Computation and Economics - Spring 2012
Assignment #5: Strategic bidding and revenue in online ad-auctions

Professor Sven Seuken
Department of Informatics, University of Zurich

Out: Monday, March 26, 2012
Due: **12:15** sharp: **Thursday, April 5, 2012**
Submissions should be made to `econcs.uzh.spring12@gmail.com`.
For submission format, check description below.

[**Total: BSc 100 Points, MSc 120 Points**] This is a group-assignment to be completed by groups of **up to 2 students each**. While you are permitted to discuss your clients with all students as much as you like, each group must write their own code and explanations. In addition, there will be a small bonus for clients that perform well in the class tournament. If you want a partner and don't have one, post to piazza as early as possible. Your submissions should be made via email to the course email address as attachments: Code in .py-files and writeup of analysis as PDF, sending .zip-archives is also ok.

Introduction

Your task in this assignment is to write two bidding agents for an ad auction using GSP, and then do some mechanism-side experiments to explore the effects of different payment rules and reserve prices.

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 GROUPNAME`, substituting your group name for `GROUPNAME`. 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`).

The simulator: You are given an ad-auction simulator. It has the following structure:

- Time proceeds in integer rounds $(0, 1, 2, \ldots, 47)$. Each round simulates 30 minutes, so the simulation models one full day.
- Values per click, w_i, are uniformly distributed between \$0.25 and \$1.75. In the first round, your bid is queried through `initial_bid()` whereas for all subsequent rounds, bids are placed through a call to `bid()`. The simulator tracks money and values in integer numbers of cents.
- If a non-zero reserve price is set, bids less than the reserve will be ignored. To encourage competition, the number of available slots in any time period is one fewer than the number of active bidders (unless there is only one active bidder). An active bidder is one with a bid above the reserve.
- Click-through rate of the highest slot: The number of clicks in the *first slot* in each round, c_1^t, follows a cosine shape:

  $$c_1^t = R(30\cos(\pi t/24) + 50), \quad t = 0, 1, \ldots$$

  where $R(\cdot)$ denotes rounding to the nearest integer value. This means, if we use 48 rounds (which models a full day) for the simulation, then the clicks for the top slot start at 80 in the first round, go down to 20 in round 24, and rise again to 80 in round 48 (with an average of 50).
- Click-through rate of other slots: Let c_1^t denote the number of clicks that the first slot receives in the current round. Then, the number of clicks in the i-th slot is given by

  $$c_i^t = 0.75^{i-1} c_1^t,$$

  which closely fits the real-world click dropoff.
- Each period is simulated by collecting bids, assigning slots, calculating clicks, and determining utilities. We first consider the GSP auction. Thus, agents are allocated to slots according to their bids (in order of decreasing bids), and the price per click they have to pay is the the bid of the agent allocated to the next lower slot. For every round the utility of agent i occupying slot j is calculated as the number of clicks of that slot, c_j^t, times the difference between the agent's value per click, w_i and the price per click, p_j^t:

  $$u_i^t = c_j^t(w_i - p_j^t) = c_j^t(w_i - b_{j+1}^t).$$

- Payments and budget: In every round, the payment per allocated agent is $c_i p_i$ (i.e. the clicks of the slot it gets times the per-click price of that slot). The budget over the entire day is \$1750. As long as you still have a positive budget, you will still be allowed to bid in the next round, even if your submitted bid will mean you will over-spend in this round. However, you can over-spend your budget in at most one round (and end up with a small deficit). If the sum of the payments from all previous rounds is greater than or equal to your budget (i.e. if you have spent or even overspent your budget), you can only bid \$0, or if you bid more than \$0, the simulation will reduce your bid to \$0.
- The per-click values w_i are drawn from a random distribution and then shuffled through the following process: first, n random values are pulled from a uniform distribution on \$0.25 to \$1.75. Note that there are $n!$ different ways of assigning these values to the

n different agents. If $n!$ is less than 120, simulations will be run for each of these $n!$ permutations. If $n!$ is greater than 120, then only 120 permutations (i.e., a subset of all possible permutations) will be randomly chosen. (You can change this threshold with the `--perms` option - for debugging, you'll often want this to be 1).

