Building a Tank Schedule using Optimization

Contents

[Generic Problem Definition 1](#_Toc171353378)

[Formulating the Problem as a Mixed Integer Program 3](#_Toc171353379)

[Python Program 6](#_Toc171353380)

[DataInputProcessing() 7](#_Toc171353381)

[DataOptimization() 11](#_Toc171353382)

[DataLocation() & DataCycle() 11](#_Toc171353383)

[DataOptimization 12](#_Toc171353384)

[Optimization 13](#_Toc171353385)

# Generic Problem Definition

To provide the reader with the problem definition we introduce the concept of a Tank Farm & Line Schedule.

**Tank Farm**

A Tank Farm is comprised of

* Tanks: Storage units used for storing petroleum products
* Lines
  + Inbound: Lines heading into the farm and connected to a subset of Tanks
  + Outbound: Lines emanating from Tanks

**Line Schedule**

A Line Schedule describes the scheduled movement of petroleum products from the Inbound Lines delivered to the Outbound Lines and can be described in a tabular format like below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Line** | **Product** | **Volume** | **Time Start** | **Time End** |
| 1 | A | 20000 | 01/01/2023 08:00 | 01/01/2023 10:00 |
| 1 | D | 20000 | 01/01/2023 10:00 | 01/01/2023 13:00 |
| 2 | 54 | 15000 | 01/02/2023 00:00 | 01/02/2023 03:00 |
| 2 | 62 | 15000 | 01/02/2023 03:00 | 01/02/2023 06:00 |
| 14 | A | 15000 | 01/01/2023 13:00 | 01/01/2023 17:00 |
| 15 | D | 20000 | 01/01/2023 12:00 | 01/01/2023 18:00 |
| 16 | 54 | 20000 | 01/02/2023 02:00 | 01/02/2023 05:00 |
| 17 | 62 | 10000 | 01/02/2023 02:00 | 01/02/2023 07:00 |

**Tank Inventory & Capacity Table**

A Tank Inventory & Capacity Table describes the volume of product present in each Tank at the beginning of the Line Schedule as well as each Tank maximum capacity.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tank** | **Product** | **Volume** | **Capacity** |
| 310 | A | 1500 | 20000 |
| 311 | D | 3000 | 30000 |
| 312 | 54 | 5000 | 30000 |
| 313 | 62 | 4000 | 30000 |
| 315 | A | 4000 | 30000 |
| 330 | D | 3000 | 20000 |
| 350 | 54 | 2000 | 20000 |
| 351 | 62 | 2000 | 20000 |

**Problem Definition – Build a Tank Schedule**

The problem statement is as follows: Given a Line Schedule and a Tank Inventory Table, build a 'Tank Schedule.' This involves routing the product flows described in the Line Schedule through the set of farm tanks. The resulting schedule can be presented in a tabular format similar to the Line Schedule, as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Line** | **Product** | **Tank** | **Volume** | **Time Start** | **Time End** |
| 1 | A | 310 | 10000 | 01/01/2023 08:00 | 01/01/2023 09:00 |
| 1 | A | 311 | 10000 | 01/01/2023 09:00 | 01/01/2023 10:00 |
| 2 | 54 | 312 | 5000 | 01/02/2023 00:00 | 01/02/2023 02:00 |

# Formulating the Problem as a Mixed Integer Program

We formulate the above problem as a Mixed Integer Program. We first describe a set of indices we will use when defining the variables:

* 1. Indices
     1. Tanks: τ ϵ Tanks = {310, 311, 312, .., 375}
     2. Products: p ϵ P = {A, D, 54, 62}
     3. Time: t ϵ Time = {1, 2, 3, …, T}
     4. Lines: l ϵ L = (1,2, 14, 15, 16, 17)
  2. Variables

The following variables are fundamental to the problem definition:

* + 1. **X (τ, p, t):** represents the volume of product p in tank τ at time t (continuous)
    2. **I (τ, l, p, t):** inbound volume of product p from line l to tank τ at time t (continuous)
    3. **O(τ, l, p, t):** outbound volume of product p from line l to tank τ at time t (continuous)
    4. **P(τ, l, p, t):** indicator of inbound volume of product p from line l to tank τ at time t (binary)
    5. **Q(τ, l, p, t):** indicator of inbound volume of product p from line l to tank τ at time t (binary)

