Amadeus Task

Oscar Nieves

March 2022

The task is to develop an algorithm which can solve the following optimization problem: suppose that you are in charge of a company which can own at most one machine at a time. This machine, call it M_i, can generate a G_i profit from day $D_i + 1$ onwards, where D_i is the day in which the machine is available for purchase for a price P_i. For any machine M_i which is not bought on day D_i, it is no longer available afterwards. At any point in time t, the machine that is already owned by the company, call it $M_i^{(\text{owned})}$, can be resold for a price $R_i < P_i$ (that is, the resale price is always lower than the purchase price), and on the very same day t another machine can be bought in its place. The only drawback to this is that for that day in which a machine is sold and another one bought, there will be no profit generated, as any new machine can only start operating on the day after it is purchased.

The main goal is to find the maximum amount of money, call it $A_{\max}$, which the company can make during the entire restructuring period.

1 Assumptions for the problem

As stated in the problem definition, we have the following constants which work as inputs to the optimizer:

- $N \leq 100$, the number of machines for sale during the entire restructuring period
- $C \leq 100$, the number of dollars which the company starts with
- $D \leq 100$, number of days which the restructuring period lasts

Furthermore, we are told that the following constraints hold:

- $1 \leq D_i \leq D$
- $1 \leq R_i < P_i \leq 100$
- $1 \leq G_i \leq 100$

Evidently, this is an integer optimization problem, and therefore the maximum amount of money that the company can make by the end of the restructuring period $A_{\max}$ (we shall denote the day number as t and the end of the restructuring as $t = D + 1$), must also be a positive integer.

Before we construct the algorithm, we need to simplify the problem by making some reasonable assumptions. First, we assume that the company does not have a machine to begin with, so at day $t = 0$ there is no machine and therefore no profit to be maid. Second, we can assume that it is impossible to buy a machine M_i if the amount of money at time t, call it A_t, is more than the

purchase price P_i. Third, we shall assume that machines for sale are only available on a single day D_i, and so they cannot be bought earlier or later than day D_i. However, the availability day D_i need not be unique to each machine M_i, for instance we could have 2 or more machines being sold on the same day, which means we only have the choice of ever purchasing 1 of them. In summary, we can make the following underlying assumptions:

1. Initially, the company CVCM does not own any machines, so at day $t = 0$ there is no profit to be made.
2. A machine M_i can only be purchased on day D_i if the amount of money the company has at time t is $A_t \geq P_i$. This means that $A_0 = A_1 = C$.
3. Machines can only be bought on D_i, not earlier nor later. Machines which are not purchased are out of the market forever.
4. D_i is not unique to each machine, in fact multiple machines could be available for purchase on the same day.
5. Let time t represent de day number, namely $t = 0, 1, 2, \ldots, D + 1$. The restructuring period begins on $t = 1$ and ends at $t = D + 1$.
6. On the final day $t = D + 1$, the company CVCM needs to sell whatever machine it still owns, meaning that no profit from the machine can be generated from that day (this is stated in the problem definition).

2 The algorithm

I will model this problem as a network flow problem. I am taking inspiration from the following document titled "Airline's Crew Pairing Optimization: A Brief Review" by Xugang Ye (link: `https://www.cis.jhu.edu/~xye/papers_and_ppts/ppts/airline_crewpairing_copy1.pdf`), since my current job at Qantas involves working with crew pairing and rostering software, and I have an elementary understanding of how this type of network problem works.

The idea behind the network is as follows: suppose that we are given a set of N machines $\mathbf{M} = \{M_1, M_2, M_3, \ldots, M_N\}$. Each machine $M_i \in \mathbf{M}$ has a set of 4 integer values, namely $\Omega_i = \{D_i, P_i, R_i, G_i\}$. Because a machine M_i is only available for sale at day D_i, as time progresses forward we can only see that specific machine available on only one of the days in the restructuring period. For example, suppose that there are 5 machines. Let machines M_1 and M_2 be available on day 1, machines M_3 and M_4 be available on day 2, and machine M_5 on day 4, and say that day 4 is the last day in the restructuring period. This basically means that if we start our count at $t = 1$, the successive subsets of machines available on each day is shown in Fig.1.