- Winning: The winners will be determined based on cumulative utility over every round of each instance, i.e., for one instance this is $\sum_{t=1}^{48} u_i^t = \sum_{t=1}^{48} c_j(w_i - p_j^t)$.[1]

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:

```
> ./auction.py --h
```

The following command can be used to test the code. It runs the auction with five truthful agents for two rounds. The `--perms 1` command forces the simulator to assign only one permutation of the same values to the agents.

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

The following command runs the auction with a reserve price of 40 cents. The `--iters 2` command specifies that the auction will be run twice, but with two different value draws. This can be useful in combination with the `--seed INT` (where INT is any integer) command, which gives you repeatable value distributions. These two together can be used to compare different agent populations.

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

Tips

- Permutations: If you're looking at a symmetric population of agents, use `--perms 1` to make the code run faster – if the strategies are the same, who has what value does not effect the outcome.
- 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.

[1]Note what this means for the role of the budget in determining your overall utility: any part of your budget that you spend will count against you, and any part of your budget that is left over will count in your favor. That's because every payment you make appears as a negative term in your utility function. In particular this also means that purposefully overspending your budget does not make sense, because that will also count against you. Really you should just focus on the sum of your utilities.

Problem Set

1. [**40 Points**] Designing a bidding agent

   You are given a truthful bidding agent in `truthful.py`. In `GROUPNAMEbb.py`, write a best-response agent that uses the so-called *balanced-bidding algorithm*. This algorithm uses a similar idea to what we saw in the lecture notes regarding the buyer-optimal envy-free bidding strategies. An agent playing a buyer-optimal envy-free equilibrium strategy chooses his bid such that he has no envy for the bidder one slot above him and such that he has no fear of retaliation by that bidder.

   The balanced bidding algorithm works very similarly. The motivation behind this algorithm is to make your bid be a best response to the other agents' bids from the previous round. This involves picking as a target slot the one that optimizes utility assuming other agents don't change their bids. There are many possible bids that would achieve the target slot, and balanced bidding says to pick the particular bid b that makes an agent indifferent between getting the targeted slot (whose price is the next lower bid) and getting the slot just above at price b (which could happen if you get 'jammed' by the bidder above.)

   So the balanced bidding strategy for player i is as follows. Fix the bids of the other players, b_{-i}, to be the ones from the previous round. Now player i

   - targets the slot i^* that maximizes his utility:

     $$i^* = \arg\max_k (s_k(w_i - p_k)),$$

     where p_k is the price he would have to pay to get slot k,
   - chooses his bid b_i for the next round so as to satisfy the following equation:

     $$s_{i^*}(w_i - p_{i*}) = s_{i*-1}(w_i - b_i).$$

   If $i^* = 0$ (i.e. if the target slot is the top slot), then we arbitrarily set $s_{i*-1} = 2s_{i*}$ to make the strategy well-defined.

   The file `TEAMNAMEbb.py` provides you with a skeleton of a balanced bidding agent. The only two things you have to do is to compute a vector of expected utilities for each slot and the optimal bid using the formulas above.

2. [**BSc 7 Points, MSc 17 Points**] Agent analysis

   To answer the following questions, run the simulation with 5 agents.

   (a) [**7 Points**] What is the average utility of a population of only truthful agents? What is the average utility of a population of only 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 20` would be a good starting point.

   (b) [**MSc 10 Points**] Compare the performance of the truthful agent i) when all other agents are truthful and ii) when all other agents use balanced bidding. Explain your findings.

   For both cases, make use of the `--seed`, and `--iters` commands. For i) use `--perms 1` additionally.

