1 Introduction

The Trading Agent Competition (TAC) is an annual tournament in which teams of programmers from around the world build autonomous agents that trade in a market simulation. Application domains for these market simulations run the gamut from travel to supply chain management to financial and energy markets to ad auctions and exchanges.

In this lab, you will build an agent to play a greatly simplified version of the TAC Ad Exchange (AdX) game, in which agents play the role of ad networks, competing to procure contracts (i.e., advertising campaigns) from advertisers (e.g., retailers), and then bid in an exchange on opportunities to exhibit those ads to Internet users as they browse the web. The game is complicated by the fact that certain demographics are more valuable to certain retailers, but user demographics are not always fully visible to the ad networks. Moreover, bidding repeats over a series of simulated days, and agents' reputations for successfully fulfilling past contracts impact their ability to procure future contracts.

This lab and the next are intended to introduce you to one of our final project options, which will be a more complicated version of the games you play in lab, but (far) less complicated than the actual TAC AdX game.

2 Game Description

The main simplifications of the game you play in this lab, as compared to the full TAC AdX game, are:

1. The game lasts only one day, so no long-term planning is necessary, and there are no reputation effects.
2. Your agent will not compete to procure campaigns. Instead your agent will be assigned one, randomly, from a known distribution (see Appendix A).
3. User demographics are fully visible.

Consequently, your agent's only job in this game is to bid on (and win) **impression opportunities**, which are opportunities to exhibit ads to Internet users as they browse the web. Your goal will be for your agent to do so in such a way as to fulfill its campaign, as inexpensively as possible.

2.1 Ad Auctions and Ad Campaigns

Each day of the simulation, a random number of Internet users browse the web. These users hail from multiple market segments, of which there are a total of 26, corresponding to combinations of `{Male, Female}` x `{HighIncome, LowIncome}` x `{Old, Young}`. A market segment might target only one of these attributes (for example, only `Female`) or two (`Female_Young`) or three (`Female_Old_HighIncome`). Users' market segments drawn from the distribution described in Appendix A.

For each user, a **second-price sealed-bid auction** is run to determine which agent to allocate that user's advertising space to, and at what price. Ties are broken randomly, so if there are two winning bidders in a market segment, each will be allocated (about) half the users in that segment at the price they bid.

Ad networks are motivated to participate in these auctions by their desire to fulfill advertising campaigns. A campaign is a contract of the form, "The ad network will show some number of ads to users in some demographic. In return for these impression opportunities, the advertiser agrees to pay the ad network said budget." More specifically, each campaign is characterized by:

- A **market segment**: a demographic(s) to be targeted.
- A **reach**: the number of ads to be shown to users in the relevant market segment.
- A **budget**: the amount of money the advertiser will pay if this contract is fulfilled.

N.B. Campaigns also have start and end days. However, in the version we are playing in lab today, the entire game only lasts 1 day, so you can ignore their start and end days.

Here is an example of a campaign: `[Segment = Female_Old, Reach = 500, Budget = $40.0]`

To fulfill this campaign, your agent must show at least 500 advertisements to older women. If successful, it will earn $40. To show an advertisement to a user, you must win the auction for that user. (Yes, we, as Internet users, are regularly auctioned off!) But note that winning an auction for a user who does not match a campaign's market segment does not count toward fulfilling that campaign.

2.2 Decisions: Bids and Spending Limits

Unlike the sealed-bid auctions we have studied in class, which are one-shot auctions, the auctions in this game are repeated, as users arrive repeatedly. However, agents can only bid in these auctions once!—before simulating them begins. Consequently, agents must reason in advance about how events might unfold over the course of the day, and perhaps make contingency plans. The AdX game provides a mechanism for making a contingency plan in the form of spending limits. These limits are upper bounds on the amount an agent is willing to spend in a specific market segment.

If your agent is allocated a campaign whose market segment is very specific (e.g., `Female_Old_HighIncome`), then it won't really have a choice about which users to bid on; it has to bid for users in precisely that market segment, or it cannot earn a positive profit. However, if its market segment is less specific (e.g., `Female`), it can bid different amounts in the `Female_Old` and `Female_Young` markets, for example, based on how much competition it thinks there will be in each. Keep in mind, though, that the order in which users arrive is random. So if it bids more on `Female_Old` than `Female_Young`, but then if all `Female_Old` users arrive before any `Female_Young`, it may end up spending its entire budget for that campaign on `Female_Old` users. For this reason, when bidding on a market segment, your agent might want to specify a spending limit in each market segment.

An agent can also specify a campaign spending limit to ensure that it does not spend more than some preset total across *all* market segments associated with their campaign. In sum, the key decisions an agent must make in the One-Day AdX game are what bids to place on what market segments, and what spending limits should accompany those bids.

2.3 Scores

At the end of each simulation, the server will compute the profit earned by each agent/ad network. This profit is the product of the proportion of the campaign's reach fulfilled and the campaign's budget, less total spending. The proportion of reach fulfilled only counts users won in the relevant market segment, and cannot be higher than 1 (it does not benefit an agent to show impression opportunities to users beyond its reach). The agent with the highest profit wins.

TAC AdX is a game of incomplete information, as each agent knows its own campaign(s) only, not those of its competitors. These campaigns are generated at random, using distributions described in Appendix A. Because of this randomness inherent in the game, we will run multiple simulations and compute average scores to determine a winner.

