AdNetwork Project Report

Agent00

Elad Haviv
Moshe Krush
Guy Shaanan

16/05/2014

1. Introduction

This project extends the functionality of the Ad Network agent in the AdX game. The agent is designed to maximize the chances of winning the AdX contracts, and upon winning such, tries to gain maximum number of targeted impressions, thus maximizing our profit, revenue and reputation (quality score).

We have based our agent on two key components:

Campaign Strategy- which helps us to decide whether we should bid on an incoming campaign opportunity or not, and if so, how much should we bid on it.

Bid Bundle Strategy- Upon winning a campaign, how much to bid on an impression opportunity.

All components are details in the code architecture paragraph.

2. Code architecture

2.1 Campaign Strategy

Our main strategy was quite simple- don't compete against ourselves. More precisely- do not try to gain impressions from identical market segments in two different campaigns.

First, we check if the campaign is reachable by asking if we can reach the targeted impressions w.r.t the targeting segment, the campaign's duration and users' ad type preferences. We are using the mobile and video coefficients in our calculations to estimate the number of unique impressions we might reach. The answer (True/False) will affect our bidding strategy (see 2.1.1).

If we decided the campaign is reachable, we need to check it against our own campaigns.
Upon each incoming campaign opportunity, we iterate over all our currently running campaigns (with *dayEnd > today+2 and impsTogo > 0*) and find the overlapping market segments between the incoming campaign and the running campaign currently examined (see 2.1.2).
After finding the overlapping market segments (which could be empty), we decide if we should make a bid offer (see 2.1.3).

We then send our bid offer (alongside the UCS bid) and wait to an impression opportunity.

2.1.1 Is Campaign Reachable

Using User Population Probabilities tables found in the Game Spec v1.3 and our own Users class (see 2.2.1) we are calculating the population size of the incoming campaign.
We are also calculating the possible reach- how many impressions we can get from this target segment: the calculation uses the mobile and video coefficients which affects the number of impressions.

If we see that we can gain enough impressions per day (i.e. possibleReach > 1.2 * avgImpsPerDay) we decide this is a reachable campaign.

2.1.2 Finding Overlapping Segments

This phase is best explained by some study cases:

#1: The incoming campaign (IC) has a target segment (TS) of [FEMALE] and the currently examined running campaign (RC) has a TS of [MALE].
We can clearly see that the two campaigns have no overlapping TS, so we decide that it is safe for us to bid on the IC because there couldn't be impression opportunities matching both market segments. (but we do continue to look for other overlapping segments in our other currently running campaigns).

#2: IC-TS: [YOUNG, HIGH_INCOME]
RC-TS: [FEMALE]
In this case the overlapping TS is [FEMALE, YOUNG, HIGH_INCOME], because every impression for FEMALE is also an impression for YOUNG and/or HIGH_INCOME (and vise versa).

#3: IC-TS: [FEMALE, OLD, LOW_INCOME]
RC-TS: [FEMALE, YOUNG, LOW_INCOME]
In this case the overlapping TS is [FEMALE, LOW_INCOME].

The `calculateOverlappingSegments` function is a member of the `SegmentQuery` class.

2.1.3 Should we bid?

Whenever we find a non-empty overlapping segment, we update a boolean variable (which is initialized with the result of `isCampaignReachable` method) with the following formula:

```
shouldWeBid = shouldWeBid &&
     (incomingCmpImpressionsFirstFewDays < runningCmpImpsToGo);
```

Where:

`runningCmpImpsToGo`: how many impression are left for the current running campaign, and `incomingCmpImpressionsFirstFewDays`: how many impressions the IC should achieve in its first few days (first few days = `min(IC duration, RC days left)`).

If we see that the IC should achieve less impressions than the current RC in the next few days, we conclude that they will impose no competition on each other, and the `shouldWeBid` indicator will remain true.

If, however, this is not the case (i.e. `incomingCmpImpressionsFirstFewDays > runningCmpImpsToGo`), we might be in a competition with the RC, a case we would like to avoid, the indicator will be evaluated to false, and we will decide to avoid bidding on the IC campaign.

