EALP

Igor Zlatković
Frankfurt, Germany, EU

WestLB Systems Pacific
Tokyo, Japan

June 18, 2001

Abstract

EALP is an algorithm for allocating goods purchased at different market prices between different customers, or different accounts of the same customer, so that the average prices remain as close to each other as possible.

Contents

1 The Problem **3**
1.1 Introduction 3
1.2 Analysis 4

2 Formulation **5**
2.1 The Problem Table 5
2.2 Equations 7

3 Run Stages **8**
3.1 Stage One (Spreading) 9
3.2 Stage Two (Flipping) 10

4 Implementation **11**

1 The Problem

The allocation problem entered my life back in the days of my internship at WestLB Systems Pacific.[1]

Monday to Friday, nine to five, the stock market in Tokyo was open. The traders were sitting at their desks, their eyes locked on the screens which were showing the prices in realtime, their ears on the phone listening to customer's desires. With each trade their customers earned or lost some money and depending on which of the two was the case, they were more or less ready to tolerate whatever they disliked about the bank. In order to have the traders perform as near to perfect as possible, I was asked to write a program which would automate a part of their task chain and while I was explained what the matter is, I faced a very basic trade problem not known to a commoner.

I am not going to torment anyone with the boring details of the internal business of the bank, but shall try to explain what the problem is using far more leisure an environment than the trading floor.

1.1 Introduction

Imagine you sit in The Beer Garden in Gaien, Tokyo. It is a great place between the trees, illuminated by laterns at night, where you make your own barbecue at your own table and drink tankard for tankard of beer, enjoying the japanese summer.

Now, Imagine the price of each tankard is changing every second, according to customer's demand and the amount on stock. Your every tankard shall cost a different price than the previous one did and you must try your luck and buy when it's cheap. Once you have the beer, you may either drink it[2], or sell it to someone at the nearby table, possibly making a profit.

You are there with two friends. You order three tankards, the other pal orders two and the third lad orders five. The order was placed at the same time, and you expect to get all ten tankards on the table for the same price. As the beer is being filled into the tankards, the price changes, and each tankard costs a different amount, but amounts are relatively close to each other. Sometimes they manage to produce two tankards at the same price, if they fill them fast, but mostly the prices are different. The Beer Garden personnel wants to avoid offending you and your friends, so they need to allocate available tankards to the three of you. They have ten tankards before them, a price attached to each, and must give three of them to you, two to your pal, and five to the third lad, but they now must choose which tankard goes to which person. Because each tankard had costed a different price, they shall choose a combination such that each of you guys has the same average price across the tankards. When they are done, you will have three tankards, your pal will have two and the third lad will have five, each tankard will have costed a different price, but the average price each of you pays per tankard will be about the same.

Sometimes the guys at the bar run out of tankards. We don't accept that. Fill it in a glass or in a coffee mug, I don't care, I want my beer. OK, the barmen have no problem using a different type of container, normal glasses are

[1] *Westdeutsche Landesbank*, a German state bank, the first to show presence on the far east.

[2] It's what I would do anyway.

fine, but in order to keep with the amount sold in tankards, they sell beer in glasses only in units of three, as each tankard can hold three times as much as a normal glass can. Now, in this scenario, we introduce a thing called a *trade unit*. In this case, The Beer Garden personnel has it slightly more difficult, as they can allocate glasses only in units of three, because some angry God of theirs forbade them to break down the trade unit.

Now, the question is how to they do this? How to choose which tankard goes to which person, so that each person has the same average price? Well, they use EALP, hehe. This is exactly what the algorithm does, allocating goods between different customers, or different accounts of the same customer, so that the average price remains the same everywhere.

1.2 Analysis

The problem looked easy at the beginning and the first thought was to use two-stage simplex to solve it, as it looked like a linear programming problem. After a second glimpse, it turned out the problem isn't nearly related to linear programming and the idea about this kind of solution vanished. I have coresponded to many people, some having remarkable degrees and expirience, but noone was able to propose a nice solution. The one or the other bold idea came out, some of them being mine, but each of them had proven to be a failure, as I could either prove their inefficiency mathematically, or find a particular problem they failed to solve.

After having spent a whole night with a pencil in my hand and Mathematica at my aid, I came up with few facts which in turn led to EALP.