Although Fig.1 shows the sequence of available subsets, call them $\mathbf{m}_t \subseteq \mathbf{M}$ where t is the day number, it does not show us a realistic picture of how the network can be constructed. The reason being that if we buy a machine on a previous day, then automatically that machine is available on the next day's subset, not because it can be purchased again, but rather because we could easily choose NOT to re-sell it. By keeping the machine on the next day too, we are essentially picking the same machine, except this time Ω_i will be modified, call it Ω_i'. How exactly? Suppose that we buy machine M_1 on day 1. Then on day 2, that same machine M_1 will have a modified set of values $\Omega_1' = \{D_1 + 1, P_1' = 0, R_1, G_1\}$. Note here, $P_1' = 0$ because the machine is no longer available for purchase, but we can however resell it at price R_1, or keep it another day so it generates a profit G_1. This means that the sequence of subsets which make up our network will look more like what

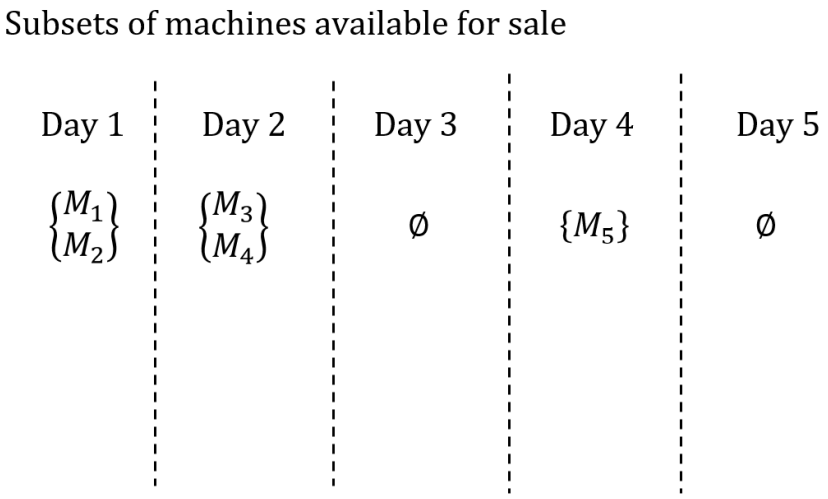

Figure 1: Sequence of available machine subsets, based on their days of purchase availability. $\emptyset$ is the null set, meaning no machines are available for purchasing on that day.

is shown in Fig.2. In this example, we purchase M_1 on day 1, and then keep it on day 2. On day 3, there are no other machines up for sale, so our only choice is to keep M_1 again. On day 4, we decide to re-sell M_1 to get M_5, and then at the end of the restructuring period (day 5) we are left with M_5, which we have to re-sell.

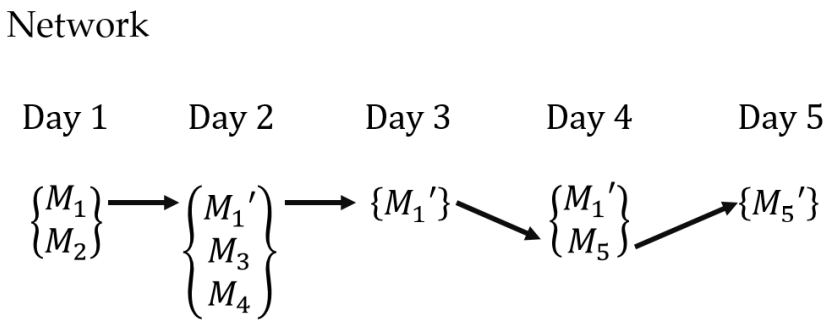

Figure 2: Sequence of machine subsets in the network flow problem, based on their days of purchase availability.

