# CS 136 Assignment 7: Sponsored Search Auctions

David C. Parkes
School of Engineering and Applied Sciences, Harvard University
Out: October 20, 2017
Due: **5pm** sharp: **Friday, October 27 2017**
Submissions to `Canvas`

[**Total: 137 Points**] *Submit a single zip file containing your code and your writeup.* The total number of points includes a small bonus for good performance in an in-class tournament.

This is a group-assignment to be completed by groups of up to 2 students. While you are permitted to discuss your agent designs with other students, each group must develop its own code and write-up its own submission. (If you need help finding a partner please post to Piazza.)

Your submissions should be made to Canvas: the code must be in .py files and the writeup of the analysis must be in PDF format. Both partners must contribute to all aspects of the assignment. It should go without saying that you must write all of the code yourself, per Harvard's Academic Integrity Policy. If you use any code whatsoever from any source other than the official Python documentation, you **must** acknowledge the source in your writeup.

Moreover, please follow the below instructions exactly when submitting your code:

- Submit a total of 2 files to canvas: your writeup in PDF format, and a .zip file containing only the following .py files: `TEAMNAMEbb.py, TEAMNAMEbudget.py, vcg.py`. If you're using an IDE, then please **do not** submit the entire project directory created by the IDE. Please **do not** submit .git files.
- If you want to create functions which both your agents can share, please create them in a new file named `TEAMNAMEhelper.py`, and submit that as well (zipped together with the other 3 .py files). Your code should run on an un-modified version of the handout code; in particular, you **should not** modify any of `auction.py, stats.py, gsp.py, history.py, truthful.py, util.py`, and you should not submit these files.
- Please **do not** use any functions not in the Python standard library. So, you should not use `pandas` or `numpy`. You can do the operations that you'll need to do using a combination of the `math` library as well as functions like `map, reduce, zip, filter`.

## 1 Introduction

You will program bidding agents to participate in a generalized second-price (GSP) auction. The first agent you will design uses a *balanced bidding* strategy, which is inspired by Defn. 10.6 in the reading. You will also design an agent to do as well as possible in a competitive environment, and in the presence of daily budget constraints on how much it can spend (see, e.g. the discussion in Section 10.6.1 in the reading). You will also explore the effect that the design of payment rules has on revenue.

## 2 Setup

**Generating .py-files:** Pick a group name, perhaps based on your initials, so it will be unique in the class. Run `python start.py TEAMNAME`, substituting your group name for `TEAMNAME`. This will create appropriately named template files for your clients. For example, if your group name was "abxy":

```
> python start.py abxy
Copying bbagent_template.py to abxybb.py...
Copying bbagent_template.py to abxybudget.py...
All done.  Code away!
```

In each of these newly generated files, you will need to change the class name `BBAgent` to your teamname plus client specification (e.g. for the balanced bidding agent: `class Abxybb`). This class name should match the file name exactly, though it can have capital letters (and the filename should be exactly the one generated by `start.py`).

**The simulator:** You are given an ad-auction simulator. We ignore quality effects, so that in a time period, only the position matters in determining the probability of a click and thus the expected number of clicks received by an ad.

The simulator works as follows:

- Time proceeds in discrete periods $(0, 1, 2, \ldots, 47)$. Each period simulates 30 minutes and includes multiple auctions, and the 48 periods correspond to a day. The simulator can be used to simulate multiple days by increasing the number of iterations.
- In the first period, an agent's bid is queried through `initial_bid()` whereas for all subsequent periods, the bid is queried through a call to `bid()`. The simulator tracks money and values in integer numbers of cents. The same bid value is used for every auction that occurs within a period. To encourage competition, the number of available positions is one less than the number of agents in the market.
- Each period is simulated by collecting bids, assigning positions, determining the expected number of clicks received by each bidder, and determining payments and utilities. Sometimes there is a *reserve price* (see Section 4). The positions are assigned in order of submitted bid.
- Number of clicks: Let $c_j^t$ denote the number of clicks received by an ad in position $j$ (from 1 to the number of positions) in period $t$. For the top position, this follows a cosine curve:

$$c_1^t = \text{round}(30\cos(\frac{\pi t}{24}) + 50), \quad t = 0, 1, \ldots, \tag{1}$$

where round$(x)$ returns the nearest integer value to $x$. The number of clicks starts at 80, falls to 20 by period 24, and increases to 80 by the end of a day.

