Auction Games

CS404 Agent-based Systems Coursework

1 Introduction

Imagine an auction of paintings by famous artists. There is an auction room, with an auctioneer that presents each piece to be sold, and all bidders then write their bids for the item onto a secret sealed note that is handed to the auctioneer. The auctioneer then declares the highest bidder the winner, takes their payment, and starts the next round with a new item. There might be a target number of paintings by the same artist to get, in which case the first bidder to get this wins; otherwise, the auction continues until everyone runs out of money or there are no more items to sell, and the bidder with the highest total value of paintings is the winner.

Your objective is to implement strategies for a Python 3 bidding bot that will participate in such an auction.

2 The Auction

The auction runs in the following way. A number of bidders connects to the AuctionServer, which then announces the winning condition, the item types (i.e, the artists) and the number of each being sold and value of each of these types, how much each money each bidder starts with, which bid the highest bidder will pay and optionally the sequence of the auction.

The AuctionServer will then announce an item type to be bid upon. Your AuctionClient will use the above information to determine an amount to bid and submit it to the AuctionServer. Once done, the AuctionServer will declare the highest bidder the winner, who will then be charged (but not necessarily the amount they bid, see below) and receives the item. If there are drawn positive bids, then the winner is chosen at random from the drawn bidders; if however the drawn bids are all at 0 (no-one bid), no-one wins, and the item is discarded.

The auction will continue until either there are no more items to sell, all bidders have run out of money, or if there is a set winning condition, e.g., a bidder managed to acquire N items of the same type, in which case they are declared the grand winner. If there is no set winning condition, the game will only end once one of the first two conditions is met, and then the bidder who ends with the highest total value in items is the grand winner.

Note that however whilst the highest bidder will always win, the auction may be set up so the highest bidder does not pay their own bid. It can be set up so that the highest bidder is only charged the second highest bid (see `winner_pays`).

You will write your strategies in AuctionClient, and then everyone's AuctionClients will all be logged into one AuctionServer, where a tournament will be played across four games:

- Game 1: First to buy 5 of any artist wins, the highest bidder pays own bid, and the auction order known.
- Game 2: First to buy 5 of any artist wins, the highest bidder pays own bid, but the auction order is not known.
- Game 3: Highest total value at the end wins, the highest bidder pays own bid, and the auction order known.
- Game 4: Highest total value at the end wins, the highest bidder pays the second highest bid, and the auction order known.

Note that for all games in this tournament, there will be four types, (Picasso, Van_Gogh, Rembrandt and Da_Vinci), the auction size will be 200, the starting budget 1000, and where the win condition is based on final value, each Picasso is worth 4, Van_Gogh worth 6, Rembrandt worth 8 and Da_Vinci worth 12.

3 Implementation

Provided to you are four Python 3 files: AuctionServer, AuctionClient, run_auction and run_client.

AuctionServer contains the definition for the AuctionServer class, which sets up and runs the auction. It has the following arguments:

- `host`: Where the server will be hosted, keep this at "localhost" for your own testing.
- `ports`: Either a list of port sockets that the clients will connect through, or if a single number, P, is given, it will use all ports from P to P + numbidders. If you get any error that a port is already in use or locked or something, just change this number to any other number above 1000 and try again.
- `numbidders`: The number of clients i.e., bidders, that will be playing.
- `neededtowin`: The number of the same type of painting a player needs to get to be the grand winner. If 0, then the total value once the auction has ended will be used to declare the grand winner instead.
- `itemtypes`: List of the different types of paintings, i.e, the different artists.
- `numitems`: A dict that can either be used to manually set how many items of each type there should be in the auction, or if unset i.e., "`numitems={}`", this is generated randomly according to `auction_size`.
- `auction_size`: If `numitems` is set, this should be 0. Otherwise, this is how many items are in the auction; items will be randomly generated until there are this many to sell.
- `budget`: The starting budget of all the players.
- `values`: A dict of the value for each type. This will be used to determine the total value for each player if `neededtowin == 0`
- `announce_order`: If True, the bidders are told the sequence of the items to be sold before auction begins. If False, they are not, and will have to use `numitems` and the past bidding record to guess which item will be sold next.
- `winner_pays`: An index of which bid the highest bidder pays. If 0, they pay their own bid, if 1 they pay the second highest, if 2 the third etc... Note however that the winner always pays at least 1, even if the second highest is 0.