Each of the directed edges of the network carries some "weight" to them. This weight, call it $w_{i,j}^{(t)}$, where i corresponds to some element in the subset $\mathbf{m}_t$ and j corresponds to some element in the subset from the next day $\mathbf{m}_{t+1}$. This means that $w_{i,i}^{(t)}$ is an edge starting at t and ending at $t+1$, in which we have selected the same machine. We will define this weight as the amount by which the total money at the end of each day $A_{i,j}^{(t+1)}$ changes, which is defined by the update equation:

$$A_{i,j}^{(t+1)} = A_{i,j}^{(t)} + w_{i,j}^{(t)}. \tag{1}$$

The idea behind the algorithm is to maximize the function

$$\max\left\{C + \sum_{t=1}^{D} w_{i_t,j_t}^{(t)} + R_{\text{final}}\right\} \tag{2}$$

where C is the initial amount of money we start with, and R_{final} is the re-sale price of whatever machine we own on the very last day $t = D + 1$. This seems a bit complicated, so let's illustrate this with a simple example. Say we have 5 machines and a restructuring period of 4 days with the following values ($\{D_i, P_i, R_i, G_i\}$):

$$\begin{Bmatrix} M_1 \\ M_2 \\ M_3 \\ M_4 \\ M_5 \end{Bmatrix} = \begin{Bmatrix} \{1, 10, 8, 12\} \\ \{1, 12, 9, 13\} \\ \{2, 8, 6, 10\} \\ \{2, 20, 14, 22\} \\ \{4, 15, 10, 18\} \end{Bmatrix}$$

The inputs are also $\{N, C, D\} = \{5, 20, 4\}$. In this particular example, the network will look like this: As we can see, there is a total of 12 possible paths we can take across the entire restructuring

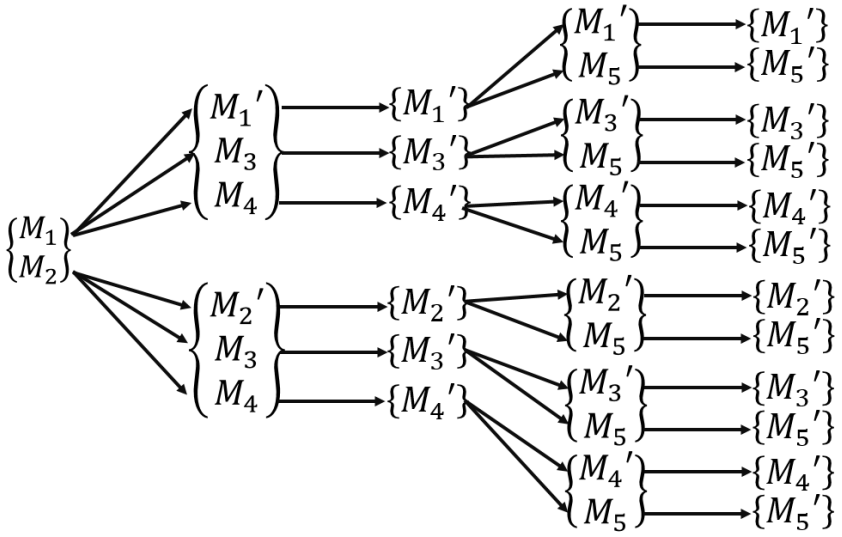

Figure 3: Example of network consisting of 5 machines and 4 restructuring days (the 5th day is when the last machine is resold).

period. The number of paths corresponds to the product of the size of each set $\mathbf{m}_t$ across all t up to $t = D$, with the size $\mathbf{m}_t$ being adjusted with a +1 for each $t > 1$ due to the possibility of selecting the same machine again. In this case, the network has $2 \times 3 \times 1 \times 2 = 12$ possible connections. We may generalize this further by letting S_t denote the size of $\mathbf{m}_t$ (number of machines in $\mathbf{m}_t$), and writing

$$\text{Total network size} = S_1 \prod_{t=2}^{D} (S_t + 1) = S_1(S_2 + 1)(S_3 + 1) \cdots (S_D + 1). \tag{3}$$

It should be noted here that we have assumed all paths are possible, but this need not be the case always. For instance, it could happen that we do not have enough money to purchase a specific

machine on the next day, so automatically that edge (i, j) cannot be created and so that eliminates one of the paths in the network. As an example, suppose M_4 is too expensive after we purchase M_2 on day 1, then the network will look like this: This in itself presents a small complication, as