I believe the problem to be NP Hard, and I am not alone with that thought. Having such problem means few things. It means that everything related to our problem has a non-linear character, even if it seems not to be so in the beginning. It also means that maybe no algorithms exist which are guaranteed to find an optimal solution.

There is a finite number of possible solutions and we can find the optimal solution by just trying each in turn, always remembering just the best one we saw so far.[3] Unfortunately, while it works with simple particular problems, no God can wait the time needed to complete the calculation with more complex ones and I would like to get my beer before the sun goes nova and the universe collapses into a grey void. That is pity, for the brute force is the only way known to produce an optimal solution to a NP Hard problem, if it can be applied. Here it can obviously not be applied, so we need something else.

Our algorithm does not guarantee to find an optimal solution, but rather to find a local optimum, given a start value. If the start value is good, the local optimum it finds will be better. Given different start values, the algorithm will find different solutions and one of them is possibly the optimal solution of the problem. Unfortunately there are no known means one could use to tell which start value is better than any other.

[3] This kind of problem solving is known as *brute force*. Simply try all possible combinations and find the best one. It works well with two people and three tankards, but it fails to complete in less than 12 minutes on an average Compaq Presario with six people and thirty tankards. Given how many people fare in The Beer Garden in summer and how much they can drink, the time needed for brute force attack is not acceptable.

The Algorithm operates in two stages. In the first stage the algorithm calculates the start value which I believe is good and then it finds the local optimum in the second stage. The local optimum which the algorithm finds this way is good enough to be used in the bank. The program with the EALP engine produced results quite comparable to those produced by the bank's current allocation software, and it was far superior to the commercial solution from Reuters, in both the result quality and calculation speed.

2 Formulation

Now, let us give it some mathematical flare.

2.1 The Problem Table

I see the problem as some kind of matrix. Actually, it is not a matrix because you cannot operate on it like on any other matrix, you cannot calculate determinants and similar. It is basically a table. Somehing with few rows and few columns, just that the rows and columns have a certain influence on each other. I cannot think of a better name for the thing at this very moment and because I do not have to think of a better name, I shall call it the *problem table*.

The problem table has its dimensions. We let the input parameters determine the dimensions and thus the algorithm implementation can allocate a two-dimensional array to hold the problem table and work with array elements. This should make the algorithm fast, and indeed, EALP is fast.

Now, when the sugar momma brings the tankards to our table, we look at them and say few facts:

- Each distinct price involved in what we just see at the table has a column in the problem table asociated with it.
- Each person who sits at the tabe and awaits her tankards to hit her lips has a row in the problem table associated with her.

Any field where a column and a row of that kind intersect contains a number of tankards a particular person buys at a particular price. Everything clear so far?

I said that every person has her own row in the problem table. Note that there could be other rows not associated with anyone, but there may not be a person without a row. Similarily, I said that every price involved in our purchase has an own column. following the analogy to the rows, there may be a column which is not associated with a price, but all prices must have their own column. Now, if a field in the problem table lays at the intersection of a row associated with a person and a column associated with a price, then it contains the number of tankards that person buys at that price.

Let's say the number of people at the table is m and the number of prices involved in the purchase is n. We add an additional row and we put the sums of the columns in the corresponding fields of that row. Similarily, we add an additional column and we put the sums of the rows in the corresponding fields of that column. Having done that, we add yet another column and store the averages of the rows in its respective fields. Now the whole thing looks at least similar to what is on the figure 1.

	price 1	**price 2**	$\cdots$	**price n**	**r-sum**	**average**
pubber 1	$x(1,1)$	$x(1,2)$	$\cdots$	$x(1,n)$	$s_r(1)$	$a_r(1)$
pubber 2	$x(2,1)$	$x(2,2)$	$\cdots$	$x(2,n)$	$s_r(2)$	$a_r(2)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
pubber m	$x(m,1)$	$x(m,2)$	$\cdots$	$x(m,n)$	$s_r(m)$	$a_r(m)$
c-sum	$s_c(1)$	$s_c(2)$	$\cdots$	$s_c(n)$	s_{tot}	a_{tot}

We have m thirsty people sitting at the table, looking at us, with desire for beer in their eyes telling more than thousand words ever could. Each of first m rows represents one of them and these rows are marked $pubber1$ to $pubberm$. We said that tankards are delivered at different prices and that we have n different prices. Each of first n columns represents one of these prices and these columns are marked $price1$ to $pricen$. So far, so good.

