then
    insertEvent(Departure, RndUni(0,2))
end if
insertEvent(Arrival, RndExp(1.2))
```

Procedure 2 RuleDeparture

```
$N \leftarrow N - 1$
**if** $N1 > 1$ **then**
    insertEvent(Departure, RndUni(0,2))
**end if**
```

3.1.2 Simulation Runs and Random Variables

In total 200 simulation runs were performed. The special feature lies in the fact that antithetic variables are used for the simulation runs. To avoid unwanted effects, the random variables for the arrivals, the service times and the measurements are kept in different vectors. For the first 100 runs the 'normal' random variables between 0 and 1 are used and for the last 100 runs the antithetic or 'flipped' variables are deployed.
Each simulation run keeps running till the standard deviation of the mean of customers in the system is less than 0.01 .

3.2 Results

In order to be able to analyze the results correctly, first of all the correlation of the two results vectors (one for the 'normal variables and one for the 'flipped' variables) is calculated. As expected, the correlation if the two vector is negative and amounts to -0.2069. The simulation results of the antithetic variables are visualized in figure 3.1.

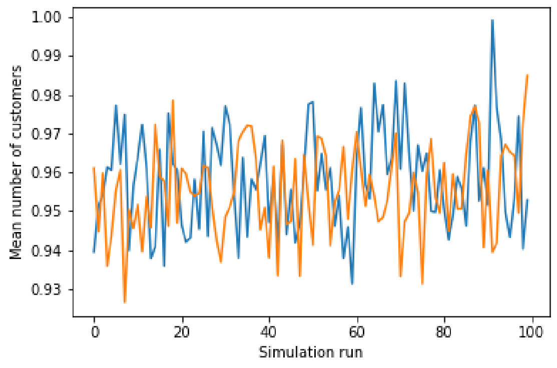


Figure 3.1: Simulation results for antithetic variables

To calculate the variance of the 200 simulation runs we use the following equation, where X_1 are the first 100 runs and X_2 the last 100 runs with the 'flipped' random numbers:

$$\begin{aligned} Var(\frac{X_1 + X_2}{2}) &= \frac{1}{4}[Var(X_1) + Var(X_2) + 2Cov(X_1, X_2)] \\ &= \frac{1}{4}[Var(X_1) + Var(X_2) + 2\sqrt{Var(X_1)}\sqrt{Var(X_2)}\rho_{12}] \end{aligned}$$

To obtain the variance of the mean customers in the system of all simulation runs, we use the variance calculated above and divide it by the number of observations (200 runs).

$$Var(mean) = \frac{1}{N}Var(X) = \frac{1}{N}Var(\frac{X_1 + X_2}{2})$$

Plugging everything together the estimated mean customers in the system are 0.956873 and the 95% confidence interval is [0.955788, 0.957958]. The results are summarizes in table 3.1. The distribution of the customers is the system is plotted in figure 3.2.

runs	**mean**	**sd**	**sd mean**	**t crit**	**ci mean**
1-100	0.958390	0.012993	0.001299	1.984217	[0.955812, 0.960968]
101-200	0.955355	0.011661	0.001166	1.984217	[0.953042, 0.957669]
total	0.956873	0.007780	0.000550	1.971957	[0.955788, 0.957958]

Table 3.1: Simulation results

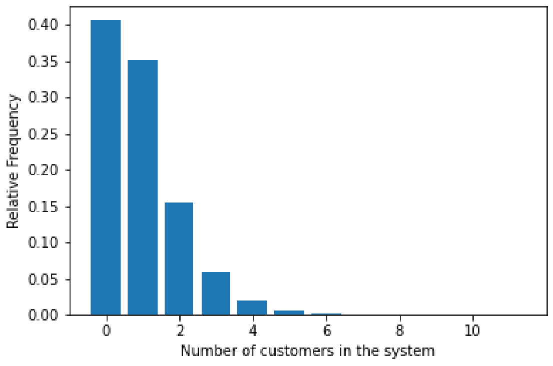


Figure 3.2: Relative frequency of customers in the system

3.3 Validation

We slightly adjust the assumptions to validate the model and its underlying rules. Instead of having a service time that is uniformly distributed between 0 and 2 seconds (mean of 1

seconds) we assume that the service times are exponentially distributed with a mean of 1 seconds. Running the simulation again and following the calculations of last subsection we estimate a mean of 1.008374 customers in the system. The 95% confidence interval of the mean is [1.007104, 1.009645].
Applying basic M/M/c (in this case we have a M/M/2 system) formulas, we can calculate that the true value for the mean customers in the system is 1.008403 (see calculations below). This comes pretty close to our estimated results and the true value lies inside the estimated confidence interval.
Thereby we can successfully validate our simulation. The change back to the uniform distribution is implemented without errors and the results is as expected close to the one of a exponential distribution.

Definitions

λ :	Arrival rate
μ :	Service rate
c :	Number of Servers
P_0 :	Probability of having zero vehicles in the system
L_q :	Expected queue length
L :	Expected average customers in the system

Calculations

$$\rho = \frac{\lambda}{\mu} = \frac{\frac{1}{1.2}}{1}$$

$$P_0 = \left[\sum_{n=0}^{c-1} \frac{\rho}{n!} + \frac{\rho^c}{c!(1-\rho/c)} \right]^{-1} = 0.411765$$

$$L_q = P_0 \frac{\rho^{c+1}}{cc!} \frac{1}{(1-\rho/c)^2} = 0.175070$$

$$L = L_q + \rho = 1.008403$$

4 Simulation - Random Number Generator

4.1 Geometric distribution

We have a random number generator that gives uniformly distributed numbers between 0 and 1. We now want to form random numbers that follow a geometric distribution, so that you get an integer N where

$$P(N = k) = \alpha^k(1 - \alpha) \text{ for } k = 0, 1, 2, \ldots$$

. To do so we use the discrete inverse-transform method presented below.