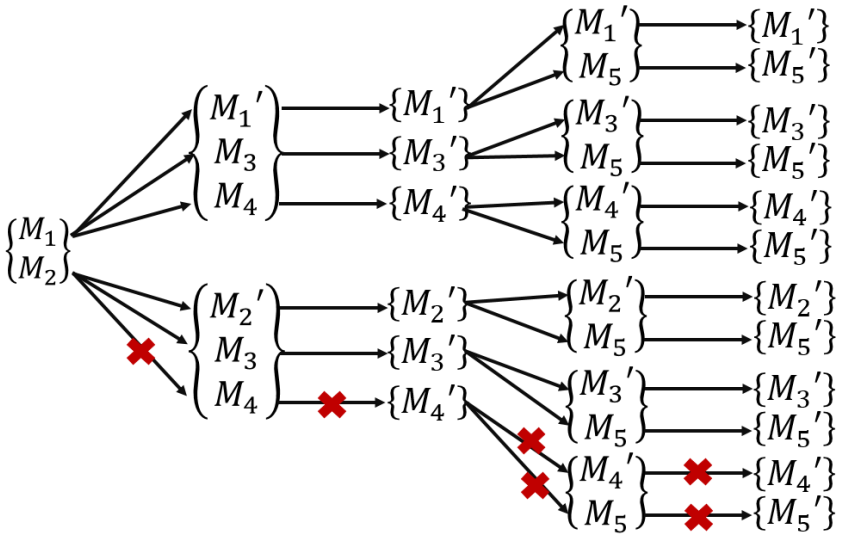

Figure 4: Example of what happens when we don't have enough money at a certain day t to purchase some machine M_i. In this case, the network ends up with 10 possible paths instead.

we won't know beforehand if we will have enough money to purchase a certain machine at a certain day. One way around this is to redefine our weight function, such that if the next machine is too expensive, we assign a large negative value which will automatically make that path obsolete in the long run, for instance:

$$w_{i,j}^{(t)}\left(A_{i,j}^{(t)}\right) = \begin{cases} G_i^{(t)} & \text{if } i = j \text{ and } t > D_i \text{ (machine is owned and operating)} \\ -P_i^{(t)} & \text{if } i = j \text{ and } t \leq D_i \text{ (machine is bought today)} \\ G_i^{(t)} + R_i^{(t)} & \text{if } i \neq j \text{ and } P_i^{(t)} = 0 \text{ (machine is owned today, sold tomorrow)} \\ R_i^{(t)} - P_i^{(t)} & \text{if } i \neq j \text{ and } P_i^{(t)} \neq 0 \text{ (machine is bought today, sold tomorrow)} \\ -L & \text{if } A_{i,j}^{(t)} < 0 \text{ (ran out of money)} \end{cases} \tag{4}$$

where $A_{i,j}^{(t)}$ is the amount of money the company has at the end of day t given the choice of machines (i, j). The variable L is a positive constant we set at the beginning of the solution process and which is large in the context of the problem. For instance, we could set it to the value of the maximum G_i across all machines multiplied by the number of days in the restructuring period:

$$L = D \max\{G_i\}. \tag{5}$$

Now, the way in which the weights are calculated is as follows: One convenient way of organizing the calculation of the weights across each path is to transform our network or graph into a table, call it a network table, where each row represents a path containing a sequence of machine states such as $\{M_1, M_1', M_1', M_1', M_1'\}$, and each column corresponds to a specific day of the restructuring period. In this sense, the network table corresponding to the network in Fig.3 is shown in Fig.6.

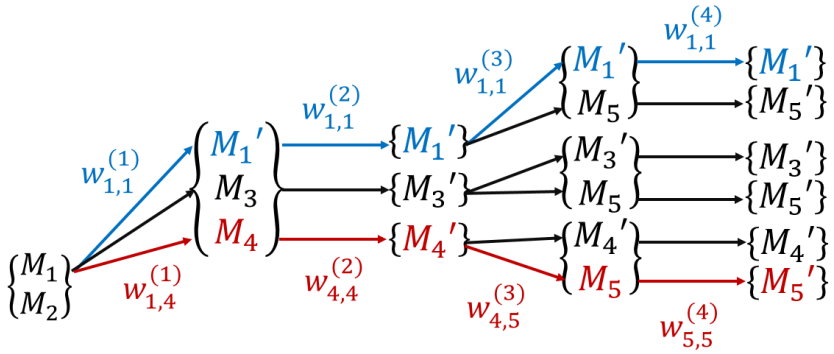

Figure 5: Example of how the weights are assigned to the network edges.