The function `announce_auction` sets up and announces the auction details to all clients, whilst `run_auction` can then run the auction until it is completed.

AuctionClient is the code for a bidding client, and this is where you will implement your strategies. Note that an AuctionServer must be initialised first so it is listening on the given ports before an AuctionClient can be initialised and connect to the server. It has the following arguments:

- `host`: Where the AuctionServer to connect to is hosted, keep this at "localhost" for your own testing.
- `port`: The port to connect to the AuctionServer to. This must be unique to this AuctionClient (no other AuctionClient can use this port), and an initialised AuctionServer must be ready and listening to this port.
- `mybidderid`: The name for your bidder. If not given with initialisation, you will be asked to input this on the command line. Note that it must be unique to your bidder, and can contain alphanumeric or underscore characters.
- `verbose`: If set to True, then your client will print out statements detailing its input and progress.

The function `play_auction` runs the loop for receiving data from the AuctionServer and passing the bids back until the auction is done.

You will also see provisionally defined functions of `determinebid` and four `bidding_strategy` functions for each of the four games, and it is in these functions you should implement the algorithm for your strategies.

Currently, they just run the function `random_bid`, which returns a random number between 1 and amount of budget the bidder has left. You can use this random strategy to test your strategy against.

`determinebid` receives all the relevant data as input and has a comment block that explains what each argument is. If you wish, you are also free to access and create self defined variables, which may be useful if you want to store state between calls of `determinebid`. You may also add code to the `__init__` or `play_auction` functions to help achieve this, but be very careful not to change either the networking code or the auction parameters or you might either crash or confuse your bot! Also note that if your bidder tries to bid more than it has budget left (accessible by `standings[mybidderid]['money']`), the AuctionServer will cap the bid to its current budget.

run_auction is a script to initialise an AuctionServer with the given parameters and start it running and waiting for AuctionClients to connect. Once it is set up, you can manually run AuctionClients by initialising them with a port the AuctionServer is listening on, and once `numbidders` AuctionClients have connected to the server on their different ports, it will run the auction.

run_clients is a script that automatically initialises and runs not only an AuctionServer, but also `numbidders` number of AuctionClients, and is provided for convenient testing.

4 Submission

Your coursework submission will consist of a single compressed file (either .zip or .tgz) containing:

- An AuctionClient.py file, encoding the strategies for each game.
- A four pages Analysis.pdf file, with the analysis of each of the four corresponding strategies and the reasons of your design choices. The pdf should be written in IEEE two-column conference format.

The coursework file should be submitted through Tabula.

Each submission will be run as follows. We first set up the auction with the desired parameters for each tournament in the run_auction.py script, then run it. This will create an auction room with those parameters. Then we will run each student's client with the following three lines in a Python script:

import AuctionClient from [submission filename]

bidbot = AuctionClient(port=[Unique port], mybidderid=[student ID])

bidbot.play_auction()

Where [submission filename] is the filename of the student's submission, [Unique port] is one of the ports the AuctionSever is configured to listen on and [student ID] is the student's unique ID. Your submission file name should be named with your student ID, followed by the suitable extension.

Please make sure that your submission is suitable for this operation.

5 Evaluation

The coursework is worth 50% of the module credit. Its marking scheme is structured as follows:

- 30% strategy implementation (how you design and structure your strategies)
- 30% strategy performance (how each strategy performs in the games)
- 40% quality of the analysis (how you describe and analyse your strategies)

Each of the four strategies will be evaluated independently and the result averaged across them.

6 Cheating/Plagiarism

This coursework is an individual piece of work. All submissions will be put through plagiarism detection software which compares against a number of sources, including other submissions for CS404, submissions at other universities, web sources, conference papers, journals and books. Please see the student handbook for more information, or ask if you need guidance.

Acknowledgements

The code has been developed by Alexander Carver, Department of Computing, Imperial College London (`alexander.carver15@imperial.ac.uk`).