Subsequently, we introduce a number of variables so we can model more complicated concepts:

* + 1. **TI (τ, l, p, t), TO (τ, l, p, t):** Similar the previous variables, these variables are defined as P(τ, l, p, t+1)-P(τ, l, p, t) and Q(τ, l, p, t+1)-Q(τ, l, p, t).
    2. **ABS\_TI (τ, l, p, t), ABS\_TI (τ, l, p, t):** The absolute value of **TI** and **TO** variables. This variable indicates the number of “switches”, that is, how many times the **Tank** has seen an on/off change in flow. This variable is used to enforce more uniform flows without interruptions per Tank.
    3. **T(τ):** An indicator of whether Tank τ has been used in the schedule (binary)
  1. Model Inputs

**Line Schedule**: This input is a modified version of the Line Schedule described in the previous section. It represents the volume of Product p routed through line l at discrete time t. Here, the discrete time t is considered as an hour counted from the beginning of the Line Schedule. The Line Schedule table will need to undergo some processing to derive the following input:

* + - * Line\_Schedule (l, p, t) for l ϵ L, p ϵ P, t ϵ Time
    1. **Tank Inventory** 
       - Tank\_Inventory (τ, p) for τ ϵ Tanks, p ϵ P
       - Capacity (τ), for τ ϵ Tanks
    2. **User**
       - Selected\_Tanks: A subset of T denoting the Tanks that can be used when building a schedule
       - Flow\_contraints\_inbound(τ) & Flow\_contraints\_outbound(τ) for τ ϵ T:

Control how many interruptions the inbound & outbound flows can have for a specific tank

* + - * Flow\_contraints\_inbound\_universal & Flow\_contraints\_outbound\_universal:

A universal limit on the interruptions of inbound & outbound flows

* 1. Constraints
     1. Balance Constraints

The Tank volume at time t + 1 equals the volume at time t plus the inbound and minus the outbound flows.

* **X**(τ, p, t + 1) = **X**(τ, p, t) + **P**(τ, l, p, t) \* **I**(τ, l, p, t) - **Q**(τ, l, p, t) \* **O**(τ, l, p, t)

for τ ϵ Tanks, p ϵ P, t ϵ Time, l ϵ L

* + - * Initial Condition

Where the inventory is defined as part of the initial conditions.

**X**(τ, p, 0) = Inventory (τ, p) for τ ϵ Tanks, p ϵ P

* + - * Tank Capacity

Where capacity is defined as part of the initial conditions.

<= Capacity (τ), for τ ϵ Tanks, t ϵ Time

* + - * Product

A line can only direct or accept product from one Tank at a time.

<= 1 for l ϵ L, p ϵ P, t ϵ Time

<= 1 for l ϵ L, p ϵ P, t ϵ Time

* + 1. Flow

The Schedule is part of the problem definitions.

= Line\_Schedule (l, p, t) for l ϵ L, p ϵ P, t ϵ Time

= Line\_Schedule (l, p, t) for l ϵ L, p ϵ P, t ϵ Time

* + 1. Inbound & Outbound Tank Flow

Defining how smooth we want the Tank flow to be. A value of 2, for example, enforces a unform (without interruptions) flow for each Tank.

* + - * <= Flow\_contraints\_inbound(τ) for τ ϵ Tanks, l ϵ L, p ϵ P
      * <= Flow\_contraints\_outbound(τ) for τ ϵ Tanks, l ϵ L, p ϵ P
      * <= Flow\_contraints\_inbound\_universal, for τ ϵ Tanks, l ϵ L, p ϵ P
      * <= Flow\_contraints\_outbound\_universal, for τ ϵ Tanks, l ϵ L, p ϵ P
    1. Manipulate the number of Tanks
       - Count the number of used Tanks

