

**Workshop in Ad Auction Agents**

**January 2011**

1. High Level Architecture

1.1. [Online Design](#online_design)

1.2. [Offline Design](#offline_design)

2. Agent Components

2.1. [Trainer](#trainer)

2.2. [Modeler](#modeler)

2.3. [Estimator](#estimator)

2.4. Optimizer

2.4.1. [Description](#optimizer_description)

2.4.2. Algorithm

2.4.2.1. [Multiday](#optimizer_md)

2.4.2.2. [Singleday](#optimizer_sd)

2.4.2.3. [Other parts of the algorithm](#optimizer_other)

2.4.3. [Conclusion](#optimizer_conclusion)

Our trainer implements the given Trainer interface. It's composed of three steps:

1. First, it calls our parser, which is an implementation of the given Parser interface, and it fills our DB structure with the data from the given game log. Our DB is built in a tree form and is divided to 4 levels:

* The root is called LogDS and it holds an AgentDS for each agent and data relevant to a full game sequence (squashing parameter etc.).
* AgentDS holds in it an array of DayDS which represents each one of the days in the game. AgentDS holds in it some personal attributes of the agent as well (name, specialty etc.).
* DayDS is the object representing a day in the game, it holds in it a map of queries containing a QueryDS as value for each one of the 16 possible queries.

Also holds in it specific status for that day (bank status etc.).

* QueryDS holds in it all the relevant data for the query.

1. We provide a data structure to the modeler and estimator, for them to fill with their estimation throughout the game. With that DS and the DS of the game log we analyze the level of accuracy of the estimations. This will enable us to find the weak link in our agent play flow.
2. We provide a textual output of our conclusions, and maybe provide better parameters for the configuration files in the different components of the agent.

If we will see that concluding from a single game log each time isn't accurate enough, we will add the possibility of merging several LogDS together (from several different game logs) and concluding from the mean results of several games instead of results of a single game each time.

The Modeler's role in the program is to estimate for a certain query, three factors:

1. CPC (Cost per click).
2. Number of impressions.
3. Position.

We evaluate each of these in the following ways:

CPC

The Modeler uses the double exponential method to estimate the predicted CPC, using a weighted average with the suggested bid (Given by the estimator).  
 We use the known CPC of past days to estimate the CPC of the requested day.  
 If we do not yet know the actual CPC of the previous day, we use the estimation given by the Modeler for that day. This will work correctly given that the Modeler will be required to estimate the K+1'th day only after it has estimated the K'th day.  
Luckily, that is indeed the case in our agent.

Impressions

The Modeler uses the external Particle Filtering component (which estimates the amount of users in each state) to calculate the amount of estimated impressions for a specific query.

The particle filtering

The particle filtering component simulates distribution of population of 10000 users among the different possible states. It does it by generating each day a new population of "particles" from the old population and by observations of the total impressions of each of the F2 queries. Because each user in the state IS has a 1/3 probability of moving to one of the F states, the number of F0, F1, F2 impressions resulting from a given user population follows a binomial distribution. This fact allows us to model the new population using the number of F2 impressions.

Position

A component called Position Analyzer uses the average advertiser positions for each query to estimate each advertiser’s yesterday’s position for each query.  
After it estimates that, the Position Analyzer uses our agent’s CPC for yesterday to know what the agent with pos (i-1)’s bid was (i being our ad’s position on the previous day). Also, we have lower bound on the bid of the agent with pos (i+1), which is our own agent’s bid for yesterday.  
Then, we check if the current bid given to the Modeler by the Estimator is high or low “enough” to support a change in position, and such estimates our own bid’s position for tomorrow.  
Note: The Position Analyzer assumes the competing agents will not change their bids drastically from day to day.

Time Complexity

The Modeler performs mathematical calculations with regard to known or estimated data, without loops or recursive functions. Therefore, its running time is fixed for each estimation request.

The Estimator estimates the clicks, the conversions and calculates the CTR (click through rate), conversion rate and the total profit.

Clicks

The clicks are estimated according to the combined data received from the Position Analyzer (the higher the ad is, the more likely it’ll get clicked), Particle Filterer and the Modeler. After this estimation, the focus level of the clicked ad is taken into consideration: generic ads won’t improve the probability of a click, but focused ads will (by a predetermined factor).

Conversions

The number of conversion is estimated by the previously estimated clicks and the data from the Particle Analyzer.

