THE AD EXCHANGE GAME

TOP3- Agent Implementation

Noam DvirYuval, Zuaretz, Eyal Agiv
[Email address]

Introduction:

In this workshop we were introduced to an Ad Exchange platform that simulates real-life internet ads market consisting of users, websites, advertisers and Ad Networks. Our goal was to implement an Ad-Network agent that will win campaigns and fulfill them well, thus increasing the agents revenue and maximizing profit.
This document details our work process and gives examples of competitions we participated in, backed by results.
First we will describe our agent's strategy, then elaborate on the modules and function that were implemented and finally show result of running our agent against other agents.

Strategy:

In the following section we will describe our strategy we used. Our strategy was derived out of several main guidelines from which we designed each of our agent's modules:

1.Efficiency:
In order to maximize profit, we wish to recognize and win campaigns with the highest potential while minimizing our expenses at times our agent is less active (have fewer impressions till completion of current campaigns) and avoid aggressive competitions against other agents (unless necessary) that could lead to high expenses and smaller profit.

2.Maintaing high score:
In order to win campaigns, maintaining a high quality rating is essential. Therefore, we had to take into consideration the amount of active campaigns and our chances of fulfilling before bidding on new campaign opportunities. Having more campaigns than we could handle might lead to lower quality score, which can be problematic for

our agent. On the other hand, winning campaigns is the only way to gain profit so we tried to win as many campaigns as long as we don't “bite more than we can chew”.

3.Utilizing Randomness:

A large portion of all campaigns are allocated randomly. Therefore, we chose to bid the possible maximum on some of the campaigns. Campaigns won randomly at maximum bid have higher completion potential. Such campaigns are a good way to improve quality score in case it is too low. From simulations analysis we concluded that bidding maximum on a larger percentage of all campaign opportunities improves the average quality score of our agent and help us get it high again when too low.

UCS Bidder Strategy:

In general we wanted the UCS level to be somewhere in the half and up- so the const and the calculation try to achieve that. The reason we want to have high UCS bid are because this can save us a lot of trouble in the QS, and second we have more flexibility in the impression bid auction phase.

An other observation is that we want UCS only when it is needed- so here we will try and bid only when necessary. This means that when we have no active campaigns the bid will be 0.

Next we will talk about how we calculate the bid.

In a few pages ahead you have a detailed explanation about getSmartReachIndex()- this function goal it to compute how much do we need the UCS based on the current active campaign we have- this means that a campaign the is ending the next day which still have impression to collect- will influence the desired UCS level of this day. On the other hand, campaign that have more days will have smaller influence on the UCS bid level of today.

The other 2 factors are bidLevelOffset and bidBidOffset.

The first handled situations where the result of last day auction. In case we got a hit (we got the level we wanted) we know our bid is ok for the current game state. If not, we will adjust the bid with factors of 1.3 and 0.7.

bidBidOffset handled situations where the current wanted level is different from the last one- so we have to increase (factor of 1.3)/decrease (factor of 0.7) our last bid accordingly.

Campaign Bidder Strategy:

There are 2 main indexes that control the bid for a specific campaign, the multiplication of the 2 is refferes as score in the code.

The first one is segmentRatio. This index try to determain how hard will it be to will impressions on the campaign segment. To do that we look at all the currently active campaign and look at the segment the contains. The higher the index is- the harder it will be to win an impression

The second is competiveFactor. In order to win the game we must win campaign opportunities.

While we do our best in order to calaulate the best bid sometime this bid is not eanogh. And this is where this index come for the help. In general- the higher this index is- the harder it is for us to win a campaign: when we win a campaign in increase this index, so next time if we win an auction, while it will be harder for use to finish all the current campaign- the profit from this campaign is high. On the other hand, when we loose the auction, we decrease this index so next time our offer will be more competitive with respect to the market. This is how we keep track with the market.