$$U \sim unif(0, 1)$$

$$X = k, \text{if } \sum_{i=0}^{k-1} p(i) < U \leq \sum_{i=0}^{k} p(i), k \geq 1$$

Applying the method to the geometric distribution gives us

$$X = k, \text{if } \sum_{i=0}^{k-1} \alpha^i(1 - \alpha) < U \leq \sum_{i=0}^{k} \alpha^i(1 - \alpha), k \geq 1$$

.

For an exemplary α of 0.5 and the first 9 integers the method gives us the results presented in table 4.1.

Table 4.1: Example for $\alpha = 0.5$

k	$P(X = k)$	$P(X \leq k)$
0	0.5000	0.5000
1	0.2500	0.7500
2	0.1250	0.8750
3	0.0625	0.9375
4	0.0313	0.9688
5	0.0156	0.9844
6	0.0078	0.9922
7	0.0039	0.9961
8	0.0020	0.9980

4.2 Density function

We have a random number generator that gives uniformly distributed numbers between 0 and 1. We now want to form random numbers that are distributed according to the following probability density function (PDF):

$$f(t) = \begin{cases} 0.5t & \text{if } 0 \leq t \leq 1 \\ 0.5 & \text{if } 1 \leq t \leq 2 \\ 1.5 - 0.5t & \text{if } 2 \leq t \leq 3 \\ 0 & \text{otherwise} \end{cases}$$

. The PDF is plotted in figure 4.1.

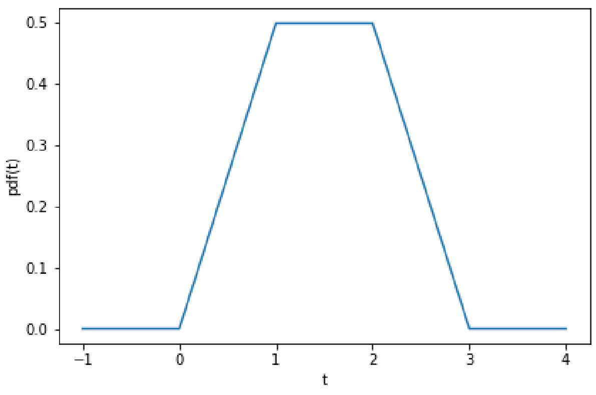


Figure 4.1: Plot of the Probability Density Function

As the PDF is continuous we can use the inverse cumulative distribution function (CDF) to generate the desired variables. The idea behind is that the CDF is standardized to only have values between 0 and 1. Using the inverse we put in numbers between 0 and 1 and get the desired values. Therefore we need to derive the CDF from the PDF and then inverse the function.

Deriving the CDF for the piecewise defined function

for $t < 0$:

$$F(t) = \int_{-\infty}^{t} f(x)dx = \int_{-\infty}^{t} 0dx = 0$$

for $0 \leq t \leq 0$:

$$F(t) = \int_0^t f(x)dx = \int_0^t 0.5xdx = 0.25x^2\Big|_0^t = 0.25t^2$$

for $1 \leq t \leq 2$:

$$F(t) = \int_0^1 0.5xdx + \int_1^t 0.5dx = 0.25 + (0.5x\Big|_1^t) = 0.5t - 0.25$$

for $2 \leq t \leq 3$:

$$F(t) = \int_0^1 0.5xdx + \int_1^2 0.5dx + \int_2^t 1.5 - 0.5xdx = 0.75 + (1.5x - 0.25x^2\Big|_2^t) = -0.25t^2 + 1.5t - 1.25$$

for $t \geq 3$:

$$F(t) = 1$$

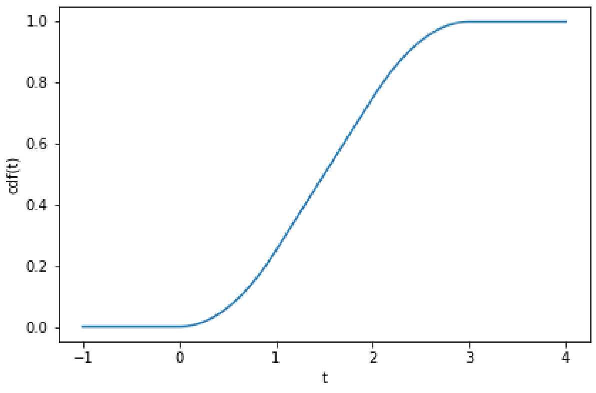


Figure 4.2: Plot of the Cumulative Density Function

Forming the inverse CDF

for $x < 0$:

not defined

for $0 \leq x \leq 0.25$:

$$F^{-1}(x) = 2\sqrt{x}$$

for $0.25 \leq x \leq 0.75$:

$$F^{-1}(x) = \frac{4t+1}{2}$$

for $0.75 \leq x \leq 1$:

$$F^{-1}(x) = 3 - 2\sqrt{-x+1}$$

for $x \geq 1$:

not defined

When replacing the x by $U \sim unif(0,1)$, we obtain the desired random numbers.

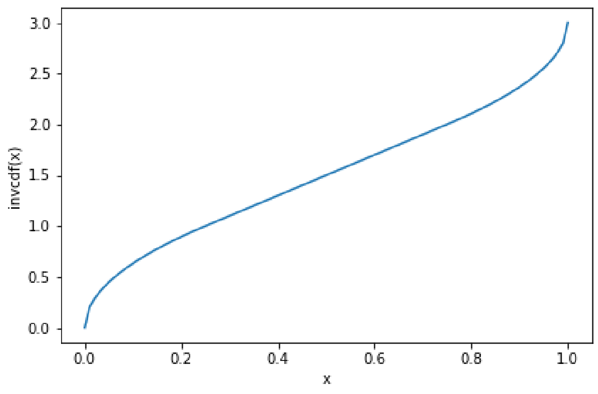


Figure 4.3: Plot of the Inverse Cumulative Density Function

5 Metaheuristics - Simulated Annealing

5.1 Problem Formulation and Problem Characteristics

Definitions

L :	Number of periods
G :	Maximum goods that can be produced in one period $t \in 1, \ldots, L$
D :	Demand vector of demand in periods $t \in 1, \ldots, L$
C_F :	Vector of fixed costs in periods $t \in 1, \ldots, L$
C_P :	Vector of production costs proportional to the amount of goods produced in periods $t \in 1, \ldots, L$
C_S :	Vector of cost of storage proportional to the amount of goods stored at the end of periods $t \in 1, \ldots, L$
X :	Vector of units produced in periods $t \in 1, \ldots, L$
Y :	Binary vector whether produce or not in periods $t \in 1, \ldots, L$
Z :	Vector of number of goods stored in periods $t \in 1, \ldots, L$

Problem Formulation

$$
\begin{aligned}
\min \quad & \sum_{t=0}^{L} C_P(t)X(t) + C_F(t)Y(t) + C_S(t)Z(t) & \\
\text{s.t.} \quad & Z(t-1) + X(t) = D(t) + Z(t) & \forall t \in 1, \ldots, L \\
& X(t) \leq GY(t) & \forall t \in 1, \ldots, L \\
& Z(t) = 0 & \text{for } t = 0 \\
& X(t), Z(t) \in N^+, Y(t) \in 0, 1 & \forall t \in 1, \ldots, L
\end{aligned}
$$

Problem Description

The problem is to minimize the total cost of the production to satisfy the demand for a product. The costs consist of variable production costs per piece C_P, fixed production costs C_F(if one produces in the period) and storage costs C_S. The mentioned costs can vary over time periods. The demand of each period must be met by the sum of production in the period and inventory from the previous period. In addition, a maximum of G pieces can be produced in a period.

5.2 Implementation

For this task we want to use an simulated annealing (SA) algorithm to find a good solution to the problem. The general SA algorithm works in the following way.

Algorithm 1 Simulated Annealing