We must assign each tankard to a person at the table, regardless if she going to drink the contents or not. If we keep the prices apart while doing it, we can define a function

$$\begin{aligned} lx(i,j) &= [\text{num of tankards for person } i \text{ at price } j.] \\ x(i,j) &\in \mathbb{N} \\ D(x) &= \{i,j \mid 1 \leq i \leq m, 1 \leq j \leq n\} \end{aligned} \tag{1}$$

Knowing what this function $x(i,j)$returns for each i,j within its definition area $D(x)$ is the goal we want to reach. You would do good to concentrate on the values returned by this function instead on how the function calculates those values. If you take the existence of this function too seriously, you might be surprised when you see that it does not exist at all. Here, we are simply assuming that there is a function we can use to discover how many tankards a particular pubber has paid a particular price for, but later we shall see that it is only an illusion, created to explain what follows.

Now, we do our allocation after the beer was purchased, so the prices are constants. We can define a set of prices:

$$P = \{price_j, price_{j+1}, ... \mid price_j \in \mathbb{R}^+, j \in \mathbb{N}, 1 \leq j \leq n\} \tag{2}$$

and a function:

$$s_r(k) = R[k] = \sum_{j=1}^{n} x(k,j) \tag{3}$$

which returns the k^{th} element of R. Every time I call this function, I get a positive natural number back, namely how many tankards the pubber k ordered. if you risk a glance at the problem table above, you shall see where this function is called. Since it is a sum of all $x(k,j)$ in its row k and I have no phantasy to give it a better name, it is called *r-sum*.

Let us take a glance at the last interresting bunch of constants we have. As we have said, a particular number of tankards has been purchased at a particular price. We can define a set of quantities per price:

$$C = \{qty_j, qty_{j+1}, ... \mid qty_j \in \mathbb{N}, j \in \mathbb{N}, 1 \leq j \leq n\} \tag{4}$$

and a function:

$$s_c(k) = C[k] = \sum_{i=1}^{m} x(i,k) \tag{5}$$

which returns the k^{th} element of C. Every time I call this function, I get a positive natural number back, namely how many tankards have been purchased at price k. If you risk a glance at the problem table above, you shall see where this function is called. Since it is a sum of all $x(i,k)$ in its column, it is called *c-sum*.

The value s_{tot} is the total number of tankards involved in the purchase. We define it as:

$$s_{tot} = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} x(i,j) \right) = \sum_{i=1}^{m} s_r(i) = \sum_{j=1}^{n} \left(\sum_{i=1}^{m} x(i,j) \right) = \sum_{j=1}^{n} s_c(j) \tag{6}$$

which is the sum of quantities per pubber, as well as the sum of quantities per price. What does that tell us? Eh? It tells us that we can allocate exactly as many tankards as have been purchased and that sounds quite logical to me. The sum of qualtities per pubber is the number of tankards those thirsty people at the table need. The sum of quantities per price is the number of tankards available for distribution. These two numbers must be as equal as numbers can only be.

Somewhere at the beginning I said we are interrested in average prices. Let us define a function:

$$a_r(k) = \frac{\sum_{j=1}^{n} (x(k,j) \cdot p(j))}{\sum_{j=1}^{n} x(k,j)} \tag{7}$$

which returns the average price a particular pubber k pays for her tankards. You can see where this function is called in the problem table above. We calculate the average for each pubber and we have a value:

$$a_{tot} = \frac{\sum_{j=1}^{n} (\sum_{i=1}^{m} (x(i,j) \cdot p(j)))}{\sum_{j=1}^{n} (\sum_{i=1}^{m} x(i,j))} \tag{8}$$

which represents the total average of all tankards at the table, regardless who drinks from them.

2.2 Equations

Let us take another glance at the equations we have so far:

$$s_r(k) = \sum_{j=1}^{n} x(k,j) \tag{9}$$

$$s_c(k) = \sum_{i=1}^{m} x(i,k)$$

$$s_{tot} = \sum_{i=1}^{m} s_r(i) = \sum_{j=1}^{n} s_c(j)$$

$$a_r(k) = \frac{\sum_{j=1}^{n} (x(k,j) \cdot p(j))}{s_r(k)}$$

$$a_{tot} = \frac{\sum_{j=1}^{n} (s_c(j) \cdot p(j))}{s_{tot}}$$

The s_r is a constant, so is the s_c. The values s_{tot} and a_{tot} are constant as well, for they can be calculated right at the beginning using constants and won't change anymore. What needs to be calculated are values $x(i,j)$ and they must be numbers which keep our constants constant and bring all $a_r(k)$ as close to each other as possible.