- The rest of the estimations are simple mathematical calculations, which use the clicks, conversions and modeler estimations.

Time Complexity

The Estimator performs mathematical calculations with regard to known or estimated data, without loops or recursive functions. Therefore, its running time is fixed for each estimation request.

**Description:**

The optimizer module is in charge of creating the BidBundle that would maximize the agent's profits. The module is made of two classes – the FirstOptimizer and the FirstOptimizerQuery class. Unlike in the sample agent we were given with, we opted for having the entire business logic of the component in the FirstOptimizer class, and have the FirstOptimizerQuery class as a simple data-holder class.

The general algorithm for maximizing the profits is the Multiday/Singleday algorithm we saw in class – that is, a recursive Multiday procedure which tries different divisions of *capacity* over the days of the game, and a recursive Singleday procedure which finds the best division of a given capacity over the different queries (and also finds the best bids and ad types for those queries while it's at it).

The procedure which commences the algorithm described above is the "PrepareForTomorrow" procedure (which is called at the end of the "HandleQueryReport" method). Then, when the "Optimize" procedure is called, we simply extract the values we've previously calculated, and create a bid-bundle out of those.

**Multiday****:**

The Multiday procedure should run over possible capacity sizes "cap" for the current day, and find the best possible sum for (Singleday(cap, today) + Multiday(remainingCap, tomorrow)), where the stop-case is when we arrive on the last day – and then we just calculate Singleday(cap), where cap is simply the entire capacity we have left.

Since there are 60 days in the game, even for just 2 different possible capacity sizes, we'll get 2+4+…+260 =~ 261 recursive calls (On the first day, yet obviously it remains large for other days as well), which is infeasible. However, many of those calculations repeat themselves, and so we could cache them.

First, we'll define K levels for the multiday's capacity size for a specific day. Level 0 will be 0, and the rest will be a rising series of capacity sizes which should depend on the advertiser's capacity we were given with at the beginning of the game. Note that K can't be too high or the program will become infeasible, even with caching.

Multiday's profit result depends on two parameters – day and the current status of the capacity window. For maximum time-efficiency (which is the main resource we wish to save), we'll save the cache in a 6-dimensional array – where the first five dimensions represent the capacity window, so that the index of dimension d is the capacity level we used for 5-d days ago, and the 6th dimension is the day index. The value of the array's cell will be the expected profit for those parameters.

Since the caching depends on the capacity window, The window should be initialized with five 0's. Also, the cache should reset at the end of each day, as results may vary due to the new information we will then possess.

Note that this caching mechanism ensures we will have no more than K5\*60 recursive calls for the Multiday procedure – since for any given day, no more than K5 different capacity window combinations can occur.

Since we do not wish to allow for negative capacity on any given day (**subject to change**), level 0 needs to be 0 capacity. However, we'd almost never wish to allocate 0 capacity for a specific day, and so the Multiday's main loop should not run for level 0. Zero capacity will only occur when level 1 (which is the lowest level besides level 0) is higher than the available capacity at a specific time. In that case, and only in that relatively-rare case, a 0 will be inserted to the capacity window. Therefore, 0 will usually not appear in the capacity window, and we thereby save many recursive calls, since now the Multiday recursive calls amount is likely to be something closer to (K-1)5\*60.

For further improving the running time, we'll want to cache the calls to Singleday as well. The result of the Singleday function depends on the given day and the capacity allocated to that day, yet it also depends on previous days, as previous days affect the users' state modeling, and thereby have an impact on the estimations we make in the Singleday function.

This raises the following problem: Assume we're really on day 30, and during a Multiday recursive call, we want to run Singleday for day 40 with a specific capacity. The result we'll get might not be the same as the result we'll get for a different path of recursive calls from day 30 to day 40, even if we'll have the same capacity allocated for day 40, because it also depends on what we plan to do on days 30-39 – and therefore if we use caching, we might extract from the cache a result which is incorrect for the current setup we're trying to estimate.

However, we'll note that trying to estimate the users' state on the long run is likely not to be accurate in itself, and so the optimizer will use the following solution: on real day D, we won't cache in the Multiday procedure Singleday's result of the following N days, and we'll cache for the rest. This enables us to have both accurate estimations for the near future and yet have the benefits of the caching mechanism, where we approximate far-future estimations without regarding to the difficult-to-predict users' state.

