A Genetic Algorithm Solution for the Generalized ATM Cash Replenishment Optimization Problem

A.Tarik Temur
tarik.temur@metu.edu.tr

Kadircan Aksoy
kadircan.aksoy@metu.edu.tr

I.Hakki Toroslu
toroslu@ceng.metu.edu.tr

1. INTRODUCTION

The ATM Cash Replenishment Optimization problem is a well known problem for the banking industry. Banks want to keep their ATM's loaded at all times, while minimizing the amount of idle cash that is in an ATM, since they can not make any interest money out of idle cash. However, loading itself also has a cost due to factors like transportation and security. This proposes an interesting game theoretic situation, where we have two conflicting costs that we want to optimize. We want to load an ATM as frequently as possible to minimize the interest loss while at the same time we want to load as seldom as possible to minimize the loading cost.

Let us define these two costs more formally. First, we assume that a reliable prediction method as to how much money will be withdrawn from a given ATM and a given time interval exists. Now we let I be a matrix of size DxD, where D is the length of the time interval which we want to optimize over, in days. We also assume that we can load an ATM only **once** per day and that no money is lost due to interest in one day (In other words, the diagonal of I is all zeros). The upper triangle of I, holds the values for the amount of money lost, for any sub-interval of the original interval. So any entry $I[i, j]$, holds the amount of money lost if we were to load the ATM on day i, up to day j. The precise details on how this matrix I is calculated can be found in (Toroslu, et.al)

For the loading cost, we are going to assume that loading any ATM has a fixed cost, say α and this value is independent of the time interval in which we are loading the ATM.

We can then look at the recurrence relation from (Toroslu, et.al) for the single ATM case:

$$c_1[i,j] = min \begin{cases} min_{i \leq k < j}(c[i,k] + c[k+1,j]) \\ I[i,j] + \alpha \end{cases} \quad (1)$$

In the scope of this paper, we investigate what happens when we generalize the problem to more than one ATM and we want to optimize all of them simultaneously.

One very crucial thing we introduce at this step, is that the load cost α does not simply linearly scale with the addition of new ATM's. The justification for this is simple and natural, loading multiple ATM's in one "go" is cheaper then loading them separately. Therefore, we let α_n to be the loading cost of n ATM's together. Furthermore, since every ATM has a different user behaviour and therefore a different interest cost, we let I_k denote the interest cost matrix of ATM number k.

2. LINEAR PROGRAMMING

For the single ATM case, the problem can be expressed as the following Linear Program, where a decision variable x_{ij}, indicates whether the ATM has been loaded on day i, up to day j :

$$\sum_{i=1}^{D}\sum_{j=i}^{D}(x_{ij})(I[i,j]+\alpha) \quad (2)$$

subject to

$$x_{ij} \in \{0,1\}$$

$$\sum_{k=1}^{D} x_{1k} = 1$$

$$\sum_{j=J}^{D} x_{Jj} - \sum_{j=1}^{J-1} x_{j(J-1)} = 0, \quad \forall_{1<J<D}$$

$$\sum_{k=1}^{D} x_{kD} = 1$$

Once a second ATM is introduced, the Linear Program has to be modified to handle different loading costs for loading the ATMs together or separately. The new decision variables are of the form $x_{ijk}^{(n)}$, where i and j denote the beginning and end of the loading period, $k \in \{0,1\}$ denotes whether if both ATMs were loaded together or not and $n \in \{1,2\}$ denotes the ATM number. The new objective function takes the following form:

$$\sum_{i_1=1}^{D}\sum_{j_1=i_1}^{D} x_{i_1 j_1 0}^{(1)}(I_1[i_1,j_1]+\alpha) + x_{i_1 j_1 1}^{(1)}(I_1[i_1,j_1]+\frac{\beta}{2})$$

© 2020 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

$$+\sum_{i_2=1}^{D}\sum_{j_2=i_2}^{D} x^{(2)}_{i_2 j_2 0}(I_2[i_2,j_2]+\alpha)+x^{(2)}_{i_2 j_2 1}(I_2[i_2,j_2]+\frac{\beta}{2})$$

subject to

$$\sum_{j=1}^{D} x^{(n)}_{1j1}=1,\ \text{for } n\in\{1,2\} \tag{3}$$

$$\sum_{j=J}^{D} x^{(1)}_{Jj0}+x^{(1)}_{Jj1}-\sum_{i=1}^{J-1} x^{(1)}_{i(J-1)0}+x^{(1)}_{i(J-1)1}=0,\ \text{for } n\in\{1,2\} \tag{4}$$

$$\sum_{j=1}^{D} x^{(n)}_{jD0}+x^{(n)}_{jD1}=1,\ \text{for } n\in\{1,2\} \tag{5}$$

$$\sum_{j_1=1}^{D} x^{(n)}_{dj_1 0}+\sum_{j_2=1}^{D} x^{(n)}_{dj_2 1}<2,\ \text{for } n\in\{1,2\} \tag{6}$$

$$\sum_{j_1=1}^{D} x^{(1)}_{dj_1 1}-\sum_{j_2=1}^{D} x^{(2)}_{dj_2 1}=0 \tag{7}$$