Now, I won't bother to prove it mathematically, but if the values $a_r(k)$ are close to each other, then they are close to a_{tot} as well. In the ideal solution, all values $a_r(k)$ would be the same and in that case they would be necessarily equal to a_{tot} and that's a fact. Thus, we can use a_{tot} as an orientation point and try to make each $a_r(k)$ as close to a_{tot} as we only can. While this is helpful, it does not permit us to make our life easier and calculate the solution on a row-per-row basis. Why not? Read on, and I shall enlighten you.

Another fact is, a change in any field $x(i,j)$ means also a change in three other fields. This is absolutely necessary, for the sums of the rows and the columns must be kept constant. Let's say you choose a value $x(a,b)$ in the row a and column b and change it by amount dx. Then you must choose another row c and another column d and change the values $x(c,b)$ and $x(a,d)$ by $-dx$, and the value $x(c,d)$ by dx. This change in four fields simultanously is what I call *flipping*.

One thing of importance is to see that values $p(k)$, $a_r(k)$ and a_{tot} need not belong to the natural number set, while all other values do. Said in the language of the implementation, $p(k)$, $a_r(k)$ and a_{tot} are floating point variables and everything else is an integer.

3 Run Stages

The algorithm runs in two stages. The first stage provides a good starting point for the optimisation, which is performed in the second stage.

The two stages are totaly independent of each other regarding the calculation and each one can be changed to run according to different rules without touching the other.

3.1 Stage One (Spreading)

The first stage, named *spreading*, starts with the zeroed problem table. All $x(i,j)$ are zero, as are all $a_r(k)$. Values $p(j)$, $s_r(i)$ and $s_c(j)$ are all given and we have calculated a_{tot} using the formula given above. Now we must take quantity s_{tot} and *spread* it across the problem table, forming one possible solution in which the constraints hold. This initial solution need not be optimal, but values $s_r(i)$ and $s_c(j)$ must give justice to their roles and really be sums of rows and columns. Having done that, we calculate the values $a_r(i)$ and hand over to the second stage.

How do we spread? Basically, we can do whatever we see fit. It is only important to find values $x(i,j)$, so that the sum of all $x(k,j)$ in any row k is reflected in $s_r(k)$ and the sum of all $x(i,k)$ in any column k is reflected in $s_c(k)$. One possible way is to use the following:

$$x(i,j) = Floor\left(\frac{s_c(j)}{s_{tot}} \cdot s_r(i)\right) \tag{10}$$

$$r(i,j) = \frac{s_c(j)}{s_{tot}} \cdot s_r(i) - Floor\left(\frac{s_c(j)}{s_{tot}} \cdot s_r(i)\right) \tag{11}$$

with $Floor()$ being a function which returns the largest natural number smaller than or equal to its argument. Is it clear that the sum

$$R_s = \sum_{i=1}^{m}\left(\sum_{j=1}^{n} r(i,j)\right) \tag{12}$$

must be a natural number? Is it clear that the equation

$$\sum_{i=1}^{m}\left(\sum_{j=1}^{n} x(i,j)\right) + \sum_{i=1}^{m}\left(\sum_{j=1}^{n} r(i,j)\right) = s_{tot} \tag{13}$$

must be true? Yes it is, it is as clear as the sky during the polar night when thousand stars glow all over it, it is as clear as the blue in the eyes of the Golden One.[4] The equation must hold, for the sum of all $r(i,j)$ is exactly the difference between s_{tot} and the sum of all $x(i,j)$; and the difference between two natural numbers, first one being the larger, can only be a natural number itself.

What is left to be done is to spread the quantity R_s across the problem table. For this, iterate through all rows and in each row k check if the value $s_r(k)$ really is the sum of all $x(k,j)$ in that row. If it is not, then find the column l where the sum of all $x(i,l)$ is not equal $s_c(l)$ and there must be such column. Having found it, add one to $x(k,l)$ and substract one from R_s. Repeat this until R_s drops to zero.