The number of clicks decreases according to a multiplicative position effect. The number of clicks received by an ad in position $j$ $(j > 1)$ in period $t$ is

$$c_j^t = 0.75^{(j-1)} c_1^t, \tag{2}$$

so that it decreases by a factor of 75% from position to position.

- The price in the GSP auction depends on the next highest bid. Given this, the utility in period $t$ to agent $i$ occupying position $j$ is

$$u_i^t = c_j^t(v_i - pay_{\text{gsp},j}^t) = c_j^t(v_i - b_{j+1}^t), \tag{3}$$

  where $v_i$ is the per-click value, and $pay_{\text{gsp},j}^t$ the price for position $j$ under the GSP auction and equals $b_{j+1}^t$, which is the amount of the next highest bid (or zero or the reserve if there is no such bid.) Thus is the number of clicks received multiplied by the utility (= value - price) per click.

- Budget constraint: The total payment by an agent in position $j$ in period $t$ is $c_j^t b_{j+1}^t$. Each agent has a *daily budget constraint*, but for most of the assignment this will be high enough not to matter. For the competition this is \$600/day and it will matter. An agent with budget still available at the start of a period is still allowed to bid. For this reason, an agent is allowed to slightly over-spend its budget. Once the budget is exhausted the bid in all subsequent periods of the day must be \$0 (the simulator will ensure this if you try to bid more than \$0.)
- At the start of each day, the per-click value $v_i$ of an agent is sampled independently from the uniform distribution on $[0.25, 1.75]$. The simulator also runs additional permutations of value assignments to agents to improve the statistical significance of the estimated utility. If the number of agents is at most 5 then all permutations of values to agents are tested. Otherwise, 120 random permutations are tested. (You can change this threshold with the `--perms` option)
- The score of an agent is the total utility over all 48 periods, and averaged over all permutations of values to agents that are used. In addition, multiple iterations can be run. This has the effect of repeating the test over multiple days. In this case the score is the average total utility over all permutations and all days. For a single day, the total utility to agent $i$ with value $v_i$ is $\sum_{t=0}^{47} u_i^t = \sum_{t=0}^{47} c_{x_{it}}^t(v_i - b_{x_{it}+1}^t)$, where $x_{it}$ is the position assigned in period $t$ to agent $i$ (and $c_{x_{it}}^t = 0$ if the agent is not assigned to a position in period $t$. *Any unspent budget has no effect on utility or score.*

**Source code:** Familiarize yourself with the provided code. You will need to change the agent created by `start.py`, as well as the file `vcg.py`. You'll want to take a look at the simple truthful-bidding agent in `truthful.py`, as well as the the GSP implementation in `gsp.py`.

**Testing:** Here are some initial test commands to run. The following command gives a list of helpful command line parameters:

```
> python auction.py --h
```

The following command can be used to test the code. It runs the auction with five truthful agents for two periods (rather than the default of 48 periods). The `--perms 1` command forces the simulator to assign only one permutation of value to the agents.

```
> python auction.py --loglevel debug --num-rounds 2 --perms 1 Truthful,5
```

The number of periods defaults to 48 if this is not set.

The following command runs the auction with a reserve price of 40 cents. The `--iters 2` command specifies that the 48 periods will be repeated twice, with different value samples (i.e., two days.) The `--seed INT` (where INT is any integer) command sets the seed of the random number generators in the simulator and allows for repeatability.

```
> python auction.py  --perms 1 --iters 2 --reserve=40 --seed 1  Truthful,5
```

**Tips**

- *Permutations*: If you're running an experiment on a population of agents that each have the same strategy, use `--perms 1` to make the code run faster. In this case this is probably OK because you're likely comparing the average utility or revenue of the auction, and not looking at specific agents.
- *Pseudo-random numbers*: If you're trying to track down a bug, or understand what's going on with some specific case, use `--seed INT` to fix the random seed and get repeatable value distributions and tie breaking.