```
$T \leftarrow T_{max}$
$best \leftarrow$ INIT()
while $T > T_{min}$ do
    $next \leftarrow$ NEIGHBOUR($T, best$)
    $\Delta \leftarrow$ VALUE($next$) $-$ VALUE($best$)
    if $\Delta E < 0$ then
        $best \leftarrow next$
    else if RANDOM() $<$ ACCEPT($T, \Delta E$) then
        $best \leftarrow next$
    end if
    $T \leftarrow$ COOLING($T, best$)
end whilereturn $best$
```

adapted from `https://github.com/antonpetkoff/optimization-methods/blob/master/latex/simulated-annealing.tex`

To implemented the SA algorithm we need to 'customize' the initialization, the temperature and acceptance and the neighbour search. These decisions will be presented in the next subsections. Note that all parameters have been tuned for a general problem with $L \leq 12$ and not for the specific sample input that is given for validation purposes.

5.2.1 Initialization

To initialize the algorithm we need a feasible start-solution. We generate such a solution by simply setting the production in a period equal to the demand in that period.

5.2.2 Temperature and Acceptance

The complexity of the problem increases with more time periods. This is why I suggest a start temperature depends on the number of time periods L as more local optima exist with more time periods.

Start temperature: $T_{max} = exp(L/12)$

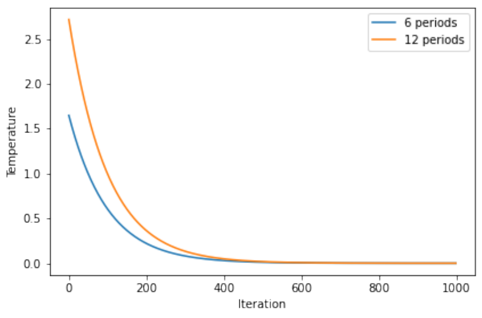
For the cooling I suggest a geometric chain. The temperature as a function of the iteration is shown in figure 5.2a. As the start temperature depends on the number of periods, the figure shows the plots for 6 and 12 periods.

$$\text{Temperature i-th iteration: } T_i = T_{max} \cdot \alpha^i \text{ ,with } \alpha = 0.99$$

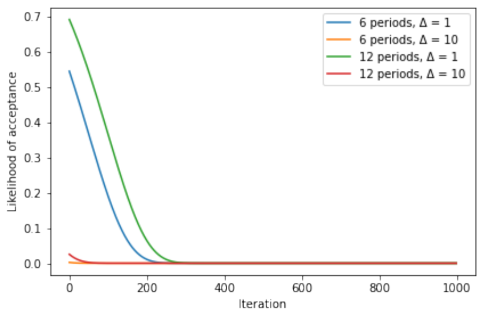
Another important decision to make is which solutions to accept. In this case I suggest the following probabilities to accept new solutions:

$$P(T, \Delta) = \begin{cases} 1 & \text{, if } \Delta < 0 \\ exp(\frac{-\Delta}{T}) & \text{, else} \end{cases}$$

As a results all better solutions are accepted. Inferior solutions will be accepted depending on the temperature of system and their difference to the current solution ('how much worse is the solution?'). Figure 5.2b shows the likelihood to accept solutions that have a objective value that is 1 or 10 units worse for a system with 6 or 12 periods. Remember that the start temperature for those systems is different and thereby also the temperature after several iterations leading to different likelihoods.



(a) Temperature



(b) Likelihood of accepting an inferior solution

Figure 5.2: Temperature and Acceptance

5.2.3 Neighbour Generation

First of all let us point out some problem characteristics that are useful to find neighbours to the current solution of the problem:

- Vectors Y and Z can always be derived from a feasible X by setting $Y(t) = 1$ for periods with $X(t) > 0$ and calculating $Z(t) = X(t) + Z(t-1) - D(t)$. Therefore only X has to be varied to find neighbours and Y and Z are calculated based on X.

- The total demand has to be equal to the total production for all feasible solutions. As a consequence the sum over all elements of X is constant for all feasible solutions of the problem. Thereby all the produced units removed from one period have to be added to other periods when generating neighbours.
- As long as the production capacity is not surpassed the production of a unit can always be moved forward as units that are produced earlier will not miss in later periods. This comes at the cost of storing the unit.
- Postponing the production of a unit is not as easy as forwarding it. On one hand the maximum capacity still shall not be surpassed. On the other hand it has to be made sure that the available units in one periods (units produced this period and stored last period) satisfy the demand in that period. Imagine a two periods problem with a demand of 5 in each period and a maximum production capacity of 10 units. We would initialize the solution by setting the production in each period to 5. Forwarding the 5 units from the second to the first period would be possible as the 5 remaining units are stored and can be consumed in the second period. However this does not work the other way round as producing 10 units in the last period does not satisfy the demand of 5 in the first period.