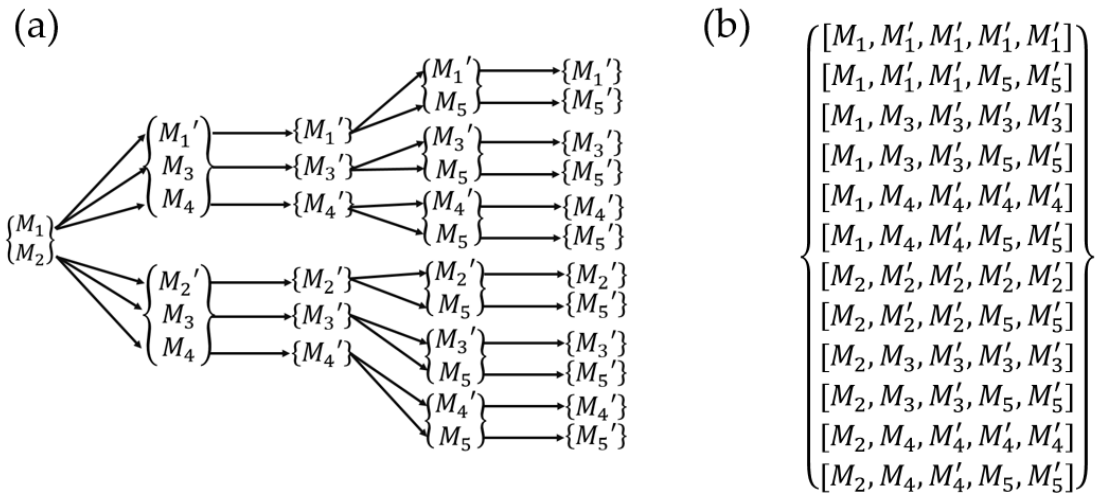

Figure 6: Equivalence between (a) graph-representation of the network and (b) table-representation.

The procedure for generating such a network table is explained in detail in the attached document titled "Table Generation method.pdf". Needless to say, once the network table is created, the optimization procedure becomes a lot simpler.

2.1 The actual algorithm

In pseudo-code, the algorithm work as follows:

```
Step 1: Define {N, C, D}, and the set of subsets M = {{M_1}, {M_2}, ..., {M_N}}

Step 2: Break down M into smaller sets by day of availability
for each t in {1, 2, 3, ..., D} do:
```

```
    for each {M_i} in M do:
        if D_i = t then:
            assign {M_i} to the subset m_t
        else:
            let m_t = ∅
    end for
end for

Step 3: create a network consisting of all possible paths
Use NetworkTable(...) function on the entire array [m_1, m_2, ..., m_{D+1}],
    assign to NT
calculate L

Step 4: Calculate weights and money amounts at each time t
define an empty array of weights W^(t)
define an empty array of money A^(t)
set each value in A^(t=0) to C
for each row number n in NT:
    for each t in {1, 2, 3, ..., D} do:
        calculate weight w_{i,j}^(t)
        append w_{i,j}^(t) to W^(t)
        calculate A_{i,j}^(t+1) = A_{i,j}^(t) + w_{i,j}^(t)
        append A_{i,j}^(t+1) to A^(t+1)
    end for
end for

Step 4: calculate R_final for all possible machines at state t = D

Step 5: find the maximum of w_{i,j}^(D) + R_final across all paths
```

Now, let's attempt to solve this manually for visualization purposes. If we look at the blue path in Fig.5, we know for instance that at $t = 1$ no machine is owned, therefore no profit can be generated on that day. We calculate each of the weights for the path involving only M_1 as follows (recall that $M_1 = \{1, 10, 8, 12\}$ and $\{N, C, D\} = \{5, 20, 4\}$):

$$
\begin{aligned}
w_{1,1}^{(1)} = -10 \quad &\longrightarrow \quad A_{1,1}^{(2)} = 20 - 10 = 10 \\
w_{1,1}^{(2)} = +12 \quad &\longrightarrow \quad A_{1,1}^{(3)} = 10 + 12 = 22 \\
w_{1,1}^{(3)} = +12 \quad &\longrightarrow \quad A_{1,1}^{(4)} = 22 + 12 = 34 \\
w_{1,1}^{(4)} = +12 \quad &\longrightarrow \quad A_{1,1}^{(5)} = 34 + 12 = 46
\end{aligned}
$$

this gives a total amount of money $A_{\text{final}} = C + \sum_t w_{1,1}^{(t)} + R_{\text{final}} = 46 + 8 = 54$ dollars at the end of the restructuring period. We can repeat the same process for each path using the Equation (4) to find out what each weight value should be.