Next there are different cases which we will present now:

From previous games we saw that when our offer is not very very competitive we will most likely to loose. On the other hand, the chance to win a random campaign is very high- because of this, when we don't have a very attractive offer- we will offer the mac bid (with some exceptions)

1. When out QS is very low (qsRedZone), the chance we win a campaign that is not random is very low. So we just hope to win the random and yhus we bid maximux
2. When QS is before the red zone (qsGrayZone)- we can still save the game- and to do that we need to finish campaign- so we offer the min bid
3. If the segmentRatio of the current campaign is very high- it is most likely we will loose money in order to finish it- or even we wont finish it at all- so we

bud the max and hope to win it as random campaign.

4. If our bid is not competitive eanoght- bid the maximum as described above.
5. Else bid competiveFactor*segmentRatio as describes above

Bid-Bundle Strategy:

The following segment describes our agent's decision-making process while bidding on impressions for a certain campaign.

We distinguish between several cases for a certain campaign at any given moment in time that might require a “special treat” when bidding on impressions :

-short campaign: as described in the 'Campaign Bidder Strategy' section, we often choose to avoid short campaigns with low profit potential, for they might interfere with completion of bigger, more important campaigns. However, if we did decide to engage in such campaigns, we keep in mind that the first day of a short campaign is critical. Poor performance on a first day of a short campaign might be problematic and could lead to unsuccessful fulfillment of the campaign which results in bad quality rating.

-“Difficult” Campaigns: campaigns that have value of “Campaign Difficulty” > 0.8 are considered difficult. In such campaigns, our agent has to make an effort to reach a large amount of impressions from the first day in order to assure completion. When bidding on impressions for a difficult campaign, our agent will take a more aggressive approach.

-Game Start: In the beginning of the competition we treat active campaigns differently from the rest of the game. Preforming badly on the first days of the competition might cause bad quality rating (not a good way to start the game). It is known that rival agents take an aggressive approach at the early stages of the competition so our agent got to line up and bid more aggressively as well. Therefore, we consider the first 5 days of the game as critical.

When given an impression opportunity, we update our active campaigns for the appropriate segment market to increased segmentRatio value. Also, we calculate the percentage of the campaign's remaining budget we wish to invest on buying impressions. We base that decision on the progress rate of impressions reach we got (imps reached per day) for a specific campaign and comparing that to the rate of impressions we need to keep in order to fulfill the campaign (imps left per day). If a campaign's reach progress is bad, our agent will allocate a larger portion of the budget for getting impressions. A larger portion of the budget will be allocated according to how bad the campaign's reach progress really is and whether or not we are currently in one of the aforementioned “special cases”. On the other hand, if the campaign's reach progress is good, we will choose to spend less money from the budget on impressions (which means our agent will have more money to invest in improving and maintaining a decent UCS rating).

Design Spec:

In the next part of the document we will described the different modules we used when implementing our agent and the relations between them. Every module has a specific role and comprises the logic our agent use when making decisions on different aspects of the game.

At each day of the competition, our agent is making 3 types of bids: UCS bid, campaign bid, and impressions bid. For each bid, we developed a class designed to analyze the current state of the game and make the optimal decision based on that.

UCS_Bidder.java: This class is responsible of deciding how much to bid on the next UCS bid based on the last bid our agent made and the rating it got (as described before).

CampaignBidder.java: Given a campaign opportunity, this class determines how much to bid on that campaign (as described before).