This latter improvement will cost us in extra Singleday calls – as a bound, we increase the number of Singleday calls for each of the closest N days from K\*N to K5\*N. Yet for the closest 4 days, we'd actually have a lower number of calls than K5, since we don't have to go over all the options of a 5-values capacity window. Instead, on the first day there'll be K calls, on the second K2, K3 on the third and K4 on the fourth.

This means that for N >= 5, instead of having N\*K Singleday calls for the first N days, there'll be (K+K2+K3+K4 + (N-4)\*K5) = ((K\*(K4-1)/(K-1)) + (N-4)\*K5) calls, which is less than N\*K5 (our previous bound).

For N <= 4, we'll get (K\*(KN-1)/(K-1)) calls.

Another issue which is raised due to this type of caching, is that when calculating Singleday for the last day, we'd want to use our entire available capacity, without any regard to the different "capacity levels" which we previously defined. Therefore, we can't cache the results in the cache we've previously described. If we don't somehow cache these results, assuming we're on day D where D <= 55, then we'll have to make the last day's Singleday calculations a total of K6 times (because we'll reach day 59 recursive call K5 times, and for each time we'll call day 60's recursively K times).

Instead, we'll keep a cache of Capacity+1 size (where Capacity is the advertiser's capacity we were allocated with at the beginning of the game), as the space requirements won't be too harsh since it's one-dimensional as we only need it for the last day. Obviously, the capacity we'll have left on the last day is somewhere between 0 and Capacity, so this cache will definitely suffice. So now the amount of times the last day's Singleday calculations will occur is Min{Capacity+1, K6}.

However, we'll further notice that since those calculations depend on merely the allocated capacity (assuming we're still far from day 60, and thus we ignore the users' state), and the allocated capacity depends only on the window entries of the past 4 days, then we can reach day 60's Singleday calculations with no more than K4 different capacity allocations. Yet since we're after the capacity allocation for day 60, and we won't mind the window afterwards since it's the last day of the competition, there is no importance to the order of the entires of the past 4 days, only their values – therefore, we can further decrease the upper bound and show it to be Min{Capacity+1, (K+3)!/((K-1)!\*4!) } (the general method to reach this answer will be explained in the Singleday section later on).

One final improvement can be made if we choose the capacity allocation levels in a way that they "overlap" – for example, if one level L2 is the average of two other levels, L1 and L3, then a window of 2 L1 and 2 L3 will generate the same available capacity for the last day as a window of 4 L2 would – and thereby save us some calls to Singleday.

As in with the other days, in the case we're N days or less away from day 60, the cache should not be used for that day (because we should take into account the users' state modeling). However, this is relatively cheap for us, as the last day's Singleday will only be called KN-1 times (when we're N days away from day 60), since there's no need to loop on the K different capacity allocations, while when we're on day 30, then we'll calculate day 30+N Singleday KN times.

As previously mentioned for the first cache described, the other caches in the Multiday procedure should also be reset at the end of each day.

Multiday should also save the information it gets back from Singleday (best capacity/ad-type/bid for each query, and the expected profit), for the setup it finds best, when Singleday is ran on the day we're currently on. We do this to save one run over Singleday, instead of having another run once we have the ideal capacity for today extracted from the Multiday function.

**Singleday:**

The Singleday procedure should, when given a day index and a capacity allocation as parameters, find out the best division of the capacity among the 16 queries – including finding out the best bids, capacity allocation and ad-types for each query (which is not inter-dependent among queries, yet must be calculated at this time, because it can affect the capacity division between queries). The general algorithm is to run over the range of possible capacity allocation for each query, then for that query, find the best bid and ad-type (simple loops with calls to the estimator – relatively cheap, as no recursion is required), and then estimate the sum of profits of the current query (by calling the estimator) and the rest of them (recursively) with (capacity – (current-query-allocated-capacity)) allocation. The stop-case is when we reach the last query, where we simply calculate the profit when allocating the entire leftover allocation for that query (with the best possible bid and ad-type).

Similarly to the Multiday's procedure, the general algorithm is infeasible – for just 8 levels of possible capacity allocation for query, we'll have 8 + 82 + … + 815 > 245 recursive calls. Therefore, we'll have to use caching here as well.