Tanking these characteristics into account I suggest two methods of finding neighbours for the SA algorithm.
The first method described in algorithm 2 picks a random production period, picks a number of units to remove from that period considering the feasibility of the problem and then either postpones, forwards or postpones and forwards units to neighbour periods.
The second method described in algorithm 3 picks two production periods and then moves the production of a random number of units from one period to the other while ensuring the feasibility of the problem. Note that those can be two periods can be any periods of the problem and do not have to be close to each other as for algorithm 2.
The first algorithm is running from the beginning on. The second algorithm is launched in later iterations of the SA algorithm. The idea behind that is that at the very beginning the algorithm finds those periods in which it is cheap to produce and later on then finds the optimal allocation of the production among those. In this particular case algorithm 3 is used in 50% of the cases after reaching a system temperature of 30% of the starting temperature.

Algorithm 2 Neighbour Generation 1

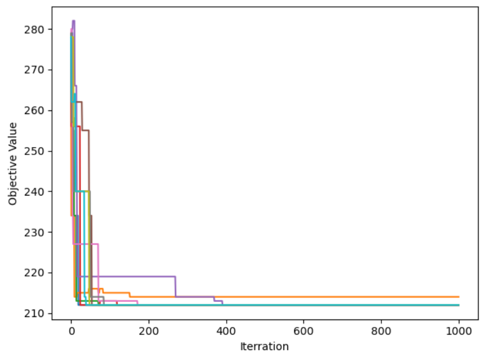
```
Pick random period t with $x_t > 0$
Calculate $max\_postpone$ for t
Calculate $max\_removal$ for t: $\min(x_t, max\_postpone_t + max\_forward_t)$
Pick random number between(1, $max\_removal_t$) and remove it from period t
Pick mode: postpone (40%), forward (30%) or mix (30%)?
switch mode do
    case postpone
        Postpone removed amount to later periods (considering g) till everything postponed or last period reached, if last period reached and amount left-> forward the remainder
    case postpone
        Forward removed amount to earlier periods (considering g) till everything forwarded or first period reached, if first period reached and amount left -> postpone the remainder
    case mix
        Pick random fraction of amount to remove and postpone it -> forward the remainder
Calculate y and z from new x
```

Algorithm 3 Neighbour Generation 2

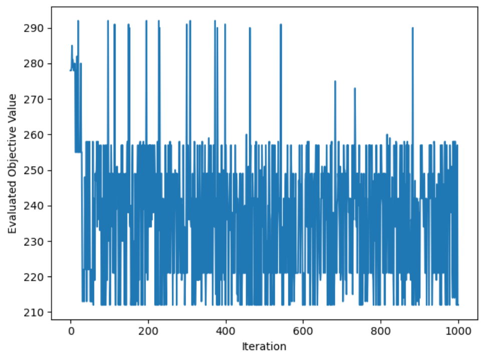
```
Pick random period t with $x_t > 0$
Pick second random period t2 with $x_t2 > 0$ and $t! = t2$
if t > t2 then
    Calculate max_forward
    if max_forward > 0 then
        Pick random number between(1, max_forward), remove it from period t and add it to t2
    end if
else if t<t2 then
    Calculate max_postpone
    if if max_postpone > 0 then
        Pick random number between(1, max_postpone), remove it from period t and add it to t2
    end if
end if
Calculate y and z from new x
```

5.3 Behaviour and Solution

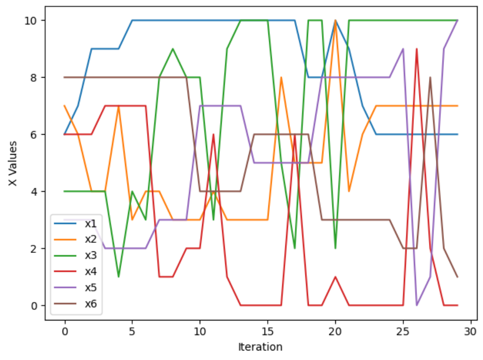
This subsection refers to the sample input given in the assignment of the take home exam. The algorithm ran 100 times with different seeds.
In figure 5.3 we can get an understanding of the behaviour of the algorithm in this particular case. Figure 5.3a shows the current (accepted) objective value as a function of the iteration for 30 random seeds. The next figure 5.3b shows the checked objective value for all iterations for one exemplary seed. The last subfigure 5.3c plots the first 30 checked x-values for one random seed to see how the decision variables vary over time.



(a) Best solution per iteration



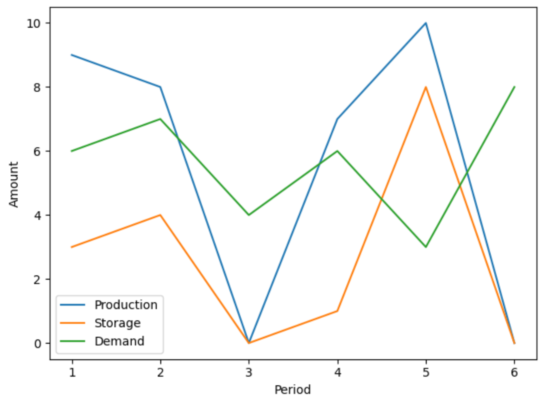
(b) Objective value of checked neighbours



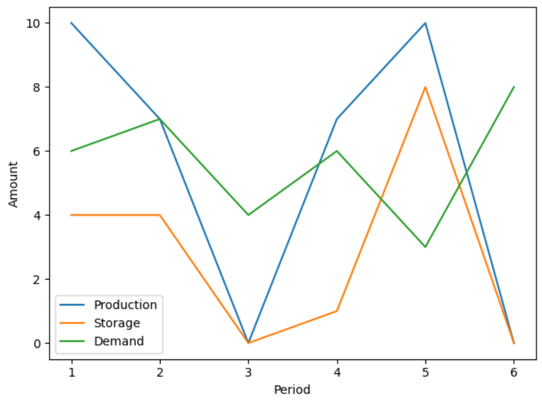
(c) Checked X for the first iterations

Figure 5.3: Behaviour of the implemented SA algorithm

For the 100 runs we get an average objective value of 212.58 with a best objective value of 212 whereas the worst objective value is 218. Gurobi gives us an optimal objective value of 212. As a results we can see that the algorithm finds the best objective value for this problem. Note that the solution is not necessarily the same as this problem has several optimal solutions with the same objective value. Plots 5.4 show the optimal solution obtained by Gurobi and one optimal solution from the SA algorithm.



(a) (One) Solution obtained by SA



(b) Solution obtained from Gurobi

Figure 5.4: Solutions for sample data of SA and Gurobi

A Appendix

A Task 1 & 2