3. **[BSc 38 Points, MSc 48 Points]** Mechanism Design Analysis

For the following questions, we introduce the idea of a *reserve price* r. This is the minimum price at which the auctioneer is willing to sell a slot (or, more generally, an item). This means, if all agents bid below the reserve price then nobody gets allocated a slot. In the case of GSP, the agent that gets the lowest of the allocated slots (there could be one or more unallocated slots) pays the maximum of the reserve price and the bid of the next lower bidder. For example, in a one slot auction with two bidders, if one bidder is above the reserve ($b_1 \geq r$) and one below ($b_2 < r$), then the first bidder pays r and not b_2. Thus, in this example, the auctioneer actually makes more revenue because of the reserve price r. In ad auctions, reserve prices are used to increase the revenue of the seller. As we have seen in the example, by setting an appropriate reserve price, the seller can indeed make more money. However, if the reserve price is set too high (e.g., $r > b_1 > b_2$ in the example), then it might happen that no bidder wins and the auctioneer makes no money. Thus, we can try to figure out the *optimal reserve price*, i.e., which reserve price maximizes the seller's revenue.

For the following questions, if answering them requires one or more simulation runs, then do that **with 5 agents**.

(a) **[7 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 point?

Note that 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 20`.

(b) **[19 Points]** Implement the VCG auction in `vcg.py`. The file has the allocation rule already implemented. You only need to implement the payment rule according to Equation (7.13) in chapter 7 of the lecture notes.

Note: You do not have to worry about the reserve price. Just implement Equation (7.13) from the lecture notes.

(c) **[7 Points]** When the reserve price is zero, what is the revenue of VCG compared to GSP when the agents are truthful? Explain your findings.

Again use the `--perms`, `--seed`, and `--iters` commands, e.g. `--perms 1 --seed 2 --iters 20`.

(d) **[MSc 10 Points]** When the reserve price is zero, explore what might happen if Google switched from GSP to VCG: run the balanced bidding agents against GSP, and at round 24, switch to VCG, by using the `--mech=switch` parameter. What happens to the revenue? Compare just GSP, just VCG, and switching.

Again use the `--perms`, `--seed`, and `--iters` commands, e.g. `--perms 1 --seed 2 --iters 20`.

(e) **[5 Points]** Bigger picture: in one paragraph, what did you learn from these exercises about agent design and implementation, mechanism design, and revenue? We aren't looking for any particular answers here, but are looking for evidence of real reflection.

4. **[15 Points]** Competition

   The balanced bidding agent from task 1. does not take into account the fact that your agent has a budget. Here, you are asked to write an agent that takes this into account. This agent will compete in a GSP slot auction with the agents submitted by the other groups, so you may want to test it against a variety of strategies. For example, if everyone spends his budget early you might want to bid more in later rounds when competition is lower, or if everyone waits for the end, you might want to bid more early etc.

   (a) **[10 Points]** In `TEAMNAMEbudget.py`, write your class competition agent. Describe in a few sentences how it works, why you chose this strategy and how you expect it to perform in the class competition.

   (b) **[5 Points]** Win the competition. We will run a GSP slot auction with all submitted agents. The auction will have a small reserve price, e.g. `--reserve=10`. The agents will be ranked according to their cumulative utility over all instances of the auction. The points from the competition will be awarded as follows: the winning team gets 5 points, the second 4 points, the third 3 points and so on.

   Likely parameters for the competition are

   ```
   ./auction.py --num-rounds 48 --mech=gsp --iters=50 --reserve=10 Team1budget,1...
   ```

   but they may change depending on the submitted agents.

   Note: You are not expected to spend many hours on writing an optimal competition agent, unless you want to. Consider a few possible strategies, try them out, pick the best one.

Comments

Cheating 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, 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.

Help If something is unclear about the assignment, please ask a question on NB as early as possible. If you need help, post to Piazza! Please don't post solution code, but otherwise code snippets are fine (use your judgment).