First Sealed Bid Auctions

Thomas M. Breuel

UniKL

Agent-Based First Sealed Bid Auctions

first sealed bid auctions:

First sealed bid auctions are the simplest form of auction:

- each participant estimates the value of the item based on information from the auctioneer
- each participant makes a sealed bid on the item
- the auctioneer selects the highest bid and awards the item to the highest bidder

auction protocol:

The auctioneer calls the following methods in sequence:

- `giveEstimate(estimate)`
- `solicitBid()`
- winning bid: update `balance` and `value`
- losing bid: call `winningBid(paid)`

default strategy:

The default strategy of the bidder is to bid a fraction of the estimated value.

common value model:

In this model, we are looking at *common value auctions*.

That is:

- each bidder assigns the same value to the object
- they receive unbiased but random estimates of the value

```
class FSBidder:
    def __init__(self,fraction=0.5):
        self.balance = 0
        self.value = 0
        self.fraction = fraction
    def giveEstimate(self,estimate):
        self.estimate = estimate
    def solicitBid(self):
        return self.estimate * self.fraction
    def winningBid(self,paid,winner):
        pass
    def gain(self):
        return self.value+self.balance
```

```
class FSBAuctioneer:
    def __init__(self,participants,uncertainty=0.1,values=None):
        self.participants = participants
        self.uncertainty = uncertainty
        if values is None: values = lambda: 100.0+10.0*rand()
        self.genvalue = values
    def auction(self):
        value = self.genvalue()
        for p in self.participants:
            estimate = value+randn()*self.uncertainty*value
            p.giveEstimate(estimate)
        bids = [clip(p.solicitBid(),0.0,10*value) for p in
            participants]
        winner = argmax(bids)
        bid = bids[winner]
        winner = self.participants[winner]
        winner.balance -= bid
        winner.value += value
        for p in self.participants:
            p.winningBid(bid,p==winner)
    def gains(self,N=10000):
        for i in range(N): self.auction()
        return tuple([p.gain()/N for p in self.participants])
```

possible objectives

- maximize wealth individually
- be the winner (make more than the other participants in N rounds)
- cooperate to maximize wealth

Simple Auctions

simple auctions

- each participant uses a fixed strategy
- strategy: bid some fraction of the estimated value
- all auctions are independent from each other
- participants have no memory of prior bids against each other

number of auction rounds

```
1 N = 10000
```

symmetric auction

```
1 participants = [FSBidder(fraction=0.5),FSBidder(fraction=0.5)]
2 print FSBAuctioneer(participants).gains()
```

```
(24.868080931969367, 24.697395708112065)
```

Symmetric strategies yield symmetric outcomes.

QUESTION Why is the payoff around 24 / item?

asymmetric strategies

```
1 participants = [FSBidder(),FSBidder(fraction=0.8)]
2 print FSBAuctioneer(participants).gains()
```

```
(0.045950372397365734, 20.846089074031369)
```

Higher bidder gains more.

negative payoffs for high fractions

```
1 participants = [FSBidder(fraction=1.0),FSBidder(fraction=1.0)]
2 print FSBAuctioneer(participants).gains()
```

```
(-2.9740161831421603, -3.0075112556659618)
```

If the participants bid too high, they both lose money.

small payoffs for high fractions

```
1 participants = [FSBidder(fraction=0.9),FSBidder(fraction=0.9)]
2 print FSBAuctioneer(participants).gains()
```

```
(2.6389559714615811, 2.5638315299245704)
```

If both participants bid high but not too high, they both still make a small profit.

strategy depends on opponents

```
1 participants = [FSBidder(fraction=0.8),FSBidder(fraction=0.85)]
2 print FSBAuctioneer(participants).gains()
3
4 participants = [FSBidder(fraction=0.9),FSBidder(fraction=0.95)]
5 print FSBAuctioneer(participants).gains()
```

```
(4.8238313020857344, 8.1663335215651323)
(1.3561667955572076, 0.586999942180665 65)
```

HYPOTHESIS Bidding higher is always the better strategy.

OBSERVATION The higher bidder doesn't always come out ahead, it depends on both participants. (Why?)

effect of uncertainty

```
participants = [FSBidder(fraction=0.8),FSBidder(fraction=0.9)]
print FSBAuctioneer(participants,uncertainty=0.2).gains()

participants = [FSBidder(fraction=0.8),FSBidder(fraction=0.9)]
print FSBAuctioneer(participants,uncertainty=0.0).gains()
```

```
(3.2677625396087593, 1.7790844026373933)
(0, 10.49870639319548)
```

HYPOTHESIS Bidding higher is always the better strategy.