```
clc
clear all
 *Question 1: Product Mix Problem*
% *Basic Problem*

c = [-4;
     -3;
     -2;
     -2;
     -1];
A = [2 0 0 0 0;
     0 2 2 2 1;
     0.2 1 0 0.5 0;
     1 0 0 0 0;
     0 0 1 0 0;
     1 1 1 0 0;
     0 0 0 1 1];
b = [36000;
     216000;
     18000;
     16000;
     2000;
     34000;
     28000];
lb = [0;
      0;
      0;
      0;
      0];

options = optimoptions('linprog', 'Algorithm', 'dual-simplex',
   'Display', 'off');
[x,fval,exitflag,output,lambda] = linprog(c', A, b, [], [], lb,
   [], [], options)
%%
clc
clear all
%% Question 2: Integer Problem
% Implemetation of Integer Problem

c = [-1;
     -5];
A = [2 -1;
     -1 1;
     1 4];
b = [4;
     1;
     12];
lb = [0;
       0];
options = optimoptions('intlinprog', 'Display', 'off');
intcon = 1:2
[x, fval, exitflag, output] = intlinprog(c', intcon, A, b, [],
   [], lb, [], options)
% Relaxed Problem
```

```
options = optimoptions('linprog', 'Algorithm', 'dual-simplex',
   'Display', 'off');
[x,fval,exitflag,output,lambda] = linprog(c', A, b, [], [], lb,
   [], [], options)
%%
% The variable x_1 can be rounded to either 1 or 2 (floor or
   ceil) and x_2 to
% 2 or 3.
%
% Combining all the differenct variable values, four pairs can
   be identified
% and have to be checked for feasibility:
%
% x_1 = 1, x_2 = 2  --> feasible, objective value: 11
%
% x_1 = 1, x_2 = 3 --> infeasible, violating constraints 2 and 3
%
% x_2 = 2, x_2 = 2 --> feasible, objective value 12
%
% x_2 = 2, x_2 = 3 --> infeasible, violating contraint 3
%
% ...
% Branch & Bound
% 1) Initialization
% Upper Bound: Relaxed Problem, z = 14.6, x_1*= 1.6, x_2*= 2.6
%
% Lower Bound: Rounding down both variables, z= 11, x_1*= 1,
   x_2*= 2
% *2) Branching, x_1<=1 (Subproblem A),  x_1 >=2 (Subproblem
   B),*
% Subproblem A

A = [2 -1;
     -1 1;
     1 4;
     1 0 ];
```

```
b = [4;
     1;
     12;
     1];
[x,fval,exitflag,output,lambda] = linprog(c', A, b, [], [], lb,
   [], [], options)

% Subproblem B

A = [2 -1;
     -1 1;
     1 4;
     -1 0 ];
b = [4;
     1;
     12;
     -2];
[x,fval,exitflag,output,lambda] = linprog(c', A, b, [], [], lb,
   [], [], options)
% 3) Branching x_2<= 2 (Subproblem C), Branching x_2>= 3
   (Subproblem D)
% Subproblem C

A = [2 -1;
     -1 1;
     1 4;
     -1 0;
     0 1];
b = [4;
     1;
     12;
     -2;
     2];
[x,fval,exitflag,output,lambda] = linprog(c', A, b, [], [], lb,
   [], [], options)
```

```
% Subproblem D

A = [2 -1;
     -1 1;
     1 4;
     -1 0;
     0 -1];
b = [4;
     1;
     12;
     -2;
     -3];
[x,fval,exitflag,output,lambda] = linprog(c', A, b, [], [], lb,
   [], [], options)
```

B Task 3

```
class Event{
        public double eventTime;
        public int eventType;
        public Event next;
}


public class EventListClass {

        private Event list, last;

        EventListClass(){
                list = new Event();
        last = new Event();
        list.next = last;
        }


        public void InsertEvent(int type, double TimeOfEvent){
```

```
        Event dummy, predummy;
        Event newEvent = new Event();
        newEvent.eventType = type;
        newEvent.eventTime = TimeOfEvent;
        predummy = list;
        dummy = list.next;
        while ((dummy.eventTime < newEvent.eventTime) & (dummy
           != last)){
                predummy = dummy;
                dummy = dummy.next;
        }
        predummy.next = newEvent;
        newEvent.next = dummy;
 }

        public Event fetchEvent(){
                Event dummy;
                dummy = list.next;
                list.next = dummy.next;
                dummy.next = null;
                return dummy;
        }
}
import java.util.*;
import java.io.*;

public class MainSimulation extends GlobalSimulation{
        static FileWriter writerMeasurements;