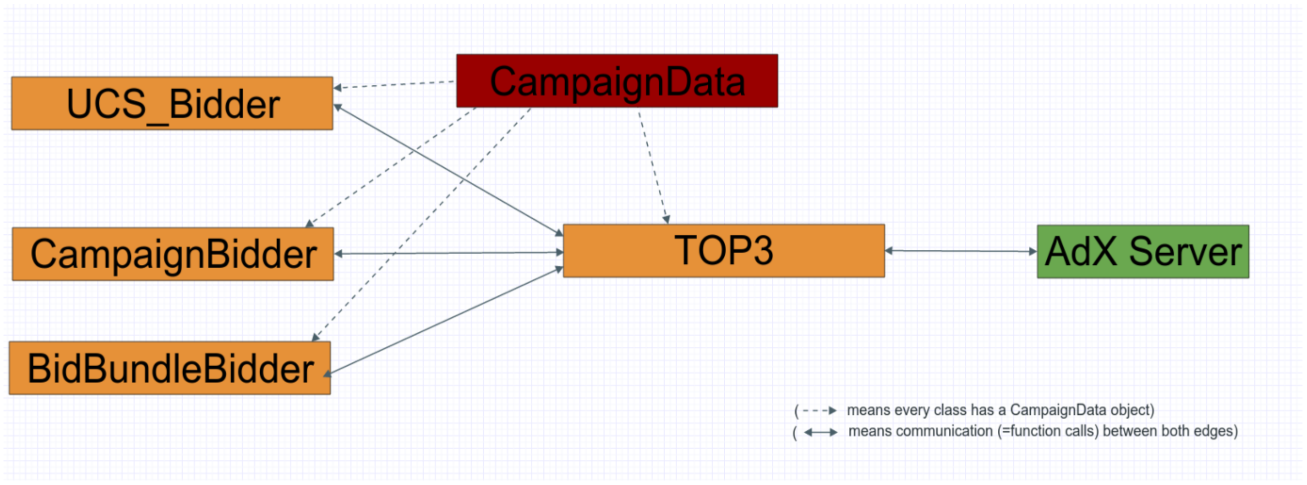
BidBundle.java: For any impression opportunity, this class's purpose is to identify the impressions from a market segment we need for one (or more) of our active campaigns and to calculate the amount of money we wish to bid on that impression, based on the game's current state (as described before).

Other than the classes listed above, our agent consist of two other classes:

CampaignData.java: This class defines an object of campaign. It saves different parameters about a certain campaign and uses it in the decision making mechanism of our agent.

TOP3.java: This is an “improved” version of the basic agent class we were given at the beginning of the workshop. In some way, TOP3.java is the brain (or heart?) of the agent. Other than maintaining a game-stats database, it performs function calls to all the other classes, provides them with the information received from the server, and acts according to the answer it gets back. It is the only class that “communicates” with the server and is responsible for initializing global variables at the beginning of the game.

The following diagram describes overall work flow of the agent and the relations between the different classes:


CampaignData
UCS_Bidder
CampaignBidder
TOP3
AdX Server
BidBundleBidder
(- - -▸ means every class has a CampaignData object)
(◂—▸ means communication (=function calls) between both edges)

UCS_Bidder:

UCS_Bidder's role is to calculate the amount of money we wish to bid on UCS, using the last bid we made and the result of this auction, as well as information from all of the agent's active campaigns.

This class consists of one main function and a few other aux. we will mention some of them:

```
public static double calcUCSBid(Map<Integer, CampaignData> myActiveCampaigns, long
day)
```

Input is all the currently active campaigns and the day we are at.

Its initiate the start value of the class parameters.

In case we don't have campaign it save the agent money by bidding 0 for current UCS.

Output is the final bid to send the server.

```
public static double calcBidLevelOffset()
```

as mentioned before, our agent take into account if we succedded in out last bid. If we got a lower level then wanted- we need to offer more, if we got higher- we need to offer less. This data is passed to *bidLevelOffset*.

`public static double calcBidBidOffset()`

if last targeted level was lower than current, we obviously need to offer more, and it is the same for the opposite side. This data is passed to `bidBidOffset`

```
public static double getSmartReachIndex(Map<Integer, CampaignData> currentCampaigns,
long day)
```

this function goal is to estimate how much do we need the UCS bid to be high.
It does so based on the current campaigns.
For each campaign we calculate the `reachIndex`, and all these are summed into `smartReachIndex`:

$$smartReachIndex = \frac{\sum_{myActiveCampign} \frac{impressionLeft}{\sqrt[4]{daysTillEnd}}}{totalImpressionLeft}$$

the reason behind the sqrt is for keeping the QS high we need to win impressions, so this will give us higher UCS level.
And as a result we get $0 \leq smartReachIndex \leq 1$.