3 The Python code which solved the test cases

The Python code that implements the algorithm described here is called "amadeus_task_oscar_nieves.py":

```
# -*- coding: utf-8 -*-
"""
Amadeus Job interview task
Author: Oscar A. Nieves
Last updated: March 22 2022

The following is a Python 3 implementation of the algorithm outlined in
the associated PDF file entitled "Amadeus Task - Oscar Nieves"
"""
import copy
import pprint
from sys import exit
pp = pprint.PrettyPrinter(indent=2)

# ---------------- Select case number -------------------------------------#
# Enter your case number here (1-7, or 100 = oscar's example)
Case_number = 1

# -------------------------- Load data ------------------------------------#
# Define Input DATA from text file
input_file = "test_case_" + str(Case_number) + ".txt"
try:
    file1 = open(input_file, "r")
except:
    print("ERROR: Text file with case number not found in directory...")
Lines = file1.readlines()
ReadLines = []
for line1 in Lines:
    if line1[-1] == "\n":
        splitLine = line1[:-1].split(" ")
    else:
        splitLine = line1.split(" ")
    for n in range(len(splitLine)):
        splitLine[n] = int(splitLine[n])
    ReadLines.append(splitLine)

# ----------------------- Functions ---------------------------------------#
def Transform(InputMachine):
    InputMachine[1] = 0
    OutputMachine = InputMachine
    return OutputMachine

def SkipEmpty(InputList):
    output = []
    for i in range(len(InputList)):
        if output == [] and InputList[i] == []:
            continue #skip iteration if InputList[i] is empty
        else:
            output.append(InputList[i])
    return output

def CountEmpty(InputList):
    counter = 0
    for i in range(len(InputList)):
        if InputList[i] == []:
```

```
            counter += 1
        else:
            break
    return counter

# Calculate list of possible paths between current set and next set
def MergeSpecial(First, Second):
    len_1 = len(First)
    len_2 = len(Second)
    new_item = [[] for i in range(len_1)]

    # Duplicate and transform last element of each row in First
    First_copy = copy.deepcopy(First)
    for i in range(len_1):
        list_i = First_copy[i]
        new_item[i] = Transform(list_i[-1])

    # Combine in special way
    if Second == []:
        new_copies = copy.deepcopy(First)
        for i in range(len_1):
            new_copies[i].append(new_item[i])
    else:
        copies = [copy.deepcopy(Second) for i in range(len_1)]
        counter_1 = len_2+1
        for i in range(len_1):
            copies[i].insert(0, [new_item[i]])

        new_copies = []
        for i in range(len_1):
            for n in range(counter_1):
                new_copies.append(copies[i][n])

        # Insert First values at the back of Second
        index0 = 0
        len_first = len(First[0])
        for i in range(len_1):
            for n in range(counter_1):
                for j in range(len_first):
                    new_copies[index0].insert(j, First[i][j])
                index0 += 1

    output = new_copies
    return output

# Generate network table of all possible paths
def NetworkTable(InputSetSequence):
    output = []
    len_1 = len(InputSetSequence)
    i = 0
    output = copy.deepcopy(InputSetSequence[i])
    while i < len_1-1:
        list_current = output
        list_next = copy.deepcopy(InputSetSequence[i+1])
        output = MergeSpecial(list_current, list_next)
        i += 1
    return output

# Weights calculator
def w(i=1, j=1, t=1, D_i=1, P_i=1, R_i=1, G_i=1, A_ij=0, L=1):
```

```
    if A_ij < 0:
        return -L # Ran out of money
    else:
        if i == 0 or j == 0: # No machine bought so far
            return 0
        else:
            if i == j and t > D_i: # machine is owned and operating
                return G_i
            elif i == j and t <= D_i: # machine is being bought today
                return -P_i
            elif i != j and P_i == 0: # machine is owned today, sold tomorrow
                return G_i + R_i
            elif i != j and P_i != 0: # machine is bought today, sold tomorrow
                return R_i - P_i

# ------------------------Run program, Processing Inputs----------------------#
inputs = ReadLines[0]
N = inputs[0]
C = inputs[1]
D = inputs[2]

if len(ReadLines) == 1: # if no machine values detected
    print("Case " + str(Case_number) + ": " + str(C))
    exit(0)

M_initial = []
for n in range(1,len(ReadLines)):
    M_initial.append(ReadLines[n])

# Sort array M in ascending order of 1st element of each machine
M = sorted(M_initial, key=lambda x: x[0], reverse=False)

# Break down M into smaller subsets
m_subsets = [[] for i in range(D+1)]
for t in range(1, D+1):
    for i in range(N):
        M_i = M[i]
        D_i = M_i[0]
        if D_i == t:
            M_i.append(i+1) #append machine number at the end
            m_subsets[t-1].append([M_i])
        if t == 1 and i == 0:
            m_subsets[t-1].append([[1,0,0,0,0]]) # no machine at start
        else:
            continue

# Generate Network table
m_input = SkipEmpty(m_subsets)
network = NetworkTable(m_input)
network_size = len(network)
start_index = CountEmpty(m_subsets)

# Set negative value for implausible edges
G_array = []
for i in range(N):
    G_array.append(M[i][-1])
L = D*max(G_array)

# Begin optimization procedure
W = [[0]*D for n in range(network_size)]
```

```
A = [[C]*(D+1) for n in range(network_size)]

for n in range(network_size):
    row = network[n]
    t = start_index
    day = t+1
    for k in range(len(row)-1):
        i = row[k][-1]
        j = row[k+1][-1]
        D_i = row[k][0]
        P_i = row[k][1]
        R_i = row[k][2]
        G_i = row[k][3]
        A_ij = A[n][t]
        w_ij_t = w(i, j, day, D_i, P_i, R_i, G_i, A_ij, L)
        W[n][t] = w_ij_t
        A[n][t+1] = A[n][t] + w_ij_t
        t += 1
        day += 1

# Compute R_final
A_final = [0 for i in range(network_size)]
A_values = copy.deepcopy(A)
for n in range(network_size):
    row = network[n]
    R_prev = row[-1][2]
    A_final[n] = A_values[n][-1] + R_prev

# Find maximum value at t = D+1
A_max = max(A_final)
index_at_max = A_final.index(A_max)

# Display output
print("Case " + str(Case_number) + ": " + str(A_max))
```