    public static void main(String[] args) throws IOException {
                ArrayList meanCustomerList = new
                   ArrayList<Double>();
                writerMeasurements = new
                   FileWriter("Measurements.csv");

                for(int i=0; i<100; i++){
                    meanCustomerList.add(runSimulation(i));
                }
                writerMeasurements.close();
                writeToCsv(meanCustomerList,
                   "MeanCustomersInQueue0.csv");
      }

          public static double runSimulation(int i) throws
             IOException{
                time = 0;
                seed = i;
                sdMean = 100.0;
                currentNumberOfCustomers = new
                   ArrayList<Integer>();
                eventList = new EventListClass();
                Event actEvent;
          State actState = new State();
          insertEvent(ARRIVAL, 0);
          insertEvent(MEASURE, 5);


          while (sdMean > 0.01){
                actEvent = eventList.fetchEvent();
                time = actEvent.eventTime;
                actState.treatEvent(actEvent);
          }

          double meanCustomers =
             1.0*actState.accumulated/actState.noMeasurements;

                writeMeasurements(currentNumberOfCustomers);

                System.out.println("Mean customers: " +
                   meanCustomers);
                System.out.println("Standard Dev: " + sdMean);
```

```
            System.out.println("Measurements: " +
               actState.noMeasurements);

            return meanCustomers;
        }

        public static void writeToCsv(ArrayList list, String
           filename) throws IOException{
            FileWriter writer = new FileWriter(filename);
            for (Object dd:list)
                    writer.write(dd +
                       System.lineSeparator());
            writer.close();
        }

        public static void writeMeasurements(ArrayList list)
           throws IOException{
            for (Object dd:list)
                    writerMeasurements.write(dd +
                       System.lineSeparator());
        }


}

import java.util.*;

public class GlobalSimulation{

        public static final int ARRIVAL = 1, DEPARTURE = 2,
           MEASURE = 3;
        public static double time = 0;
        public static double sdMean;
        public static int seed;
        public static ArrayList<Integer>
           currentNumberOfCustomers = new ArrayList<Integer>();
        public static EventListClass eventList = new
           EventListClass();
        public static void insertEvent(int type, double
           TimeOfEvent){
                eventList.InsertEvent(type, TimeOfEvent);
        }
}
import java.util.*;
import java.io.*;

class State extends GlobalSimulation{

        public int numberInQueue = 0, accumulated = 0,
           noMeasurements = 0;

        Random rndArrival = new Random(seed);
        Random rndServer = new Random(seed+1);
        Random rndMeasure = new Random (seed+2);

        public void treatEvent(Event x){
                switch (x.eventType){
                        case ARRIVAL:
                                arrival();
                                break;
                        case DEPARTURE:
                                departure();
                                break;
                        case MEASURE:
                                measure();
                                break;
                }
        }

        private void arrival(){
                numberInQueue++;
                if (numberInQueue <= 2)
                        //insertEvent(DEPARTURE, time + 2.0 -
                           (2.0*rndServer.nextDouble()));
                        insertEvent(DEPARTURE, time +
                           (2.0*rndServer.nextDouble()));
                insertEvent(ARRIVAL, time + expRndArrival(1.2));
        }

        private void departure(){
                numberInQueue--;
                if (numberInQueue > 1)
                        //insertEvent(DEPARTURE, time + 2.0 -
                           (2*rndServer.nextDouble()));
                        insertEvent(DEPARTURE, time +
                           (2*rndServer.nextDouble()));
        }

        private void measure(){
                accumulated = accumulated + numberInQueue;
                noMeasurements++;
                currentNumberOfCustomers.add(numberInQueue);
                if (currentNumberOfCustomers.size()>10)
                        sdMean =
                           1.0*sd()/Math.sqrt(noMeasurements);
                        //System.out.println(sdMean);
                insertEvent(MEASURE, time + expRndMeasure(5));
        }

        public double expRndArrival(double expectedValue) {
                //return (Math.log(1.0 -
                   rndArrival.nextDouble())/(-1.0/expectedValue));
                return
                   (Math.log(rndArrival.nextDouble())/(-1.0/expectedValue)
        }

        public double expRndMeasure(double expectedValue) {
```

```
            //return (Math.log(1.0 -
               rndMeasure.nextDouble())/(-1.0/expectedValue));
            return
               (Math.log(rndMeasure.nextDouble())/(-1.0/expectedValue)
        }

        public static double sd (){
            Double mean = currentNumberOfCustomers
                                                        .stream()
                                                        .mapToInt(a
                                                           -> a)
                                                        .average()
                                                        .orElse(-1);
            double temp = 0;
            for (int i = 0; i <
               currentNumberOfCustomers.size(); i++)
            {
                    int val =
                       currentNumberOfCustomers.get(i);
                    double squrDiffToMean = Math.pow(val -
                       mean, 2);
                    temp += squrDiffToMean;
            }


            double meanOfDiffs = (double) temp / (double)
               (currentNumberOfCustomers.size());


            return Math.sqrt(meanOfDiffs);
        }

}
```

C Task 5

Skeleton code from: https://machinelearningmastery.com/simulated-annealing-from-scratch-in-python/

```
import numpy as np
from numpy import asarray
from numpy import exp
from numpy.random import rand
from numpy.random import randint
from numpy.random import seed
from matplotlib import pyplot
import pandas as pd
from exact_solution import get_exact_solution