CampaignBidder:

```
public static long getCampaignBid(CampaignData campaign, Map<Integer,CampaignData>
allActiveCampaign, int day,CampaignData lastCampaign, double qs)
```

this is the main function of the class and its role is to get all the indexes for other function in the class, and pass the final campaign bid to be send to the server.
As input it gets all the data it need

`public static double setStartingScore(CampaignData campaign)`

this function is called only in day 0- and its role is to give the first campaign bid.
Because we don’t have any data to work with in this point, it will look only at the pending campaign.
The result will be

$$score = \frac{reach}{length * segmentSize}$$

```
private static double setScore(CampaignData campaign, CampaignData lastCampaign,
double qs, int day, Map<Integer,CampaignData> activeCampaigns)
```

this function will compute the score for each day (except 0). Score will be the prive for a single imp. It does so with the help of the following parameters:

`competiveFactor`- which segment on how much we want to win next campaign

`segmentRatio`- what are the chances of finishing this campaign.

it also take into account what is the current QS and act accordingly.

Score will be either the maximum bid, the minimum bid or $competiveFactor * segmentRatio$

```
private static double getSegmentRatio(CampaignData campaign,
Map<Integer,CampaignData> activeCampaigns)
```

for each segment in the campaign we check how hard and busy it is.

The result is

$$segmentRatio = \frac{\sum_{on\ all\ segments} asicSegmentSize\ *\ impBySegmentInDay}{totalSegmentSize * CampaignLength}$$

* $impBySegmentInDay$ will be explain shortly

higher result indicate that the campaign is harder to finish

```
public static double getSegmentRatioByDay(Set<MarketSegment> basicMarketSegment,
long day, Map<Integer,CampaignData> activeCampaigns)
```

this function calculate how much a basic segment is hard on a specific day, this is done by the calculate:

$$segmentRatio = \sum_{overAllCampaigns} \frac{campaignReach}{campaignSegmentSize * campaignLength}$$

this number is used by `getSegmentRatio` to calculate the hardness of a segment on a specific day

BidBundleBidder:

This class is desinged to calculate the agent's bid on every impression opportunity given a bid-bundle. It has one main function that is called from TOP3.java class and one other auxiliary function that is called from the main function.

```
public static AdxBidBundle sendBidAndAds(Map<Integer, CampaignData> myCampaigns,
int day,Map<Integer,CampaignData> activeCampaigns)
```

Input: (1) `Map<Integer,CampaignData> myCampaigns`: list of all of the agent's currently active campaigns represented as an array of `CampaignData` objects.

(2) `int day`: current day of simulation

(3)`Map<Integer,CampaignData> activeCampaigns`: list of all of the game's currently active campaigns represented as an array of `CampaignData` objects.

Output: `AdxBidBundle bidBundle`: our agent's bids on every impression opportunity in a given bid-bundle.

Functionality: For every impression opportunity, calculates the amount of money we wish to bid based on the agent's logic described in the 'Bid Bundle Strategy' section.

```
public static boolean isCampaignActive(CampaignData campaign,int day)
```

Input: (1) `CampaignData campaign`: specific campaign.

(2) `int day`: current day of simulation

Output: `boolean`

Functionality: return 'true' iff input campaign is active.

CampaignData:

We used the original class that was given to us at the beginning of the workshop. However, we choose to add some class fields. We will now describe those fields and their purpose.

`double campaignDifficulty`: Evaluate how hard would it be to finish the campaign (get the required impressions within the campaign life span).