The caching here mechanism will work similarly to the one in the Multiday section. Once again we'll have capacity-allocation levels (their amount denoted by M), this time for each query. Level 0 should be set to 0, as we won't always want to bid on each of the queries. We'll have a three-dimensional array for the first 15 queries, where the first index is the query index, the second one is the capacity left, and the third one is the day index. The value is the resulted profit. For the last query, we'll have a two dimensional array, where the second dimension is the day index, and the first one will be of size (Capacity+1), as we'd like to use the entire remaining capacity for the current day. Similarly to the last day's cache in the Multiday procedure, this cache should actually have a smaller upper bound, as usually (all days except last) we'll reach the Singleday function with a capacity which is one of the K pre-defined multiday capacity allocation levels, and then the first 15 queries will each take an allocation which is one of the M predefined levels of Singleday capacity allocation per query.

This, together with the fact only their sum matters and there is no importance to the order (more on that in the next paragraph), makes it very likely that we'll reach the last query with the same values many times, and thereby save time by using the cached result. (Possible improvement: change order of the queries array on day 1, such that the last query will be for a "specialized" product of the advertiser).

Another similarity to one of the caches we defined in the Multiday section is that having "overlapping" capacity allocation levels can improve performance in this procedure as well, due to the result only depending on the query index, day index and available capacity, which is the total capacity allocation for the day minus the total capacity allocated for previous queries – with no importance to the order or actual sizes of the previous queries' allocations.

Unlike the caches described in the Multiday section, however, the caches in the Singleday procedure might need to be reset upon finishing the original call to the function (which occurs at the Multiday procedure) rather than at the end of the day. This happens when the day parameter to the Singleday procedure is no more than N days away from the current day – due to our will to receive better estimations which also rely on a more updated users' state modeling.

Then, when we're on day D, calculating Singleday for day (D+N-1) for example, we won't consider cached data from previous calls to Singleday, and rather call the estimator additional times, so that the estimations will also take into account the supposed actions we took in days (D, …, D+N-2) when modeling the users' state.

On the other hand, when calculating Singleday for a day which is more than N days away from the current one, the Optimizer will cache the result and keep it until the end of the day. As previously mentioned, for days which are more than N days away from the current one, we don't consider the users' state modeling for the estimations, as this would interrupt our caching mechanism and be costly time-wise, and at the same time it's quite difficult to predict an accurate picture of the users' state on the long run.

Keeping the cache for the entire day length can save us quite a few recursive Singleday calls – even though we cache the Singleday result in the Multiday functions for days which are more than N days away, we still have to calculate for each of those days the result of Singleday K times. Even though each of those K Singleday runs starts with a different capacity allocation, we might get to a recursive call where the result for a recursive call in one or more of the M levels (of Singleday capacity allocation for query) can be fetched from the cache.

This also implies that the K levels of Multiday capacity allocation and M levels of Singleday capacity allocation per query should be inter-connected, as this will greaten our chances for having another kind of "overlappings", where we'll reach calculations which we can simply extract from the cache instead of repeating – for example, say two of the Multiday levels L1 and L2 (L1<L2) have a difference of R, then if we set one of the Singleday's levels to R, then once we reach Singleday with L2, we won't have to make any recursive calls for the Singleday level equal to R – as we'd already have the answer for cached from when Singleday was called with L1 (since Singleday has level 0 as 0, and so (L1-0) = (L2-R)).

Without taking into account the time saved by our caching when we use the cache between different calls to Singleday on the same day, and also dropping out the effects of the two different "Overlapping" levels methods which were previously discussed, we can show an upper bound to the running time of the Singleday procedure which is much less than M1 + … + M16:

we wish to know how many recursive calls will be saved in one Singleday run by using the cache – like in Multiday's cache for the last day's Singleday's result, we'll notice that for a specific query D, there's no importance to the order of the levels chosen for the queries before D, only their sum. Since we do not take into account the possibility of "overlappings", this question is identical to asking how many options are there to pick D numbers out of M possible levels, when repeating is allowed and there's no importance to the choosing order. This is a common probability problem, whose answer is known to be .



In Multiday's last-day running time analysis, we only had to take into account the previous four days' chosen levels, meaning D=4, M=K, and so the result was ((K+3)!)/(4!\*(K-1)!).

In Singleday's analysis, however, all previous chosen levels must be taken into account.

Therefore, the number of options (and therefore the upper bound) is:



This is a much better upper bound. We previously showed that without caching, the running time for M=8 is more than 245. In comparison, by placing M=8 in the above formula, we get 735,470, which is much more sane, and likely to be feasible (depending also on the running time of the estimator's "estimateQuery" method).