2.1.3.1 When `shouldWeBid == true`

In this case, we are going to bid on the IC. Our trials lead us to a simple formula which proved itself to be quite effective (we hope):

```
cmpBid = pendingCampaign.getReachImps() * factor / qualityRating;
```

Where:

`pendingCampaign.getReachImps()` - the reach level of the IC,

`qualityRating`- quality rating as extracted from the ADX reports,

`factor`- a constantly updating multiplicative factor, which helps us set the budget at a reasoning price that will lead to a high profit.

The `factor` is initialized with a number that was deduced from the games we performed against other groups and turned to be a good starting point: 0.1003125. (Against the dummy agents this number had to be quite bigger: 0.25).

The factor is updated (increased / decreased) in this way: if we won the last campaign we bid on, it means that our offer was the lowest, so we could try to increase the factor a bit, and thus increase our profit. If, however, we didn't win, our offer might have been too high[1], so we will decrease the factor a bit.

As always, some trial and error lead us to these coefficients:

```
if(wonLastCampagin){
    factor += (factor-0.1)/8;
}
else{
    factor -= (factor-0.1)/4;
}
```

2.1.3.2 When `shouldWeBid == false`

In this case, we've found overlapping market segments between the IC and one of our currently running campaigns, and we decide to avoid bidding. We do that not by skipping sending the bid offer, but by setting the bid budget to the as high as possible:

```
cmpBid = 0.99 * pendingCampaign.getReachImps() * qualityRating
```

There are two reasons for doing this:

- There will probably be at least one agent to offer a lower budget than ours, so with a high probability we will not win this campaign.
- If we do win, we will put all our efforts to win impression opportunities for this campaign, resulting in a high profit (we saw this is proving itself to be a good approach).

[1] We are taking the risk that our offer was indeed the lowest, but some other agent was randomly selected as the winner.