**P**(τ, l, p, t) <= **T**(τ), for τ ϵ Tanks, p ϵ P, t ϵ Time, l ϵ L

**Q**(τ, l, p, t) <= **T**(τ), for τ ϵ Tanks, p ϵ P, t ϵ Time, l ϵ L

* + - * Turn off some Tanks (Selected\_Tanks)

**P**(τ, l, p, t) = 0, for τ ϵ Selected\_Tanks, p ϵ P, t ϵ Time, l ϵ L

**Q**(τ, l, p, t) = 0, for τ ϵ Selected\_Tanks, p ϵ P, t ϵ Time, l ϵ L

* 1. Objective:

One possible objective is to limit the number of used Tanks. In this case the objective can be expressed as

Min

# Python Program

The problem formulation and solution are handled using a Python program. The files are located in a Git repository named Colonial-Storage-Solver. The program is executed by running app.py. Below is a basic overview of the program, with more detailed information on each class provided in the following sections:

1. **Input Files:**
   * The program expects two input files placed in the input\_raw folder:
     + TFInvSample.csv
     + TRM\_ScheduleData.csv
2. **App.py:** This is the execution file, and the following is happening in this order
   * **Class Initialization**
     + **DataInputProcessing()**: This class is initialized first. It reads the input files and transforms them into a set of dimensional tables, which are then placed in the input\_location and input\_cycle folders.
     + **DataOptimization()**: This class is initialized next. It runs a series of validations on the processed data and prepares the necessary data structures to build the optimization problem. All the structures are stored in the member dictionary inputs
   * **Reading Model Inputs:** We read the following files from the input\_optimization folder. These files are generated by the application that calls the program and represent the user preferences manifesting as problem constraints:
     + Selected\_Tanks : Read from input\_optimization/tanks.csv and it is the set of Tanks to be used. It is stored as an array into inputs[‘user\_tanks’].
     + Flow\_contraints\_inbound & Flow\_contraints\_outbound: Read from input\_optimization/flowTanks.csv and stored into inputs[‘flow\_contraints’]

One such example is

inputs['flow\_constraints'] =

[{310, 'Inbound': 2, 'Outbound': 2}, {311, 'Inbound': 4, 'Outbound': 4}]

* + - Flow\_contraints\_inbound\_universal & Flow\_contraints\_outbound\_universal: Read from input\_optimization/misc.csv. One such example is universal Tank flow bounds

A screenshot of a computer

Description automatically generated

Stored as inputs[‘flow\_contraints\_univ\_1’] and inputs[‘flow\_contraints\_univ\_2’]

* + **Optimization Process:**
    - The program calls a static function of the OptimizationModel class with the inputs dictionary as an argument. This function builds the optimization model, sends it to the solver, and processes the solution into a schedule that is saved in the results folder.

In the following sections, we will describe the three classes in the order they are used: DataInputProcessing(), DataOptimization(), and OptimizationModel.

## DataInputProcessing()

The class is contained in class\_data.py file and if instantiated first in app.py. Its main task is to read the raw input files and transform the data into a set of dimension and fact tables, following the principles of dimensional modeling. We generate dimension tables for the main problem entities: (i) Tanks, (ii) Lines, and (iii) Products.

The purpose of this transformation is two-fold: (i) the resulting tables make it easy to review and audit the data, and (ii) it provides a cleaner path for building the optimization structures in subsequent steps.

The constructor calls the following functions in order:

* staging()
* create\_dimension\_tanks()
* create\_dimension\_products()
* create\_dimension\_lines()
* create\_fact\_line\_schedule()
* create\_fact\_tank()

The functions above also require the presence of two additional files input\_location/topoI.csv and input\_location/topoO.csv. These are called topology files and are related to the tank farm for which the problem is solved. They define the Tank Farm topology, i.e., a mapping between the Tank and the lines it is connected to. The two files correspond to the inbound and outbound lines.

**staging()**

There are two input files necessary for the optimization sequence and are placed in input\_raw folder. The files are (i) Tank Inventory, and (ii) a Line Schedule files. The files must have a particular structure and a basic description is given below:

Tank Inventory: This file has been typically exported under the name “TFInvSample”, is at a Tank granularity and describes the Tank volume at the particular snapshot of time typically at the beginning of the Line Schedule.