**Comments**

- *Numbering convention*: The positions (slots) in this write-up are 1-indexed (top slot is slot number 1) but 0-indexed in the code. Don't be confused by this!
- *Ties*: In the case of two tied bids one of the two is randomly chosen to be allocated th higher slot.
- *State*: The agent you write can access history from the previous period within each day. This could allow for more sophisticated strategies.
- *Workarounds*: The code is designed so that it's hard to mess up the main simulation accidentally, but because everything is in the same process, it is still possible to cheat by directly modifying the simulation data structures and such. Don't!
- *Bugs*: If you find bugs in the code, please let us know. If you want to improve the logging or stats or performance or add animations or graphs, feel free :) Send those changes along too.
- *Questions*: If something is unclear about the assignment, please ask a question on Piazza. Please don't post solution code but it is OK to post code snippets are fine (use your judgment).
- *Debugging tips*: Be careful about division by zero errors! Some of you had these in problem set 2. Also make sure list indices are always integers!

# 3 The Balanced Bidding Agent

1. **[2 Points]** What is your team name?

2. **[30 Points] Designing a bidding agent**

   You are given a truthful bidding agent in `truthful.py`.

   In `TEAMNAMEbb.py`, write an agent that best-responds to the bids of agents in the previous period. In particular, the agent follows a *balanced bidding* strategy. Consider period $t$. Let $b_{-i}^{t-1}$ denote the bids from the agents other than $i$ in the period $t-1$. Suppose there are $m$ positions. Balanced-bidding proceeds as follows:

- Given bids $b_{-i}^{t-1}$, agent $i$ targets the position $j^*$ that maximizes

$$\max_{j \in \{1,\ldots,m\}} [pos_j \cdot (v_i - t_j)], \tag{4}$$

where $pos_j$ is the position effect (this falls by a multiplicative factor of 0.75 each slot) and $t_j$ is the price the agent would pay for position $j$ given bids $b_{-i}^{t-1}$. For example, in GSP auction, $t_j$ equals the $j$-th highest bid in $b_{-i}^{t-1}$.

- (*Not expecting to win*) If price $t_{j^*} \geq v_i$ in this target position, then bid $b_i^t = v_i$ in period $t$.
- Otherwise:
  (a) (*Not going for the top*) If target position $j^* > 1$, then set bid $b_i^t$ to satisfy

$$pos_{j^*}(v_i - t_{j^*}) = pos_{j^*-1}(v_i - b_i^t). \tag{5}$$

  (b) (*Going for the top*) If $j^* = 1$, then bid $b_i^t = v_i$.

This strategy is motivated by the balanced bidding discussion the reading (Section 10.4.1).[1] The file `TEAMNAMEbb.py` provides a skeleton of a balanced bidding agent. You will need to complete the code to compute the expected utility for each position and the optimal bid.

Note that in the code the bids in the previous round can be accessed by looking at `history.round(t-1)`; the function `slot_info` does this for you.

3. **[20 Points] Experimental Analysis**

   **To answer the following questions, run the simulation with 5 agents. By default the budget is $5000, which is not binding. Leave it this way!**

   (a) **[10 Points]** What is the average utility of a population of truthful agents? What is the average utility of a population of balanced bidding agents? *Compare the two cases and explain your findings.*

   Make use of the `--perms`, `--seed`, and `--iters` commands, e.g. `--perms 1 --seed 2 --iters 200` would be a good starting point.

   (b) **[10 Points]** In addition, what is the average utility of one balanced-bidding agent against 4 truthful agents, and one truthful agent against 4 balanced-bidding agents? For the new experiment, make use of the `--seed`, and `--iters` commands, but you will now want to run multiple permutations. Note that you can add multiple agents types using:

```
> python auction.py  --perms 10 --iters 200 Truthful,4 abxybb,1
```

   *What does this suggest about the incentives to follow the truthful vs. the balanced bidding strategy?*

[1] Cary et al. "Greedy bidding strategies for keyword auctions" *Proc. ACM EC 2007* pp 262–271, show that this balanced bidding strategy converges to the spiteful Nash equilibrium of the GSP auction.