Are there to many facts in such a short time here? It is a fact that a row with a sum mismatch must exist as long as R_s is larger than zero. It is also a fact that a column with a sum mismatch must exist as long as such a row exists. Furthermore, it is a fact that if there is a sum mismatch, then the actual sum must be smaller than the required sum. If these facts are unclear, then go get

[4] Don't know the *Legend Of The Zlatković*? check `http://www.fh-frankfurt.de/~igor/tales/index.html`

yourself a mug of your favourite caffeinated beverage and stare at the formulae above until the fog collapses and these facts become clear.

When this is through, you will have one possible solution in the problem table. Now it is time to give that solution to the second stage and let the flipping optimize it further.

3.2 Stage Two (Flipping)

In this stage, we start with a problem table which came to existence in the first stage. We begin simple and sort for a bit. We sort the columns according to the corresponding price, smallest first. We end up with reordered columns whose corresponding prices increase, left to right. Now we sort the rows according to the value $a_r(i)$, smallest first. We end up with rows reordered so that values $a_r(1)$ to $a_r(m)$ increase, top to bottom. All sorting operations are done only on rows labeled pubber *something* and on the columns labeled price *something*, yes? Leave other rows and columns alone.

Now, there are facts. We know that a_{tot} is greater than $p(1)$ and also greater than $a_r(1)$. We also know that a_{tot} is smaller than $p(n)$ and also smaller than $a_r(m)$. We know this, because a_{tot} is the average of $p(j)$ and theoretically also the average of $a_r(i)$ and averages are always between the smallest and the greatest value they are made of. We can now find the greatest $p(j)$ which is smaller than or equal to a_{tot} and draw a vertical line which divides its column from the next column. We can find the greatest $a_r(i)$ which is smaller than or equal to a_tot as well and we can draw a horizontal line which divides its row from the next one.

	price 1	**price 2**	$\cdots$	**price n**	**r-sum**	**average**	
pubber 1	$x(1,1)$	$x(1,2)$	$\cdots$	$x(1,n)$	$s_r(1)$	$a_r(1)$	LBA
pubber 2	$x(2,1)$	$x(2,2)$	$\cdots$	$x(2,n)$	$s_r(2)$	$a_r(2)$	FFA
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	
pubber m	$x(m,1)$	$x(m,2)$	$\cdots$	$x(m,n)$	$s_r(m)$	$a_r(m)$	
c-sum	$s_c(1)$	$s_c(2)$	$\cdots$	$s_c(n)$	s_{tot}	a_{tot}	
	LBA	FFA					

We call the last column which has $p(j)$ smaller than or equal to a_{tot} LBA (Last Below Average) and the first column on the other side may be called FFA (First After Average). We have a similar situation with the rows. We ended up with a problem table divided into four quadrants.

Now we do some flipping. In order to make a flip, we need four fields which are at the intersection between two distinct rows and two distinct columns. Such fields always form a rectangle, when you look at them and imagine they were the corner points. We define such set of four fields as *flippable set*. If we now have a flippable set where each field lays in a distinct quadrant, then we have an *improvable flipping set*.

We define a *distance* between two rows as an absolute value of the difference between their $a_r(i)$.

$$d(i,j) = |a_r(i) - a_r(j)| \tag{14}$$

Now we must iterate through all possible improvable flipping sets. It's OK, they are not that many, actually, they are just a few. Having started, measure the distance between the two rows which build up the current improvable fliping set. Next, let dx be one and flip. After flipping, recalculate the averages $a_r(i)$ and compare the new distance with the one measured before the flip. If the distance improved, means if it got smaller, then you have made a good flip and here you break and start all over again. If the distance didn't improve, then flip back, restore the state as it was before the flip and continue with the next improvable flipping set. When you manage to iterate through all possible improvable flipping sets without making a good flip, you are done. The current state of the problem table is then the local optimum.

The whole process is short, because there are not that many improvable flipping sets. It would not harm to iterate through flippable sets instead, but it would do no good either. I shall not bother to prove it here, but if you flip a flippable set which is not an improvable flipping set, you are guaranteed not to make an improvement of the distance. This is why the flippable sets wherein no field shares a quadrant with any other field are called improvable.

4 Implementation

There is a fairly simple implementation in C programming language. You can download it from the following Locations:

- Official EALP page,
  `http://www.fh-frankfurt.de/~igor/projects/ealp/ealp.tar.gz`

If there is an implementation I know nothing about, and I know nothing about any implementation which does not appear here, please be so king to drop me a note about it. I would like to see, I would like to know, and I would like to include a reference to it here.