OBSERVATION The higher bidder doesn't always come out ahead, it also depends on the accuracy of the value estimate. (Why?)

effect of value distribution

```
1 def values(): return rand()*100.0
2
3 participants = [FSBidder(fraction=0.8),FSBidder(fraction=0.9)]
4 print FSBAuctioneer(participants).gains()
5
6 participants = [FSBidder(fraction=0.8),FSBidder(fraction=0.9)]
7 print FSBAuctioneer(participants,values=values).gains()
```

```
(2.7588602679667558, 6.2259375117512885)
(1.2447756191551234, 3.1303866547008337)
```

QUESTION Are there value distributions for which the order of strategies is different?

Nash Equilibrium

different symmetric strategies

```
for fraction in linspace(0.1,1.1,11):
    participants = [FSBidder(fraction=fraction),FSBidder(fraction=
        fraction)]
    gains = FSBAuctioneer(participants).gains()
    print fraction,gains
```

```
0.1 (46.859423258231914, 47.056438617941616)
0.2 (40.930243835629398, 41.876834014461281)
0.3 (36.072790356088838, 35.65768238812516)
0.4 (30.330413639497849, 30.323879726443398)
0.5 (25.394484130337275, 24.145714064596763)
...
0.8 (8.2394011634957796, 8.0746473329929866)
0.9 (2.5960990092790803, 2.552865130140999)
1.0 (-2.8837855062480551, -2.8976846017991194)
1.1 (-8.6459163976106801, -8.5852007384903679)
```

Note that if the players are cooperating, they should simply agree to bid as little as possible. They then maximize their gain (of course, the seller of the item loses).

unstable strategy

```
1 participants = [FSBidder(fraction=0.1),FSBidder(fraction=0.1)]
2 print (0.1,0.1),FSBAuctioneer(participants).gains()
3
4 participants = [FSBidder(fraction=0.1),FSBidder(fraction=0.2)]
5 print (0.1,0.2),FSBAuctioneer(participants).gains()
```

```
(0.1, 0.1) (46.702872563423881, 47.1858351541836)
(0.1, 0.2) (0, 84.005480421922641)
```

However, such a strategy requires cooperation, since any one of the players can make more money by increasing their bids.

stability

```
for fraction in linspace(0.8,1.0,5):
    participants = [FSBidder(fraction=fraction),FSBidder(fraction=
        fraction-0.03)]
    print fraction,
    print "  %.2f %.2f"%FSBAuctioneer(participants).gains(),
    participants = [FSBidder(fraction=fraction),FSBidder(fraction=
        fraction)]
    print "  %.2f %.2f"%FSBAuctioneer(participants).gains(),
    participants = [FSBidder(fraction=fraction),FSBidder(fraction=
        fraction+0.03)]
    print "  %.2f %.2f"%FSBAuctioneer(participants).gains()
```

```
0.8   10.57 7.25   8.32 7.94   6.00 8.48
0.85   6.89 5.35   5.29 5.38   3.85 4.99
0.9   3.61 3.16   2.50 2.69   1.67 1.61
0.95   0.28 0.83   -0.26 -0.20   -0.55 -1.73
1.0   -2.98 -1.52   -2.97 -2.93   -2.70 -4.97
```

Note that around `fraction=0.9`, participant 2 is worse off by increasing their bid.

Nash equilibrium

- non-cooperative game
- each player knows all the choices of all the other players
- it's the choice of strategy for each player where no player gains anything by changing their strategy

For games with continuuous parameters, we need to check:

- local optimum
- global optimum

It's (supposedly) rational for all players to choose the Nash equilibrium.

Repeated Auctions

so far

- non-cooperative auctions
- no memory
- fixed strategy (bid fraction of estimated value)
- two participants

QUESTION Is bidding a fraction of the estimate the best possible strategy in that situation? Can we do better?

adaptive strategies

- repeated auctions
- one fixed player, one adaptive player
- players can observe some of the results from previous auctions

Can the adaptive player beat the fixed player?