A screenshot of a table

Description automatically generated

The staging function does the following:

* Replace the “Product” with a corresponding mapping. For example, A2, A3, A4 can be mapped to A.
* If Volume < 0 (data error) set it to zero

Line Schedule**:** This file is typically exported under the name “TRM SxcheduleData” and described a “Line Schedule”. A pair of lines indicated by “BATCHSTART” and “BATCHSTOP” indicate a particular product movement for the corresponding line within the indicated time frame.

**A table with numbers and letters

Description automatically generated**

The staging function does the following:

* Apply the product mapping to the “Batch” column by replacing the middle component

**create\_dimension\_tanks()**

This function creates a Tank dimension table which is saved as input\_location/dim\_tanks.csv. The table is based off the aforementioned Tank Inventory input file. We rename some of the existing columns, divide the volumes by 1000 and round to zero, and for every Tank indicate which lines this Tank can accept or deliver product to as indicated by columns LinesIn & LinesOut.

The columns LinesIn & LinesOut are built using the topology tables mentioned in the beginning of this section.

The dimension table looks like this:

A table with numbers and lines

Description automatically generated

**create\_dimension\_products()**

This function creates a Product dimension table which is saved as input\_location/dim\_products.csv. The table is based on both the Tank Inventory & Line Schedule input files by selecting all products available in both files. The “TankFile” column indicates if the product is available in the Tank Inventory file. The other two columns indicate if the product shows up in the inbound and outbound lines.

The dimension table looks like this:

A screenshot of a computer

Description automatically generated

**create\_dimension\_lines()**

This function creates a Line dimension table which is saved as input\_location/dim\_lines.csv. The table is based off the aforementioned Line Schedule input file by getting the unique set of lines.

The “Tank” column indicates which lines the Tank is connected to, and it is built using the topology files mentioned in the beginning of this section. The dimension table looks like this:

A screenshot of a computer

Description automatically generated

**create\_fact\_lineSchedule()**

This is probably the most crucial and complicated function of this class. It transforms the Line Schedule file by changing its granularity to an hourly level. In the example below, each row corresponds to an exact hour. Grouping by Line and Product should result in the original Line Schedule file with the difference that timestamps are rounded down to the hour.

A table with numbers and letters

Description automatically generated

This transformation is necessary because the optimization model considers flow variables (X) that represent product flow at discrete time intervals. Since the chosen time interval is one hour, we need to transform the Line Schedule to an equivalent hourly schedule.

**create\_fact\_tanks()**

This function creates a table of similar granularity to dim\_tanks but it contains Tank related facts.

**A screenshot of a table

Description automatically generated**

## DataOptimization()

The DataOptimization class inherits from DataCycle() which on turn inherits from DataLocation(). We describe these two classes next.

### DataLocation() & DataCycle()

These classes perform a number of validations on the dimension and fact tables we’ve built in the previous step. One way to group these validations is by considering the dimension tables to relate to "Location." The dimension tables contain fixed information about the physical tank farm under consideration and, therefore, can be considered to define the Location of the tank farm.

On the other hand, the fact tables contain the "facts" defined by the particular cycle under consideration. Therefore, we can consider them to define the Cycle under consideration.

In light of the above, we created two classes: DataLocation, which contains the dimension table validations, and DataCycle (that inherits from DataLocation) which contains the fact table validations. Below is a basic overview of the validations:

* DataLocation
  + validation\_level0\_dim: this function checks the table metadata, for example whether the tables have the columns we expect.
  + validation\_level1\_dim\_tanks: Check for correct granularity and null values for certain columns
  + validation\_level1\_dim\_products: Check for correct granularity and null values for certain columns
  + validation\_level1\_dim\_lines: Check for correct granularity and null values for certain columns
  + validation\_level2\_dim\_1: Check the topology tables and whether their Tank values match between them and the dim\_tanks table.
  + validation\_level2\_dim\_2: This validation rule ensures that there is a feasible path (from a tank to the line) to serve the product(s) at the end of each stubline (output). The process begins with dim\_lines, where it retrieves the tanks associated with each line. Then, it retrieves the products associated with each tank. Finally, it verifies that the product from dim\_lines is a subset of the products associated with the tanks.
* DataCycle
  + validation\_level1\_fact\_tanks: Check for correct granularity and null values for certain columns

The conditions listed below can lead to model infeasibility, which can result in many hours of debugging. These conditions represent the most frequent causes of model infeasibility.