2.1.4 Campaign Strategy Workflow

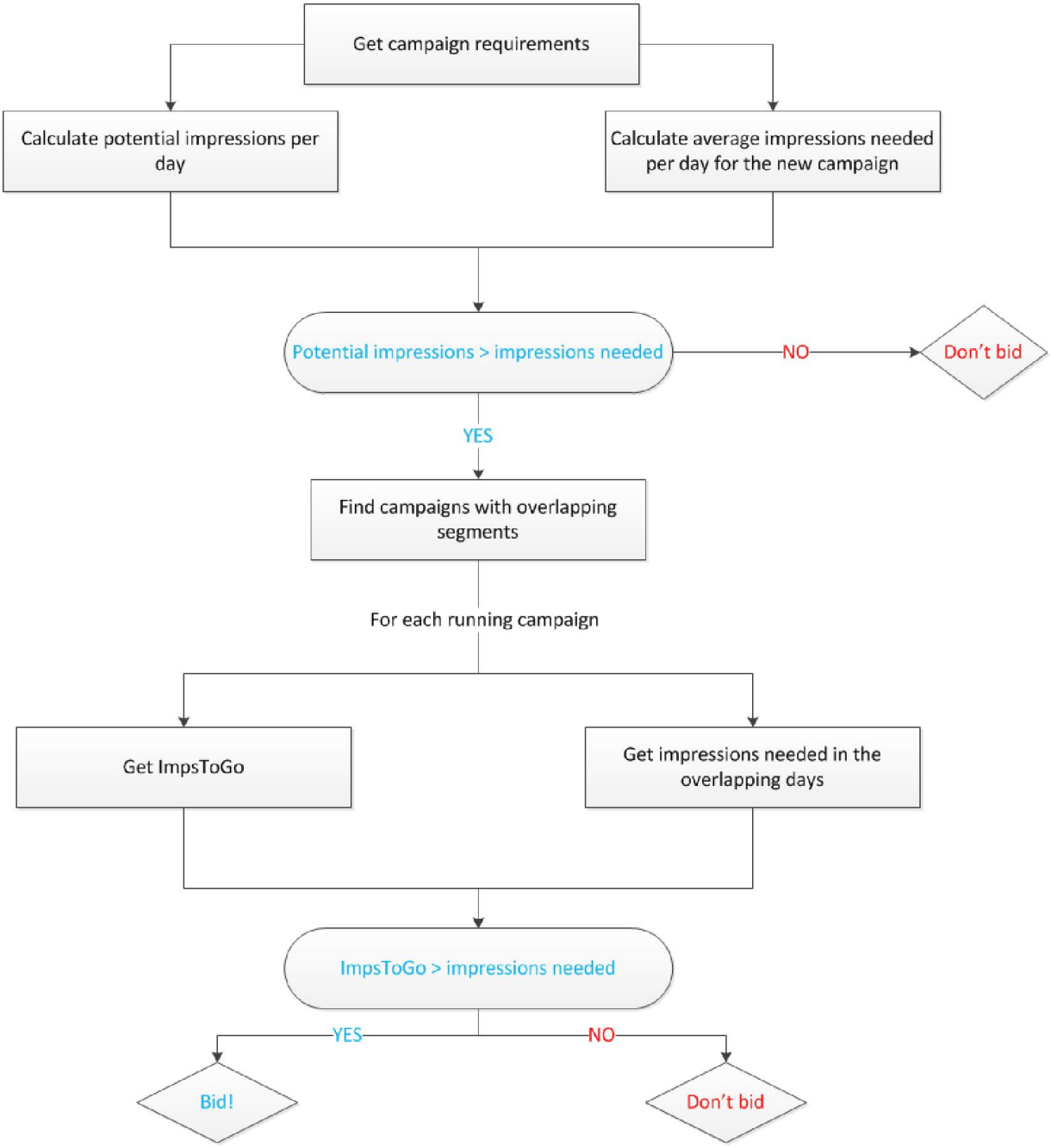


Diagram 1

2.1.5 Open Issues

It seems that when updating our factor, we sometimes could not reach the minimal reserve campaign budget because we have set it too low. In this case we are trying to increase it until we are back in the allowed range. (`fixOffer` method). As our calculations are initially made w.r.t this range, this behavior remains an open issue for us.

2.2 Impression Opportunities Bidding / Bid Bundle Strategy

This part consist of several key elements:

- Users class- uses stored data like users population/preferences (as described in Table2, Table 3 on the Game Specification Document v1.3) to quickly execute data queries (like counting the potential impressions for a given set of market segments). This class is being updated with user ad-type preferences during the game (collected from the ADX publishers reports) as those aren't available as a given statistics. (See section 2.2.1).

- "LoadBalancer"- a bar-like model that stores information on all running campaigns- the campaign data (including campaign characteristics such as duration, reach, etc), the budget (for campaigns won by other agents- we save our suggested budget) and the winning agent. With each day, we are adjusting the bars according to new and old campaigns. (See section 2.2.2).

- "Neuro"- a multi-dimensional matrix model that is used to learn the demand of each market-segment against a set of opponents (other agents). (See section 2.2.3).

- Bid bundle calculator- using information from the LoadBalancer, Users and Neuro classes, calculates the bid bundle offer for the impression opportunity. (See section 2.2.4).

- Querying mechanism (See section 2.2.5).

2.2.1 Users

This class stores the users population as can be found in Table 2, Table 3 in the Game Spec v1.3. Upon game start, we call initialize() method to refine the tables, due to the fact that only a portion of the given publishers participates in each game, and we want to reserve the ratios for users preferences and priorities.

Methods in this class:

```
generatePriorities(Set<MarketSegment> segments):
```

Given a set of market segments, returns a list of entries with specific user/device/ad-type/website and potential impressions count of the users belonging to that set.

`updateAdTypePreferneces()`:
Update the ad type preferences (text / video) for each website (publisher). This helps us to refine our bid bundle offer.

`countImpressions(Set<MarketSegment> segments, double mobileCoef, double videoCoef)`:
Count how many impressions we can get from the given segments, w.r.t mobile and video coefficients.

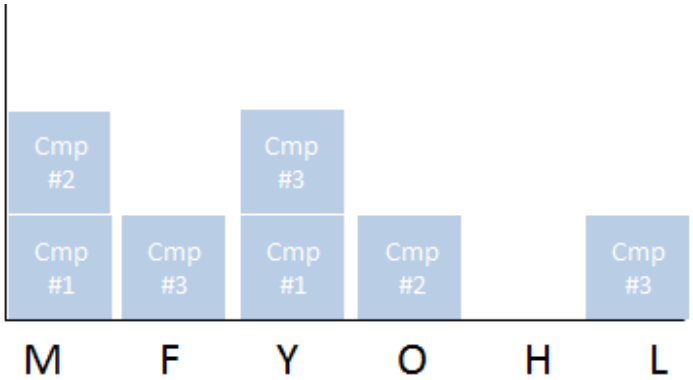
`countSegments(Set<MarketSegment> segments)`:
Return the population size of the given segments.