adaptive bidder

- estimate the strategy used by the fixed bidder by averaging the actual fraction of past winning bids
- update our fraction as we go along
- the fact that we estimate the fraction based on the winning bid lets us learn a fraction that wins
- that is, our estimate is automatically a little higher than the fraction used by competitors

adaptive bidder

```
1  class FSBidderWithStats:
2      def __init__(self,fraction=0.5):
3          self.balance = 0
4          self.value = 0
5          self.fraction = fraction
6          self.fractions = [fraction]
7      def giveEstimate(self,estimate):
8          self.estimate = estimate
9      def solicitBid(self):
10         return self.estimate * mean(self.fractions)
11     def gain(self):
12         return self.value+self.balance
13     def winningBid(self,paid,winner):
14         if not winner: self.fractions.append(paid/self.estimate)
```

adaptation

```
1 participants = [FSBidder(fraction=0.7),FSBidder(fraction=0.9)]
2 print FSBAuctioneer(participants).gains()
3 participants = [FSBidder(fraction=0.7),FSBidder(fraction=0.5)]
4 print FSBAuctioneer(participants).gains()
5 participants = [FSBidder(fraction=0.7),FSBidderWithStats(fraction
      =0.5)]
6 print FSBAuctioneer(participants).gains()
```

```
(0.89109407065113988, 9.4605019546218099)
(30.877465914995586, 0.51291892834194075)
(0.63863819792715981, 6.9224323901677502)
```

The adaptive bidder performs almost as well as a good fixed strategy.

adaptation vs all fixed strategies

```
for fraction in linspace(0.1,1.1,11):
    participants = [FSBidder(fraction=fraction),FSBidderWithStats(
        fraction=0.05)]
    auctioneer = FSBAuctioneer(participants,uncertainty=0.05)
    print fraction,
    print "  %.2f %.2f"%auctioneer.gains(),
    print "  %.3f"%mean(auctioneer.participants[-1].fractions)
```

```
0.1   5.61 87.58   0.114
0.2   7.30 74.45   0.225
0.3   5.28 64.57   0.340
0.4   5.45 53.09   0.449
0.5   4.83 42.11   0.562
...
0.8   1.45 10.41   0.900
0.9   0.37 -0.65   1.017
1.0   -0.44 -11.75   1.133
1.1   -1.60 -20.84   1.233
```

Adaptation wins over any of the fixed strategies, until it actually starts losing money. (I.e. it succeeds at always winning the auction, but beyond some point, that becomes itself a losing strategy.)

adaptive bidder

```
1 class FSBidderWithStats1(FSBidderWithStats):
2     def solicitBid(self):
3         return self.estimate * clip(mean(self.fractions),0.0,0.95)
```

Here is a simple fix for that behavior: we just limit our bids to up to 95

test of improved adaptive bidder

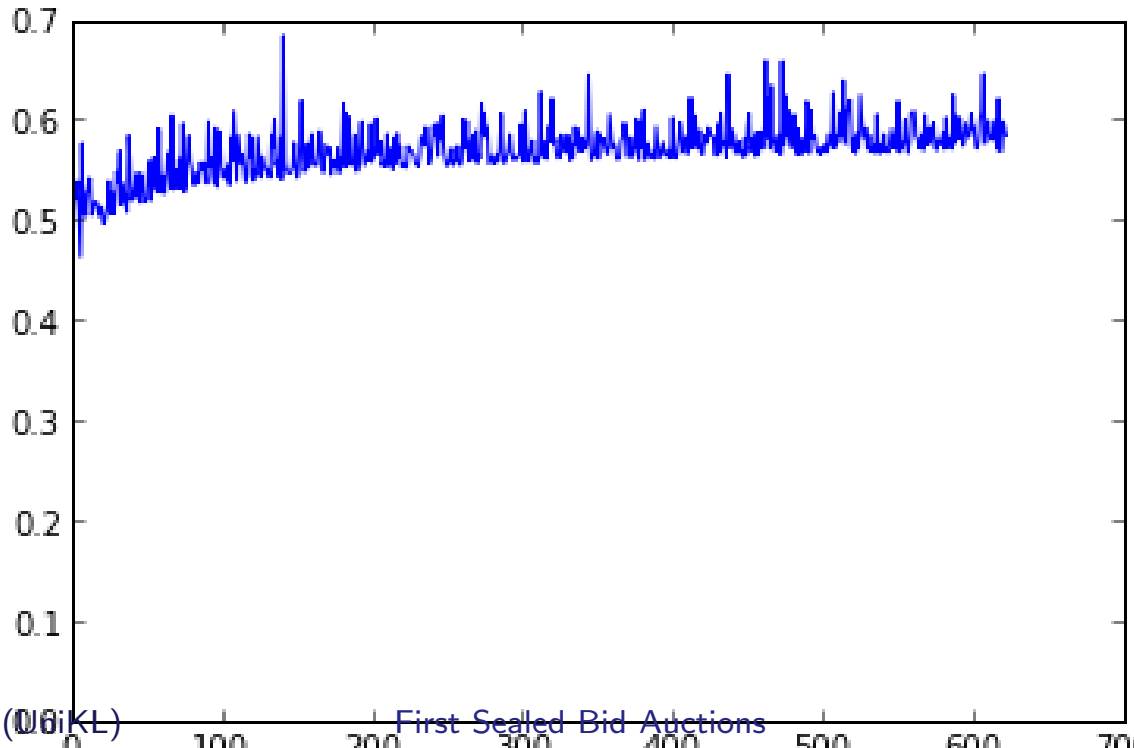
```
for fraction in linspace(0.1,1.1,11):
    participants = [FSBidder(fraction=fraction),FSBidderWithStats1(
        fraction=0.05)]
    auctioneer = FSBAuctioneer(participants,uncertainty=0.05)
    print fraction,
    print "  %.2f %.2f"%auctioneer.gains(),
    print "  %.3f"%mean(auctioneer.participants[-1].fractions)
```

```
0.1   5.71 87.45   0.114
0.2   5.82 75.72   0.227
0.3   6.45 63.66   0.338
0.4   4.05 53.91   0.454
0.5   4.16 42.56   0.564
...
0.8   1.12 9.93   0.908
0.9   1.43 2.91   0.990
1.0   -1.20 0.16   1.030
1.1   -10.57 -0.04   1.106
```