# objective function
def objective(x, y, z, cp, cf, cs):
    return np.sum(cp * x) + np.sum(cf * y) + np.sum(cs * z)


def postpone(x, amount, time_period, g, less):
    x_new = x.copy()
    if amount > 0:
        x_new[time_period - 1] = x[time_period - 1] - amount
        while amount > 0:
            next_period = min(amount, g - x[time_period])
            x_new[time_period] = x[time_period] + next_period
            amount = amount - next_period
            time_period -= 1
    return x_new


def forward(x, amount, time_period, g,less):
    x_new = x.copy()
    if amount > 0:
        x_new[time_period - 1] = x[time_period - 1] - amount
        while amount > 0:
            next_period = min(amount, g - x[time_period - 2])
            x_new[time_period - 2] = x[time_period - 2] +
               next_period
            amount = amount - next_period
            time_period -=1
    return x_new


def feasible_solution(x, demand):
    y = np.zeros(len(x))
    z = np.zeros(len(x))
    for i in range(0, len(x)):
        if x[i] > 0:
            y[i] = 1
        else:
            pass
        if i == 0:
            z[i] = x[i] - demand[i]
        else:
            z[i] = z[i - 1] + x[i] - demand[i]
    return y, z


def neighbor(x, y, z, d, g, temp, t):
    x_new = x.copy()
    current_x_t = 0
    while current_x_t < 1:
        rand_period = randint(1, len(x) + 1)
        current_x_t = x[rand_period - 1]
    random_case = rand()
    if random_case < 0.5 and t < 0.3* temp:
    #if random_case < 0.5:
        current_x_t2 = 0
        second_period = 0
        while current_x_t2 < 1 and rand_period != second_period:
            second_period = randint(1, len(x) + 1)
            current_x_t2 = x[second_period - 1]
        if rand_period > second_period: # forward production
            max_forward = min(g-x[second_period-1],
               x[rand_period-1])
            if max_forward > 0:
                x_forward = randint(1, max_forward+1)
                x_new[rand_period-1] = x[rand_period-1] -
                   x_forward
                x_new[second_period-1] = x[second_period-1] +
                   x_forward
        else: # postpone production
            max_postpone_production = min(g-x[second_period-1],
               x[rand_period-1])
            max_postpone_demand = x[rand_period-1]
            for t in range(rand_period-1, second_period):
                demand_till_period = d[0:t].sum()
                production_till_period = x[0:t].sum()
                max_postpone_demand =
                   min(production_till_period -
                   demand_till_period, max_postpone_demand)
            max_postpone = min(max_postpone_demand,
               max_postpone_production)
            if max_postpone > 0:
                x_postpone = randint(1, max_postpone + 1)
                x_new[rand_period-1] = x[rand_period-1] -
                   x_postpone
                x_new[second_period-1] = x[second_period-1] +
                   x_postpone
    if rand_period == len(x):
        max_postpone = 0
    else:
        demand_till_period = d[0:rand_period].sum()
        production_till_period = x[0:rand_period].sum()
        max_postpone_prod_demand = production_till_period -
           demand_till_period
        max_postpone_spare_capacity = (len(x) - rand_period) *
           g - x[rand_period: len(x)].sum()
        max_postpone = min(max_postpone_prod_demand,
           max_postpone_spare_capacity)
    if rand_period == 1:
        max_forward = 0
    else:
        max_forward = (rand_period - 1) * g - x[0:rand_period -
           1].sum()
    max_remove = min(current_x_t, max_forward + max_postpone)
    if max_remove > 0:
        remove_x_t = randint(1, max_remove + 1)
    else:
        remove_x_t = 0
    rand_case = rand()
    if rand_case < 0.4:
        # postpone
        postpone_x = min(remove_x_t, max_postpone)
        x_new = postpone(x, postpone_x, rand_period, g, True)
        rest = remove_x_t - postpone_x
        if rest > 0:
            x_new = forward(x_new, rest, rand_period, g, True)
    elif rand_case < 0.7:
        # forward
        forward_x = min(remove_x_t, max_forward)
        x_new = forward(x, forward_x, rand_period, g, True)
        rest = remove_x_t - forward_x
        if rest > 0:
            x_new = postpone(x_new, rest, rand_period, g, True)
    else:
        postpone_x = randint(0, min(max_postpone, remove_x_t) +
           1)
        forward_x = randint(0, min(max_forward, remove_x_t -
           postpone_x) + 1)
        x_new = postpone(x, postpone_x, rand_period, g, True)
        x_new = forward(x_new, forward_x, rand_period, g, True)
    y_new, z_new = feasible_solution(x_new, d)
    return x_new, y_new, z_new



# simulated annealing algorithm
def simulated_annealing(cp, cf, cs, d, g, n_iterations, temp):
    # generate initial point
    x_best = d.copy()
    y_best = np.ones(len(d))
    z_best = np.zeros(len(d))

    best_scores = list()
    scores = list()
    checked_scores = list()
    checked_x = list()
    improvement_iterations = list()

    # evaluate the initial point
    best_eval = objective(x_best, y_best, z_best, cp, cf, cs)
    best_scores.append(best_eval)
    scores.append(best_eval)
    checked_scores.append(best_eval)
    # current working solution
    x_curr, y_curr, z_curr, curr_eval = x_best, y_best, z_best,
       best_eval
    checked_x.append(x_curr)
    # run the algorithm
    for i in range(n_iterations):
        # calculate temperature for current epoch
        t = temp * (0.99**i)
        # take a step
        x_cand, y_cand, z_cand = neighbor(x_curr, y_curr,
           z_curr, d, g, temp, t)
        checked_x.append(x_cand)
        # evaluate candidate point
        candidate_eval = objective(x_cand, y_cand, z_cand, cp,
            cf, cs)
        checked_scores.append(candidate_eval)
        # check for new best solution
        if candidate_eval < best_eval:
            # store new best point
            x_best, y_best, z_best, best_eval = x_cand, y_cand,
                z_cand, candidate_eval
            # keep track of scores
            best_scores.append(best_eval)
            improvement_iterations.append(i)
        # difference between candidate and current point
            evaluation
        diff = (candidate_eval - curr_eval)

        # calculate metropolis acceptance criterion
        metropolis = exp(-diff / t)
        # check if we should keep the new point
        if diff < 0 or rand() < metropolis:
            # store the new current point
            x_curr, y_curr, z_curr, curr_eval = x_cand, y_cand,
                z_cand, candidate_eval
        scores.append(curr_eval)
    return [x_best, y_best, z_best, best_eval, best_scores,
        scores, checked_scores, checked_x,
        improvement_iterations]