# 4 Experiments with Revenue: GSP vs VCG auctions

In this section we compare the revenue properties of the GSP and VCG auctions with different reserve prices. A reserve price, $r > 0$, sets a minimum price for any position. Used carefully, reserve prices can increase revenue. The reserve price in the GSP works as follows: only bids (weakly) above $r$ can be allocated. The agent in the lowest allocated position pays the maximum of the reserve price and the maximum bid of unallocated bidders. The VCG auction works in a similar way and is explained below.[2]

4. **[55 Points] Auction Design and Reserve Prices**

   **Run all simulations with 5 agents. Leave the budget to its default of $5000.**

   (a) **[20 Points]** Complete the code that runs the VCG auction in `vcg.py`. The allocation rule is already implemented. You need to implement the payment rule. Because of the reserve price, it is most convenient to use the recursive form of the VCG payment rule (see the proof of Theorem 10.3 in the reading).

   In particular, suppose there are 3 bidders with bids weakly greater than the reserve price, and $b_1 \geq b_2 \geq b_3$, and say that $b_4$ is the fourth highest bid. Bidders 1–3 are allocated the top 3 positions.

   Let $t_{\text{vcg},i}(b)$ denote the expected payment by bidder $i$ in a single auction given bid profile $b$. For bidder 3, this is $t_{\text{vcg},3}(b) = pos_3 \max(r, b_4)$. For bidders 1 and 2, in positions 1 and 2 respectively, this is $t_{\text{vcg},i}(b) = (pos_i - pos_{i+1})b_{i+1} + t_{\text{vcg},i+1}(b)$.

   (b) **[10 Points]** What is the auctioneer's revenue under GSP with no reserve price when all the agents use the balanced-bidding strategy? What happens as the reserve price increases? What is the revenue-optimal reserve price?

   You can set the reserve price in the simulation with the command line argument `--reserve INT` (where INT is the reserve price in cents). Also use `--perms 1` and `--iters 200`.

   (c) **[10 Points]** What is the auctioneer's revenue under VCG with no reserve price when all agents are truthful? What happens as the reserve price increases? Explain your findings and compare with the results of part (a).

   Again use the `--perms`, `--seed`, and `--iters` commands, e.g. `--perms 1 --seed 2 --iters 200`.

   (d) **[10 Points]** Fix the reserve price to zero. Explore what might happen if a search engine switched over from the GSP to VCG design. For this, run the balanced-bidding agents in GSP, and at period 24, switch to VCG, by using the `--mech=switch` parameter. What happens to the revenue?

   Again use the `--perms`, `--seed`, and `--iters` commands, e.g. `--perms 1 --seed 2 --iters 200`.

   (e) **[5 Points]** In one paragraph, state what you learned from these exercises about agent design, auction design, and revenue? (There is no specific right answer).

[2] One way to think about it is that the reserve is made operational by including a "dummy bidder" whose bid is the amount of the reserve, and ignoring all bids with value less than $r$.

# 5 The Competition

5. **[30 Points] Budget constraints**

   The balanced-bidding agent does not consider the budget constraint when deciding how to bid. In the final part of the assignment, your task is to design a *budget-aware agent*.

   This agent will compete in the simulated GSP auction against the agents submitted by other groups. You might like to test your design against a variety of other strategies.

   For example, you could write an agent that measures competition, or tries to drive up the payments of other agents so that they pay more and exhaust their budgets! Example ideas include:

   - Trying not to bid too much when click-through rates are low.
   - Try to bid when other agents are not bidding very much and the price is lower.

   **For the purpose of the competition, the daily budget constraint will be set to $600 (use the `--budget` flag). We will run a GSP auction with a small reserve price. Auctions will contain around 5 agents.**

   The competition will be structured as a tournament, with agents placed into groups of 5 or 6 and the top few agents making it into a semi-final and final. The precise structure of the tournament will depend on the number of agents submitted.

   (a) **[25 Points]** In `TEAMNAMEbudget.py`, write your competition agent.
   Describe in a few sentences how it works, why it is designed this way, and how you expect it to perform in the class competition. *You are not expected to spend many hours on writing an optimal agent, unless you want to. Consider a few possible strategies, try them out, pick the best one.*

   (b) **[5 Points]** Win the competition (!) Likely parameters for the competition are
   `./auction.py --num-rounds 48 --mech=gsp --reserve=10 --iters 200`
   (followed by the list of submitted agents)