One more issue which should be addressed by the Singleday procedure is that besides estimating the best profit for the day, it also needs to output the bids, capacity allocation and ad-types for each query which produce this estimated best profit. Unlike in the Multiday procedure, where returning the value is simple because we're only really interested in the best capacity allocation for the first day, here we're required to return information about all queries.

The way we'll extract that will be by keeping three local arrays of size 16, which will keep the best bids, best capacity and best ad-type for each query. Then, at each recursive call in Singleday, we'll create three identical arrays, and pass them on to the next recursive call.

When we're at a recursive call on query number Q, upon finding a better profiting setup than what we've had so far, we'll update the arrays that were passed to this recursive call, for only query Q and the queries which come after it. This is because query Q only "knows" what's the best setup from itself forward - The updated information will be the current setup for query Q, and the rest of the information will come from the three copy arrays which we passed on to the recursive call for query index (Q+1).

That way, once we're done with the original call of Singleday, we'll have three arrays filled with the correct bids/ad-types/capacity-allocations which lead to the best estimated profit.

**Other parts of the algorithm:**

Besides the Multiday and Singleday procedures, the Optimizer has a few more less-meaningful yet non-negligible tasks:

1. As previously mentioned, the Multiday procedure is called from the "PrepareForTomorrow" procedure, which is called by the "HandleQueryReport" procedure. The latter is only called at the end of day 2, and so we don't actually use the Multiday/Singleday procedures for the first two days.

Instead, we'll use those 2 days for "practice-bidding", meaning, trying out different bids for different queries which will help us:

* 1. Model our opponenets.
  2. Model the users' states.
  3. Find out the minimum bar for acceptable bids.

2) (**Subject to change)** The Optimizer is also in charge of setting a campaign daily limit. Our implementation will try to use that to further increase "specialized" products sales, by doing the following: First, we'll sum up the limits we calculated for each of the queries (the limit of each query is directly dependent on the allocation we calculated for that query in the Singleday procedure), and set that value to be the campaign daily limit. Then, we increase the limits of queries of specialized products/manufacturers by pre-defined bonus-percentages. Thus we enable more sales of "specialized" products, while keeping the original general limit which we found to be ideal for the current day.

**Conclusion:**

The Optimizer's main objective is to find the best possible combination of bids/ad-types/limits of all queries for the entire length of the game according to estimations generated by the Estimator. Because of that, it is easy to conclude that the Optimizer's main constraint is Time. Due to that fact, almost everything about the Optimizer involves with improving its running time, and every small change in parameters can lead to a huge difference in the execution time.

This is why when writing the Optimizer, we should rather neglect "good programming" rules, if we could instead save some more running time. Even extracting information of a cache which is saved as a hashmap inside a hashmap (As we originally designed for the Multiday cache to be), when the map is relatively large, can be far less efficient than extracting the same information from an array (which wastes some space, but that's far less of a concern) when that extraction might occur a really large number of times. Therefore, the optimizer's data structures should be mainly made of arrays. This might also apply for having some variables kept as class fields, instead of passing them to and back from functions a great number of times.

In this document, we mentioned the parameters K, N and M. It is impossible to predetermine what are the values we'll give them in the final version of the Optimizer. The main task once the optimizer (and also the Modeler + Estimator) is complete, will be to try out different values for those parameters and have them as high as possible while maintaining a legitimate running time.

However, given the values of those parameters, we can set an upper bound to the running time of the Optimizer on any given day (which is not dependent on the actual levels for the Multiday and Singleday procedures):

N≥5:



N≤4:



Where:

K = The number of Multiday capacity allocation levels.

M = The number of Singleday capacity allocation for query levels.

N = The number of days following the current day for which we wish to have estimations which don't ignore the users' state modeling.

Cap = The advertiser's capacity, given to us in the beginning of the simulation.

Est = The running time of the Estimator's "estimateQuery" procedure.

(Note this bound is the worst case scenario for any day, yet obviously as the game progresses, the bound for the current day decreases – since the earlier we are in the game, we have more days forward to go over in the Multiday procedure. Therefore the running time will be the highest on day 1, 2nd highest on day 2, and so on…)

Alon Horowitz

Amit Cwajghaft

Nir Biran

Ran Ziv

Gilad Einbar

Oded Oren