3. GENETIC ALGORITHM

To be able to have an efficient genetic algorithm, we need to define a chromosome structure that responds well to random mutations and one where we can define a meaningful cross-over operation, without resulting in any (or the minimal number of) invalid solution instances. Let us define the following binary matrix as a chromosome:

Let a chromosome C be a DxN matrix of binary digits. Every row corresponds to a day and every column, an ATM.$C_{ij}=1$ means that on day i, the ATM j has been loaded. We do not know up to which day but we do not need to keep this information in the chromosome explicitly either. Because for any column, (i.e any ATM) we can simply infer that an interval between two consecutive 1 entries, defines a load period. The number of 1 entries on a row directly gives us α_n, since it tells us the ATM's are loaded together on that day. From here, we can calculate the fitness of a chromosome with a simple $\mathcal{O}(DN)$ procedure.

4. EXPERIMENTAL RESULTS

For reasonable inputs, where $ND<\epsilon$ for a small ϵ, we are able to use an exhaustive approach to find the globally optimum loading schedule. In inputs of this size, we see that the Genetic Algorithm always finds the optimum solution in a few iterations, regardless of the interest rate or the loading costs.

For larger and more interesting experiments, unsurprisingly, as the solution space grows, the algorithm takes longer to converge. Also unsurprisingly, when the interest rate shrinks, the load costs become competitive with the interest loss and the algorithm has a harder time finding an optimum solution. Consequently, when it grows, the algorithm easily detects the optimum, that is loading every day. However in every case with realistically sized inputs, the algorithm takes no more than a few minutes to converge, on an ordinary PC.

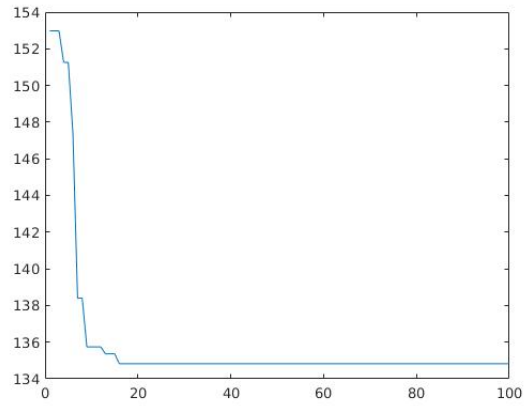

Figure 1: 5 ATM's 7 Days

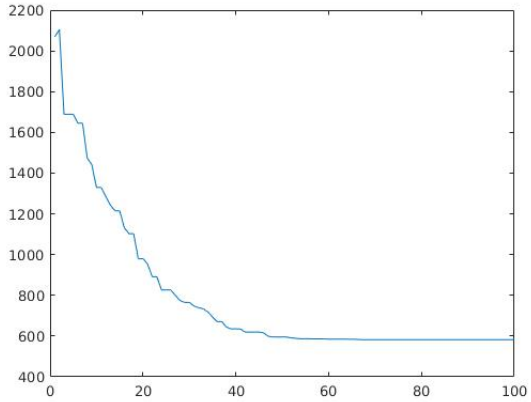

Figure 2: 15 ATM's 15 Days, % 1 interest

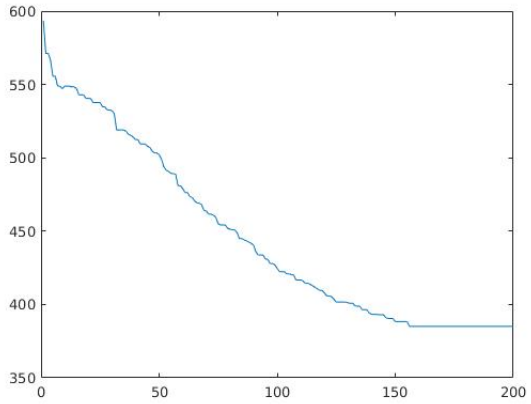

Figure 3: 15 ATM's 15 Days, % 0.1 interest

5. REFERENCES

Ozer, F., Toroslu, I. H., Karagoz, P., & Yucel, F. (2018, October). Dynamic Programming Solution to ATM Cash Replenishment Optimization Problem. *In International Conference on Intelligent Computing & Optimization* (pp. 428-437). Springer, Cham.