2.2.2 LoadBalancer
Imagine a graph, which its x-axis is compound of the 6 market segments (M, F, Y, O, H, L), and the y-axis is a list of all the campaigns that are targeted at this market segment

Each list element is of type `CampaignSegmentTracker` and stores:

- segment: The market segment of the campaign (the same as the x-axis)
- daysLeft: How many days left for that campaign
- avgDailyBudget: The average daily budget
- agent: The winning agent ("self" means our agent)
- campaign: `CampaignData` object

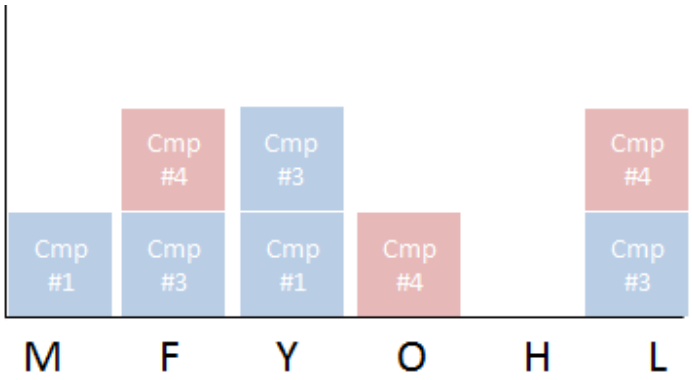
For example: lets say we have 3 campaigns:
Cmp#1: days: 1-10, [M Y], avrgDailyBudget = 10
Cmp#2: days: 3-6, [M O], avrgDailyBudget = 20
Cmp#3: days 4-7, [F Y L], avrgDailyBudget = 5
On day 5, the graph would look like that:



Graph 1

This way we can keep track of the market segments being addressed in the running campaigns.

On day 6, when a new campaign (Cmp#4) is introduced, and Cmp#2 ends, the LoadBalancer will be updated accordingly through an "`advanceDay()`" method:



Graph 2

Note: if other agent than us has won the campaign, we can't tell how much it paid, but we can almost guarantee it paid less than us. So we set the budget (and the average daily budget) to be our budget- we assume the other agent has paid less than us, thus resulting in an average daily budget lower than us. When we will attempt to bid on this bid bundle, we will use the average daily budget value, which should be greater than the opponent's one, and will help us win this bid bundle.

2.2.3 Neuro

A set of 2d matrices for each market segment possibility (6 matrices for 'age', 4 matrices for 'income', 2 matrices for 'gender' etc). The rows/columns represent different opponents (agents). For example, the cells in row 2 and column 2 define the behaviour of opponent 2 with respect to the other opponents (and itself) on a given market segment.

Thanks to symmetry, we're using only the cells that are on the lower triangle of the matrix

Upon an update request, we go over an AdNetworkReport updating the relevant matrices (according to the entries keys) such that on each matrix the relevant opponents (those that are registered in the LoadBalancer as playing the relevant market segment) cells are being updated with scores according to a success/failure to get a "good" impression ratio.

Upon a test request, the relevant matrix (according to a bid-bundle entry key) is being queried for the maximum score on the cells of the current opponents (given by the LoadBalancer). The return value is a scaling factor which is bounded to [0.75,1.25] (using arctan) that represent the demand on this particular entry. High demand will increase the price we'll pay and vice versa.

For example, in a game of 5 opponents, suppose that we failed to reach a good ratio on a specific entry while playing against opponent 2 (in this segment). We need to increase the demand value on its row/column:

2	2			
	2			
	2			
	2			

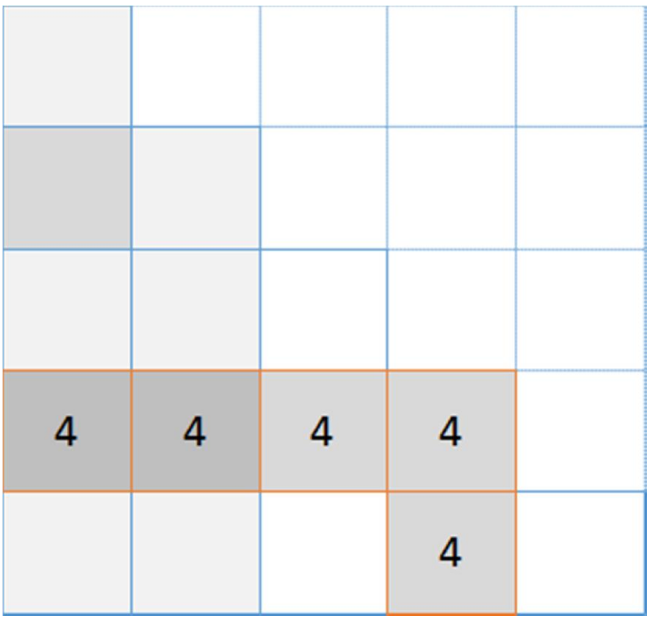
Graph 3

Then, lets say both opponents 1 and 4 are competing against us (with success) in the same entry:

1				
1				
1				
1/4	4	4	4	
1			4	

Graph 4

And finally, opponent 4 alone (with success against us):



Graph 5

Those gradients are now representing the demand of this specific entry (user/ device/ ad-type/ website) so that now we can learn to put more money when playing against the relevant (the max valued cell corresponds) opponents.

Note: there's no decreasing learning factor because the changes are taking place during the whole game (there's no good way to separate a learning phase and a testing phase).

2.2.4 Bid Bundle Calculator

We are using the information from the LoadBalancer, Users and Neuro classes in our bid bundle calculation process, as follows:

Upon an impression opportunity, we are querying the LoadBalancer by going over all the relevant market segments for the running campaign (RC) and selecting the maximum value of the average daily budget from all "blocks".

Looking at Graph 1, on day 5 with RC = Cmp#1 (i.e. market segment [M Y]), our query will return "20" because Cmp#2 has the highest average daily budget out of all the campaigns with [M] or [Y] as targeted market segments, and we want to offer the maximum we can (while maintaining our budget) in order to win this bid bundle.

Now we need to adjust our bid to match the characteristics of the target segments (i.e. a potential impressions count w.r.t user/ device/ ad-type/ website preferences), and take into account the video and mobile coefficients.

From the Users class, we are fetching the entries of user/ device/ ad-type/ website combinations (each of those includes the potential impressions) corresponding to the RC target segment (`generatePriorities` method). From each entry, we are building the AdxQuery object, and adjusting our bid: we take a "gamma correction" of the following numbers, multiplied by each other:

- amount of impressions we can get from this market segment
- mobile coefficient
- video coefficient
- scaling factor which is given by Neuro (representing the demand on this entry).

```
rbid_adjusted = rbid * Math.pow(
     impressionsRatio *
     campaign.mobileCoef() *
     campaign.videoCoef() *
     Neuro.test(...)
     ,
     1.0/8.0 )
```

Note: the full formula is found in the code.

We then set the daily impression limit to be 125% of the target, and the daily budget limit to be 125% of the average impression cost (multiplied by the number of impressions to go), and send the bid bundle.

2.2.5 Querying mechanism

The adx.query package (alongside the interfaces and Query class) implements a SQL-like querying mechanism, ment for easily querying our data models.
Influenced from C#'s LINQ, our mechanism is a big finite state machine with SQL-like syntax which helps us query lists / arrays (any Iterable items) by choosing fields/indices, imposing constraints, applying aggregate functions (like sum, count, max…) etc.

As we explained in the bid bundle calculator section, we are looking for the max average daily budget value in our LoadBalancer. We can retrieve it very easily with this mechanism:

```
LoadBalancer.getInstance().get(campaign.targetSegment)
    .select().property("avgDailyBudget").max()
    .exec()
```

The `get(campaign.targetSegment)` returns a `list<CampaignSegmentTracker>`, and we are querying that list.