# seed the pseudorandom number generator
seed(1)

# define the total iterations
n_iterations = 1000

# # cost of production in periods t
cp = asarray([3, 4, 3, 4, 4, 5])
```

```
cf = asarray([12, 15, 30, 23, 19, 45])
cs = asarray([1, 1, 1, 1, 1, 1])
# demand per period t
d = asarray([6, 7, 4, 6, 3, 8])
g = 10

# random problem generator

#seed(0)
# g = randint(10, 30)
# periods = randint(4, 13)
# d = randint(5, g * 0.8, size=periods)
# cp = randint(3, 6, size=periods)
# cf = randint(10, 50, size=periods)
# cs = randint(1, 3, size=periods)
# print(f'g: {g}')
# print(f'd: {d}')
# print(f'cp: {cp}')
# print(f'cf: {cf}')
# print(f'cs: {cs}')

# initial temperature
temp = exp(len(d)/12)

# exact solution
x_opt, z_opt, obj_opt = get_exact_solution(cp, cf, cs, d, g)
print(f'x opt: {np.round(x_opt,1)}')
print(f'obj val: {obj_opt}')

# plot of solution
pyplot.figure()
pyplot.plot(range(1,7), x_opt, label='Production')
pyplot.plot(range(1,7), z_opt, label= 'Storage')
pyplot.plot(range(1,7), d, label = 'Demand')
pyplot.ylabel('Amount')
pyplot.xlabel('Period')
pyplot.legend()
pyplot.tight_layout()
pyplot.savefig('PlotSolutionGurobi.png')

# perform the simulated annealing search
best_score_per_seed = list()
iterrations_for_best_per_seed = list()
scores_list = list()

for i in range(100):
    seed(i)
    best_x, best_y, best_z, score, best_scores, scores,
       checked_scores, checked_x, improvement_iterations =
       simulated_annealing(cp, cf, cs, d, g, n_iterations, temp)
    best_score_per_seed.append(score)
    iterrations_for_best_per_seed.append(improvement_iterations[-1])
    scores_list.append(scores)
    print('Done!')
    print(f'x:{best_x}, y: {best_y}, z:{best_z}')
    print(f'score: {score}')

print(f'Best average: {np.mean(best_score_per_seed)}')
print(f'Max score: {max(best_score_per_seed)}')
print(f'Iterations average:
   {np.mean(iterrations_for_best_per_seed)}')
df = pd.DataFrame({'Score': best_score_per_seed,
                 'Iterrations':iterrations_for_best_per_seed})
df.to_csv('Scores.csv')


# print('f(%s) = %f' % (best_x, best_y, best_z, score))

# line plot of scores for 10 iterrations
pyplot.figure()
for l in scores_list[0:10]:
    pyplot.plot(range(len(l)), l)
pyplot.xlabel('Iterration')
pyplot.ylabel('Objective Value')
pyplot.tight_layout()
pyplot.savefig('BestObjValesPlot.png')

# line plot of best scores
pyplot.figure()
pyplot.plot(range(len(best_scores)), best_scores)
pyplot.xlabel('Improvement Number')
pyplot.ylabel('Objective Value')
pyplot.tight_layout()
pyplot.savefig('BestObjValPlot.png')

# line plot of current scores
pyplot.figure()
pyplot.plot(range(len(scores)), scores)
pyplot.xlabel('Iteration')
pyplot.ylabel('Objective Value')
pyplot.tight_layout()
pyplot.savefig('ObjValPlot.png')

# line plot of checked scores
pyplot.figure()
pyplot.plot(range(len(checked_scores)), checked_scores)
pyplot.xlabel('Iteration')
pyplot.ylabel('Evaluated Objective Value')
pyplot.tight_layout()
pyplot.savefig('CheckedObjValPlot.png')

# line plot of checked x
pyplot.figure()
pyplot.plot(range(30), checked_x[:30], label =
   ['x1','x2','x3','x4','x5','x6'])
pyplot.xlabel('Iteration')
pyplot.ylabel('X Values')
pyplot.legend()
pyplot.tight_layout()
pyplot.savefig('CheckedXPlot.png')

# plot of solution
pyplot.figure()
pyplot.plot(range(1,7), best_x, label='Production')
pyplot.plot(range(1,7), best_z, label= 'Storage')
pyplot.plot(range(1,7), d, label = 'Demand')
pyplot.ylabel('Amount')
pyplot.xlabel('Period')
pyplot.legend()
pyplot.tight_layout()
pyplot.savefig('PlotSolutionSA.png')

import gurobipy as gp
from gurobipy import GRB
import numpy as np
import pandas as pd
import scipy.sparse as sp
import matplotlib.pyplot as plt


def get_exact_solution(cp, cf, cs, d, g):
    time_periods = np.arange(1, len(d) + 1)
    model = gp.Model('ProductionMgmtProblem')

    x = model.addVars(time_periods, vtype=GRB.CONTINUOUS,
       name='x_')
    y = model.addVars(time_periods, vtype=GRB.BINARY, name='y_')
    z = model.addVars(np.arange(0, len(d) + 1),
       vtype=GRB.CONTINUOUS, name='z_')

    model.setObjective(gp.quicksum(x[t] * cp[t - 1] for t in
       time_periods)
                        + gp.quicksum(y[t] * cf[t - 1] for t in
                           time_periods)
```

```
                   + gp.quicksum(z[t] * cs[t - 1] for t in
                      time_periods), GRB.MINIMIZE)

model.addConstrs(z[t - 1] + x[t] == d[t - 1] + z[t] for t
   in time_periods)

model.addConstrs(x[t] <= g * y[t] for t in time_periods)

model.addConstr(z[0] == 0)

model.optimize()

obj_value = model.objVal

x_star = []
y_star = []
z_star = []
for t in time_periods:
    x_star.append(x[t].x)
    y_star.append(y[t].x)
    z_star.append(z[t].x)

return x_star, z_star, obj_value
```