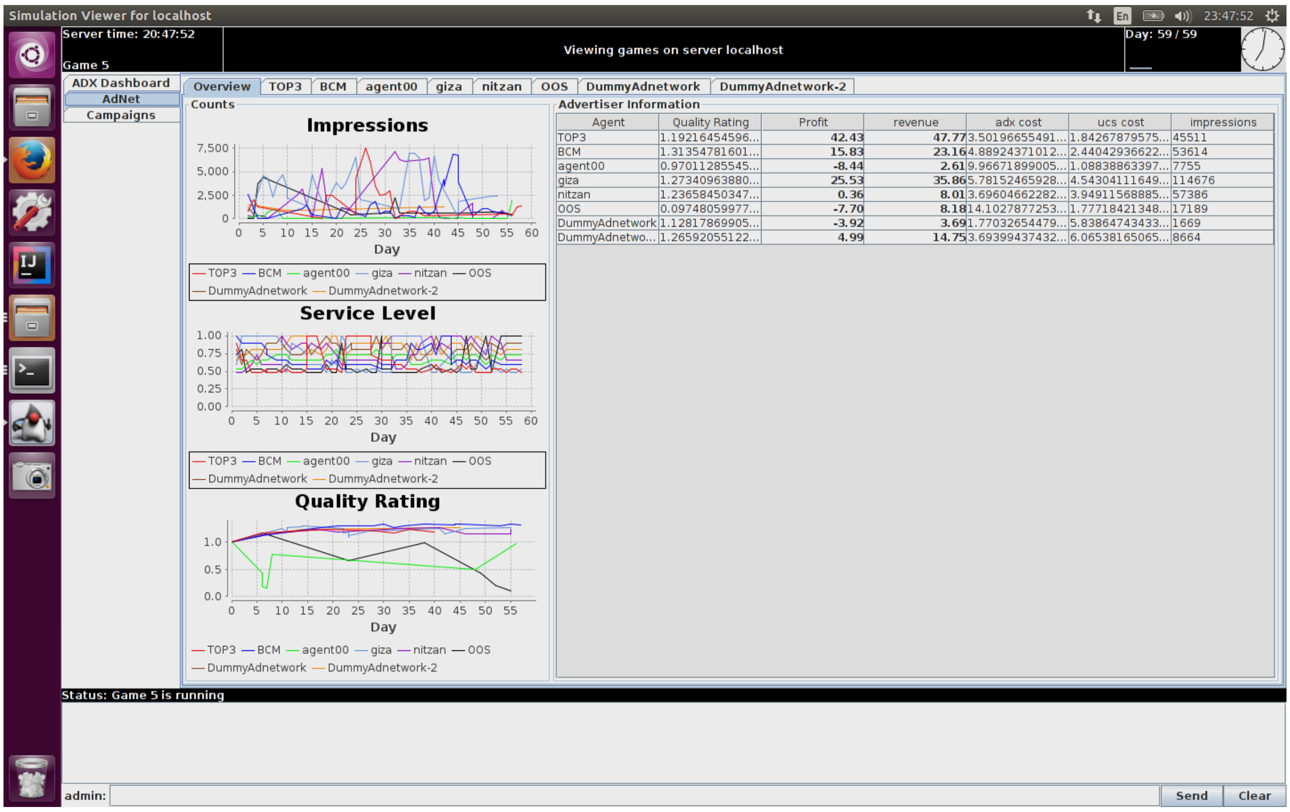
`double remainingBudget`: the amount of money left in the campaign's budget.

`double impressionRatio`: rate of impressions reach (imps per day) we need to maintain on average in order to successfully fulfill the campaign.

`double currentRatio`: the current impression reach rate our agent got at a certain campaign so far. (impressions reached / # of days since beginning of the campaign)

`double ratioTillfinish`: impressions left to reach / days left of the campaign

Simulation Run:



Server time: 20:47:52

Game 5

Viewing games on server localhost

Day: 59 / 59

Advertiser Information

Agent	Quality Rating	Profit	revenue	adx cost	ucs cost	impressions
TOP3	1.19216454596...	42.43	47.77	3.50196655491...	1.84267879575...	45511
BCM	1.31354781601...	15.83	23.16	4.88924371012...	2.44042936622...	53614
agent00	0.97011285545...	-8.44	2.61	9.96671899005...	1.08838863397...	7755
giza	1.27340963880...	25.53	35.86	5.78152465928...	4.54304111649...	114676
nitzan	1.23658450347...	0.36	8.01	3.69604662282...	3.94911568885...	57386
OOS	0.09748059977...	-7.70	8.18	14.1027877253...	1.77718421348...	17189
DummyAdnetwork	1.12817869905...	-3.92	3.69	1.77032654479...	5.83864743433...	1669
DummyAdnetwo...	1.26592055122...	4.99	14.75	3.69399437432...	6.06538165065...	8664

Status: Game 5 is running