3 API for AdX One-Day Game

3.1 `OneDayBidBundle` Object

To avoid the gruesome communication overhead required to conduct each ad auction in real time (each day there are 10,000 simulated users!), the agents use a `OneDayBidBundle` object to communicate their bids to the server all at once.

A `OneDayBidBundle` object is constructed using three arguments:

1. **campaign_id**: the ID for the campaign you are bidding on.
2. **day_limit**: a limit on how much you want to spend in total on that day.
3. **bid_entries**: a list of `SimpleBidEntry` objects, which specify how much to bid in each market segment.

A `SimpleBidEntry` object is also constructed using three parameters:

1. **Market Segment**: there are a total of 26 possible market segments.
2. **Bid**: a `double` value.
3. **Spending Limit**: a `double` value that represents the maximum value the agent is willing to spend in the given market segment.

For example, say your agent decides to bid 1.0 in all market segments in your campaign, and it wants to limit spending in total and in each market segment to the campaign's budget. First, you would create a new `SimpleBidEntry` for the campaign's market segment that bids 1.0 and limits spending in the market segment to the campaign's budget:

```
bid_entry = SimpleBidEntry(
    market_segment=self.campaign.market_segment,
    bid=1.0,
    spending_limit=self.campaign.budget
)
```

Note: if your agent's market segment is general, this entry will bid in all the specific market segments that are subsets of its market segment at the given price using the given limit. For example, if the campaign's market segment is `Female_Old`, this bid entry will bid 1.0 with a spending limit of the campaign's budget in both `Female_Old_HighIncome` and `Female_Old_LowIncome`.

Next, you would create a list of bid entries with this particular `bidEntry`:

```
bid_entries = [bid_entry]
```

Finally, you would create a `OneDayBidBundle` for your campaign that includes these bid entries and limits total spending to your campaign's budget.

```
bid_bundle = OneDayBidBundle(
    campaign_id=self.campaign.id,
    day_limit=self.campaign.budget,
    bid_entries=bid_entries
)
```

3.2 `MyAdXAgent` Class

You should implement your agent strategy in `my_adx_agent.py`. More specifically, you should implement the `get_action` function, which returns a `OneDayBidBundle` with all the agent's bids for the (one-day) game.

The agent is notified about its campaign via an observation dictionary that is passed to `get_action`. You can access the campaign's data using:

```
campaign_data = observation["campaign"]
self.campaign = Campaign.from_dict(campaign_data)
```

3.2.1 Agent Naming

As in previous labs, your agent needs a name. You can give it a name in the main function block.

3.2.2 Helper Functions

The `core.game.market_segment` module provides helper functions to work with market segments. To iterate over all possible segments, use:

```
for segment in MarketSegment.all_segments():
    ...
```

To check if one market segment is a subset of another (i.e., whether users in `segment2` also belong to `segment1`), use:

```
if MarketSegment.is_subset(segment1, segment2):
    ...
```

(**N.B.** Market segments are subsets of themselves.)

4 Competition

We will run a competition at the end of lab, consisting of 100 simulations of the one-day AdX game. To connect your agent to the competition server, modify the configuration variables at the bottom of `MyAdXAgent.py`:

```
server = True
host = "adx-server.cs.brown.edu"
port = 8080
```

Then run your agent with:

```
python3 MyAdXAgent.py
```

This will connect your agent to the AdX game server using the asynchronous function `connect_agent_to_server()`. Your agent will automatically submit bids and play against nine competitors in each round.

5 Testing

To make sure that your agent works as intended, we have provided local testing support in the `LocalArena` class. In particular, you can run full AdX simulations directly from your Python environment, without connecting to the server. To test your agent locally, simply run:

```
python3 MyAdXAgent.py
```

This command will launch 10 agents in total (your agent, one `AggressiveBiddingAgent`, and several `RandomAdXAgent` opponents) and execute 10 rounds of the one-day AdX game. You can adjust the number of rounds, agents, or verbosity settings by modifying the `LocalArena` configuration at the bottom of the file.

If this test succeeds, you should see how your agent performs against the sample bots. Use this setup to debug and refine your bidding strategy before submitting your final version.

A Campaign and User Distributions

Each campaign targets one of 20 possible market segments, a combination of at least two of the three attributes (chosen uniformly at random). A campaign's reach is given by the average number of users in the selected segment (listed in the tables below) times a random reach factor, selected from the set $\{\delta_1, \delta_2, \delta_3\}$, where $0 \leq \delta_i \leq 1$, for all i. The exact values of these factors are tailored to the number of agents in the game. In particular, for the ten-agent games we plan to run, we will use $\delta_1 = 0.3$, $\delta_2 = 0.5$, and $\delta_3 = 0.7$. The budget is always \$1 per impression.

Table 1: User Frequencies

Segment	**Average Number of Users**
`Male_Young_LowIncome`	1836
`Male_Young_HighIncome`	517
`Male_Old_LowIncome`	1795
`Male_Old_HighIncome`	808
`Female_Young_LowIncome`	1980
`Female_Young_HighIncome`	256
`Female_Old_LowIncome`	2401
`Female_Old_HighIncome`	407
Total	**10000**

Table 2: User Frequencies: An Alternative View

	Young	**Old**	**Total**
Male	2353	2603	4956
Female	2236	2808	5044
Total	4589	5411	10000

	Low Income	**High Income**	**Total**
Male	3631	1325	4956
Female	4381	663	5044
Total	8012	1988	10000

	Young	**Old**	**Total**
Low Income	3816	4196	8012
High Income	773	1215	1988
Total	4589	5411	10000