* + validation\_level2\_fact\_1: This function groups the product flows by Line/Product and checks if the sum of the existing volume and inbound volume meets or exceeds the outbound requirements. If not, it throws an error.
  + validation\_level2\_fact\_2: This function performs a detailed check on volume flows by aggregating them hourly. It verifies whether the running sum of inbound versus outbound volumes ever becomes negative. If it does, it throws an error. This critical validation prevents the model from becoming infeasible and saves many debugging hours.
  + validation\_level2\_fact\_3: A simple rule that checks whether dim\_tanks and fact\_tanks are of the same granularity.
  + validation\_level2\_fact\_4: Check dim\_tanks and fact\_tanks and whether ‘Volume’ column is less than ‘Working’. If not then throw an error.

To summarize, we organize our data into a dimensional format to provide a structured and well-defined set of tables, **facilitating the validation processes described above**. The rules described above are crucial, as they ensure the resulting model is always feasible.

### DataOptimization

This class builds and stores the data structures required to build the optimization model. It inherits from DataCycle, and all derived data structures are stored as members of the self.input dictionary. Below, we describe the members of self.input dictionary are:

* dim\_tanks
* dim\_products
* dim\_lines
* fact\_LineScedule
* fact\_tanks
* CycleStart
* T
* Time
* index

All the above class members are inherited by DataCycle except that last one, index, a dictionary that is built by the function self.function\_index and defines the state space for the variables of the optimization problem. This operation uses two data structures, build by the topology files we’ve seen before input\_location/topoI.csv and input\_location/topoO.csv.

The two data structures used for building index are:

self.topo\_i, self.topo\_o = self.load\_topology()

For example, inputs['index']['x'] defines the state space of the variable X and is a list of tuples that look like this (this is a limited length sample):

**[(310, 'A', 0), (310, 'A', 1), (310, 'A', 2), (310, 'A', 3), (310, 'A', 4), (310, 'A', 5), (310, 'A', 6), (310, 'A', 7), (310, 'A', 8), (310, 'A', 9)]**

inputs['index']['i'] defines the state space of the variable i and is a list of tuples that look like this:

**[(310, '01', 'A', 0), (310, '01', 'A', 1), (310, '01', 'A', 2), (310, '01', 'A', 3)]**

In a similar fashion we build the state space for the rest of the variables. The reader can review the relevant code to get more insights on how inputs is built.

## Optimization

The built and solution of the Optimization model is performed by a class called OptimizationModel containing a number of static member functions. The main function to call is model\_stage3(ID, inputs)

If you inspect the body of the function you will

* The first thing is to create a model object and add the variables to it. This is a fairly straightforward operation. For example, adding the variables X is performed by

x\_index = index['x']

x       = model.addVars(x\_index,  lb = 0.0, ub = 200, vtype = GRB.CONTINUOUS)

* Next we pass the model functions that will add the constraints such as
  + model=OptimizationModel.model\_baseline\_const1(model,inputs)

This function adds the basic balance flow constraints

* + model=OptimizationModel.model\_baseline\_const2(model,inputs)

This function aligns the variables with the Lines Schedule volumes

* + model=OptimizationModel.model\_flow\_const2(model,inputs)

This function set the constraints relating to Tank flow (variables **TI, TO**)

* + model=OptimizationModel.model\_tank\_const1(model,inputs)

This function turns off the Tanks that are not used

* + model=OptimizationModel.model\_tank\_const3(model,inputs)

This function constructs a variable that counts the number of utilized Tanks

* + model=OptimizationModel.model\_obj(model,inputs)

This function sets the objective function

* After the optimization is complete we check the model.Status and if the model has completed successfully we run the following function

DataAnalysis.schedule(ID, inputs)

that takes the optimal solution and produces a schedule which is saved in the results folder.