plot of adaptation

```
1 participants = [FSBidder(fraction=0.5),FSBidderWithStats1(fraction
      =0.05)]
2 auctioneer = FSBAuctioneer(participants,uncertainty=0.05)
3 print auctioneer.gains()
4 plot(participants[-1].fractions)
```

```
(3.0639228444711732, 43.028826531685169)
```

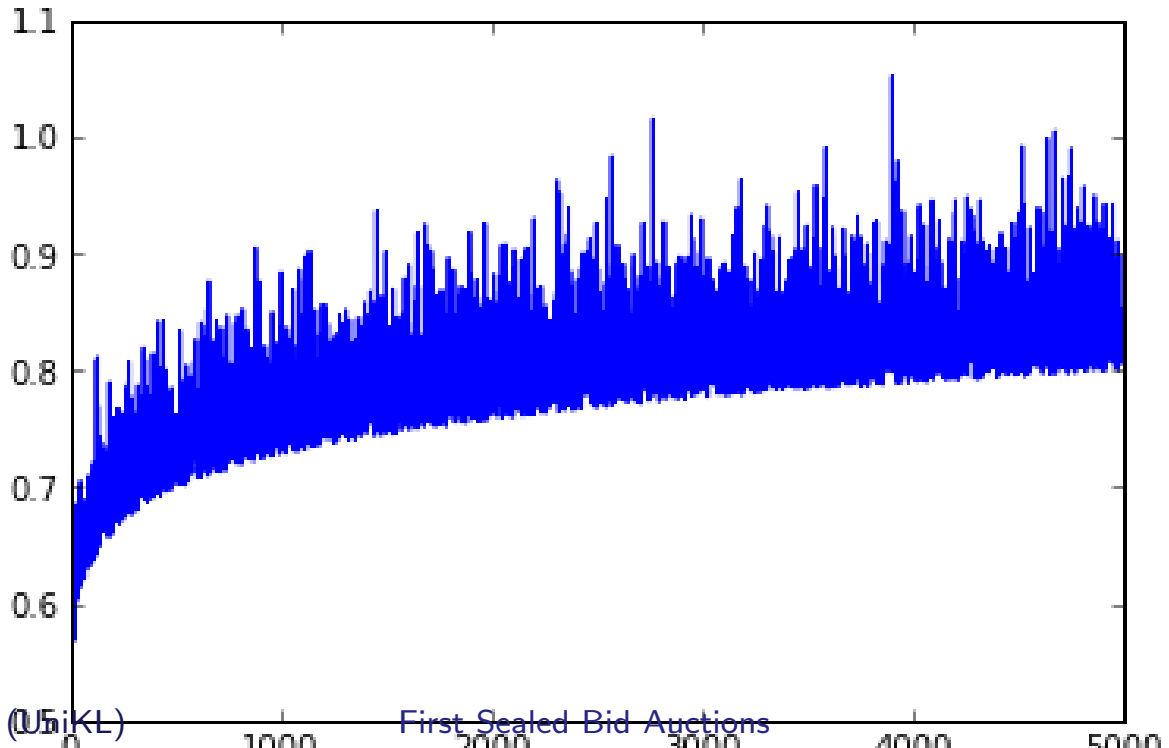


Competing Adaptive Strategies

competing adaptive strategies

```
1 participants = [FSBidderWithStats1(fraction=0.5),FSBidderWithStats1
      (fraction=0.5)]
2 auctioneer = FSBAuctioneer(participants,uncertainty=0.05)
3 print auctioneer.gains()
4 plot(participants[-1].fractions)
```

```
(11.678262824813881, 11.764510709039063)
```



```
class FSBidderWithRecentStats:
    def __init__(self,fraction=0.5,n=100):
        self.balance = 0
        self.value = 0
        self.fraction = fraction
        self.fractions = [fraction]
        self.n = n
    def giveEstimate(self,estimate):
        self.estimate = estimate
    def solicitBid(self):
        return self.estimate * clip(mean(self.fractions[-self.n:])
            ,0.0,0.95)
    def gain(self):
        return self.value+self.balance
    def winningBid(self,paid,winner):
        self.fractions.append(paid/self.estimate)
```

```
for i in range(10):
    f = rand()
    participants = [FSBidderWithStats1(fraction=f),
        FSBidderWithRecentStats(fraction=0.5,n=100)]
    auctioneer = FSBAuctioneer(participants,uncertainty=0.05)
    print f,auctioneer.gains()
```

```
0.298958784683 (2.3599382304020242, 10.496134879240977)
0.174149108002 (2.5326416900687456, 12.742573483894661)
0.0434126693325 (2.5829150443016986, 12.15148132157023)
0.264893924412 (2.559803929331872, 11.7563266189927)
0.392771062266 (2.5391035587089252, 11.664054523601465)
...
0.738000964179 (1.4617586328497738, 1.4259323777246988)
0.362883017765 (2.4808759641525731, 10.361542634060955)
0.535201917114 (2.3544250472773798, 3.7183754099088024)
0.31314846005 (2.5217747608623133, 12.720551912219635)
```

```
for i in range(10):
    participants = [FSBidderWithRecentStats(n=1000),
        FSBidderWithRecentStats(n=100)]
    auctioneer = FSBAuctioneer(participants,uncertainty=0.05)
    print auctioneer.gains()
```

```
(2.7434600125520783, 6.6032355295393383)
(2.5811071053871535, 6.5731864326515934)
(2.745010668207315, 6.3405336787135225)
(2.5540896005935094, 6.4856089587364112)