2.2.6 Bid Bundle Strategy workflow

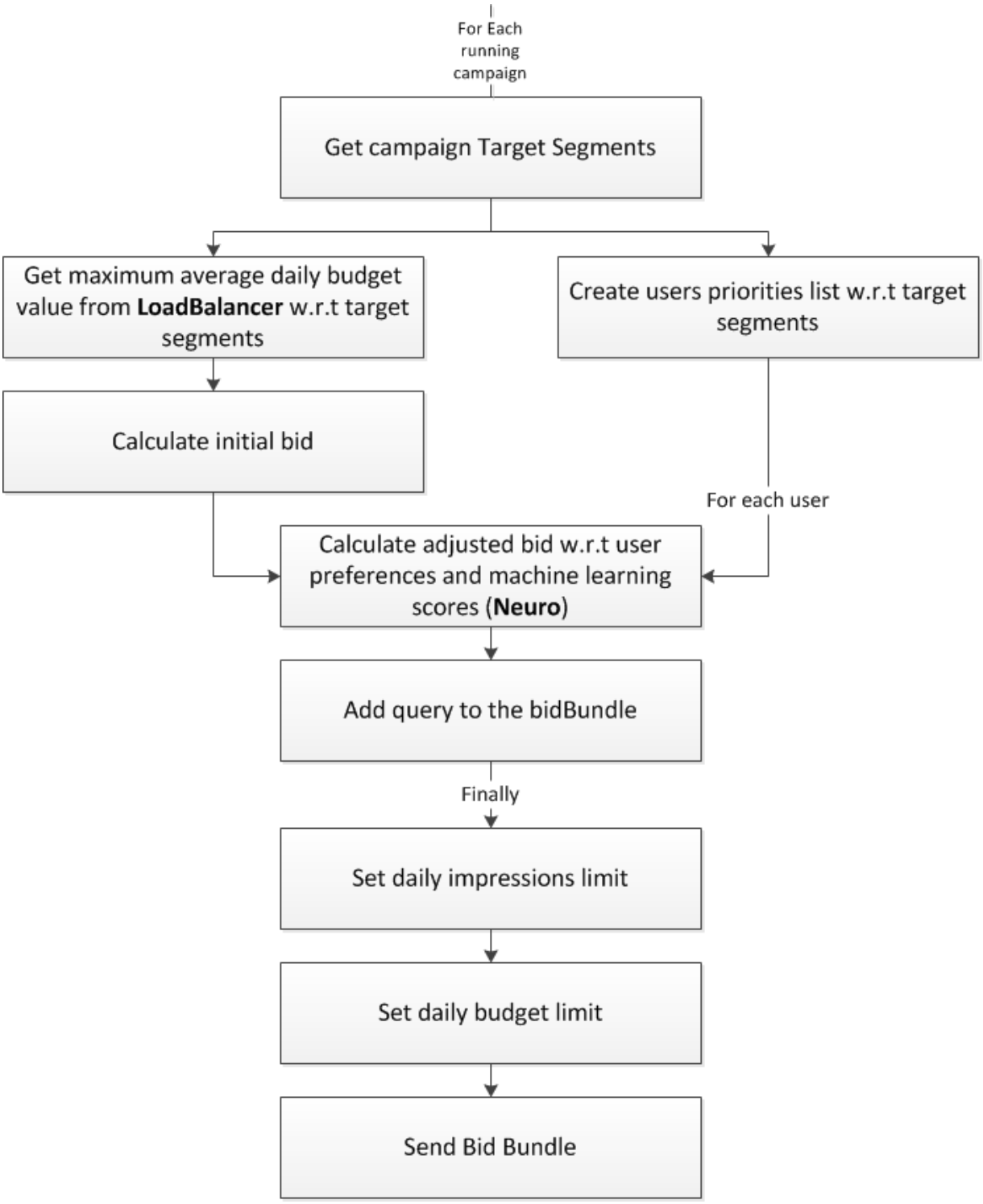


Diagram 2

3. Simulation Results

We performed this simulation on 06/05/2014 10:31, #974.
Its results:

Agent	Quality	Profit	Revenue	Adx cost	UCS cost	Impressions
anl	0.43	1.24	13.99	5.64	7.10	6699
agent00	1.25	76.01	95.75	15.76	3.97	73047
WinnieTheBot	0.19	-0.05	0.17	1.4E-6	0.21	2051
blue	0.04	-0.29	0.85	7.5E-10	1.13	3345
giza	0.18	-2.20	1.11	1.64	1.66	5746
tau	0.14	-7.29	7.58	9.04	5.82	4332
Agent2	1.10	22.14	32.74	3.62	6.96	101281
DummyAdnetwork	0.73	22.26	37.82	10.52	5.02	28770