The Python script works as follows: in the same directory as the script, put in the text file containing each case, call them "test_case_1.txt" and so on. On line 17 of the Python script, put in the case number you want to calculate the output of, for instance 1 through 7 for the test cases given. I have separated the test cases into separate text files to make this easier. Also, I have included case number 100, which corresponds to the example I talked about in this document.

The output of the code for each case, compared to the value computed manually (or given in the problem definition) is shown in the table below:

Case number	Output (manual or given)	Output (from Python script)
1	44	44
2	11	11
3	12	12
4	10	10
5	39	39
6	39	39
7	0	0

Table 1: Output values for each test case.

As we can see, all test cases match the values I expect from manual computation. I should note here that some of these cases, like case 3 and 4 only have one machine for sale, and buying that machine

gives a different final money value than 12 and 10 respectively. The reason I chose the output to be 12 and 10, which is the input amount of money, is because this value is larger than what you would get from buying those machines, and as such it is more cost-efficient to not buy any machine at all in those cases.

3.1 Some minor remarks on the code and algorithm

I realized after testing a few versions of the code that I didn't account for the case in which we don't buy any machine on Day 1 (which is an option), even if there are machines available on that day. For this, I added lines 157-158, so that an "empty" machine state is present in the network. This inevitably changes the size of the network that I already anticipated, so the computation in Equation 3 is no longer valid once I add this empty machine. However, the operation of the algorithm remains unchanged.