(2.5905043973753403, 6.6829001796019494)
...
(2.8996478737312135, 6.9960880495022399)
(2.7584225132618334, 6.4864116822432143)
(2.7391120390326598, 6.9330406621626928)
(2.679416773950722, 6.4994738174572584)
```

Questions / Exercises

Analysis of Non-Iterated Common Value Auctions

We considered first-sealed-bid common value auctions without iteration above (that is, a series of auctions with two particiants that don't know each other and have no memory).

- Enumerate the different parameters and assumptions describing those auctions.
- Is a bid proportional to the estimate value the most general useful fixed strategy?
- If not, are there better fixed strategies (where a strategy is a function `solicitBid(estimate)`)
- Create an agent that attempts to automatically determine the best fixed strategy for a given set of auction parameters.

Additional questions to think about / experiment with:

- Note that not only the parameters of the value distribution can be chosen, but the value distribution itself.
- What happens when we pick a uniform value distribution? A Cauchy value distribution?
- What happens when we limit values and bids to be non-negative? Do negative values make sense?

Agent-Based Nash Equilibrium

Assuming the simple `FSBidder` classes, write code that tries to approximate a Nash equilibrium for a given auction and set of auction parameters. That is, it should determine the fixed fractional bid for each bidder for which equilibrium is reached. Assume that the solution is symmetric (i.e., that it's the same fraction for both bidders).

- Your code need not be efficient; you can run many simulations.
- You do have to deal with the fact somehow that the estimates you get out of simulations are noisy.

Non-Iterated Private Value Auctions

Repeat the above analysis for first-sealed-bid auctions, but with *private values*. That is, instead of sampling a common value for all the participants and rewarding participants based on that common value, draw a separate value for each participant from the distribution. As before, clearly lay out hypotheses and test them with agent-based models. Present your analysis in the form of a worksheet.

Your answer need not be as detailed as your analysis for common value auctions, but try to come up with results that provide useful insights into private value auctions through the use of agent-based models.