Lets see how our strategies worked:

3.1 Campaign Strategy

On day 7, there is an incoming campaign with the following features:

```
Campaign opportunity - Campaign ID 548356046: day 8 to 17  HIGH_INCOME OLD,
reach: 6075 coefs: (v=2.2392104142292575, m=1.757121482518556)
```

The log below shows our calculations to whether we will bid on this campaign or not:

```
06/05 07:32:59 | Running cmp segment: [MALE, HIGH_INCOME, OLD]
06/05 07:32:59 | Pending cmp segment: [HIGH_INCOME, OLD]
06/05 07:32:59 | Found overlapping segments: [MALE, HIGH_INCOME, OLD]
06/05 07:32:59 |        runningCmpLeftDays = 4
06/05 07:32:59 |        runningCmpImpsTogo = 808.0
06/05 07:32:59 |        incomingCmpImpressionsFirstFewDays = 2430.0
06/05 07:32:59 | AVOID BIDDING!
```

The TS of the current RC inspected is [MALE, HIGH_INCOME, OLD] and the TS of the IC is [HIGH_INCOME, OLD]. So we can clearly see that both campaigns have overlapping TS, so we are trying to understand if it “smart” for us to bid on this IC.

We see that the RC has 4 days left, and it needs to reach 808 more impressions.
In this amount of time (4 days) the IC should reach a total of 2430 impressions (it is 10 days long, so it needs an average of 6075/10 = 607.5 impression per day, thus 2430 in its first 4 days).

As we explained in section 2.1.3, we are avoiding bidding this campaign, and setting the bid as explained in section 2.1.3.2.

Another example can be shown on day 9, with this IC:

```
Campaign opportunity - Campaign ID 397649448: day 11 to 13  OLD, reach: 3246 coefs:
(v=2.0295917562525796, m=2.0978535908926235)
```

Going over all our running campaigns, we found a RC with the same TS:

```
06/05 07:33:30 | Running cmp segment: [OLD]
06/05 07:33:30 | Pending cmp segment: [OLD]
06/05 07:33:30 | Found overlapping segments: [OLD]
06/05 07:33:30 |        overlappingSegmentSize = 5411
06/05 07:33:30 |        runningCmpLeftDays = 10
06/05 07:33:30 |        runningCmpImpsTogo = 43288.0
06/05 07:33:30 |        incomingCmpImpressionsFirstFewDays = 3246.0
```

In this case, the IC should reach far less impression than the currently examined RC in the same amount of days (and there are no other RCs) so we decide to not avoid bidding, and we are setting the bid as explained in section 2.2.2.1.

3.2 BidBundle Strategy

This strategy results are best viewed from the adNetowrkReport received after sending the bid bundle offers:

```
06/05 07:34:20 | [ 170939051 [ male   45-54 high+  ] bestbuy mobile text   3  3  0.583894 ]
06/05 07:34:20 | [ 170939051 [ male   45-54 high+  ] cnet      mobile text  17 17  1.326653 ]
```

We can see two identical segments, in the same campaign, with the same preferences (i.e. mobile and text), were we got all the impressions we bid for, yet the bid offer (the rbid_adjusted) is quite different:

- 0.583894 / 3 = 0.1946313
- 1.326653 / 17 = 0.0780

This happens thanks to the knowledge we have on user preferences on different websites and the target demand we learned during the game (Neuro).

In another simulation, it's clear that an average impression cost in 'agent00' (with Neuro) is much lower than in the others (without Neuro):

Advertiser Information

Agent	Quality R...	Profit	revenue	adx cost	ucs cost	impressi...
agent00	0.17752782...	17.78	34.02	16.2310311...	0.0	36477
old_1	0.99403392...	-14.03	19.26	26.0190620...	7.26744307...	18222
old_2	0.76553085...	-5.97	14.31	12.3029413...	7.97582214...	12137
old_3	0.72537570...	-5.45	20.60	18.4385733...	7.61820220...	18462

4. Open Issues

- UCS bidding makes an implementation similar to hedging a stock price. Min and max bidding values are being updated according to success/failure to reach a target score and the actual bid plays its role in between.
  However, we couldn't find any rationality regarding the UCS score. This led us to ignore the suggested strategy with a low (constant) bidding offer.