Figgie Bot
Final Presentation

What is "Figgie"

Figgie is a game

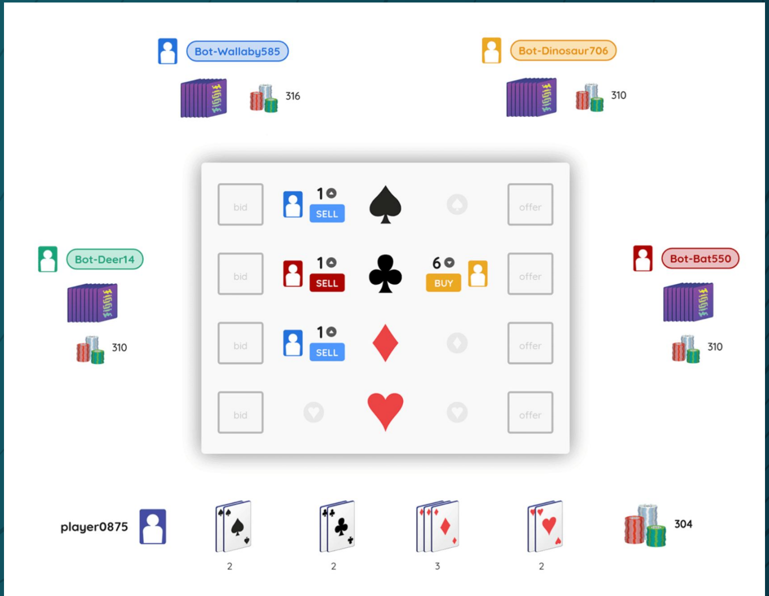
- Developed at Jane Street
- Meant to simulate live floor trading
- Imperfect information
  - (Opponents cards are hidden)

4 or 5 people

10 or 8 cards per person

50 or 40 dollar per person

$ $ $ $ $



<u>Key:</u> find out the **goal suit** (the suit of same color with 12 card suit)

<u>Award:</u>

- Each player get $10 for each card from goal suit
- Winner (with the most card from goal suit) gets the rest of the pot

Breakdown of project

1. **Agent**
   a. Develop agent to play the game
2. **Game Engine**
   a. System to run play the game
   b. Allow people and agents to connect
3. **UI**
   a. UI to interact with the game as a user

Agent

Initial Approach

- ReBeL Algorithm
  - AI bot that excels at imperfect-information games, specifically poker and Liar's Dice
  - Uses RL and Search
  - Created by Facebook AI researchers

Complications

- Difficult to adapt to Figgie, as opposed to poker
- Uses many advanced RL techniques

Algorithm 1 ReBeL: RL and Search for Imperfect-Information Games

function $\text{SELFPLAY}(\beta_r, \theta^v, \theta^\pi, D^v, D^\pi)$ ▷ β_r is the current PBS
  while $!\text{ISTERMINAL}(\beta_r)$ **do**
    $G \leftarrow \text{CONSTRUCTSUBGAME}(\beta_r)$
    $\bar{\pi}, \pi^{t_{\text{warm}}} \leftarrow \text{INITIALIZEPOLICY}(G, \theta^\pi)$ ▷ $t_{\text{warm}} = 0$ and π^0 is uniform if no warm start
    $G \leftarrow \text{SETLEAFVALUES}(G, \bar{\pi}, \pi^{t_{\text{warm}}}, \theta^v)$
    $v(\beta_r) \leftarrow \text{COMPUTEEV}(G, \pi^{t_{\text{warm}}})$
    $t_{sample} \sim \text{unif}\{t_{\text{warm}} + 1, T\}$ ▷ Sample an iteration
    for $t = (t_{\text{warm}} + 1)..T$ **do**
      if $t = t_{sample}$ **then**
        $\beta'_r \leftarrow \text{SAMPLELEAF}(G, \pi^{t-1})$ ▷ Sample one or multiple leaf PBSs
      $\pi^t \leftarrow \text{UPDATEPOLICY}(G, \pi^{t-1})$
      $\bar{\pi} \leftarrow \frac{t}{t+1}\bar{\pi} + \frac{1}{t+1}\pi^t$
      $G \leftarrow \text{SETLEAFVALUES}(G, \bar{\pi}, \pi^t, \theta^v)$
      $v(\beta_r) \leftarrow \frac{t}{t+1}v(\beta_r) + \frac{1}{t+1}\text{COMPUTEEV}(G, \pi^t)$
    Add $\{\beta_r, v(\beta_r)\}$ to D^v ▷ Add to value net training data
    for $\beta \in G$ **do** ▷ Loop over the PBS at every public state in G
      Add $\{\beta, \bar{\pi}(\beta)\}$ to D^π ▷ Add to policy net training data (optional)
    $\beta_r \leftarrow \beta'_r$

Source: https://arxiv.org/pdf/2007.13544.pdf

Followed Approach: Discrete-Event Simulation

- Use expected value of trades on order book to evaluate possible trades
  - Two methods to find expected value:
    - "Fundamentalist," and "Bottom Feeder"

Fundamentalist — Card Counting

- 12 possible decks
- Card counting determines probability of each deck
- Uses this to determine buy and sell prices

Algorithm 3: Card-counting method for asset j, returning a list of known asset cards each agent holds

```
Set n to how many units of asset j you are initially dealt
Initialize a 4 element list L, such that for x = self.num we set L[x] = n, and all other elements of L
  are 0
Let T be the list of trades for asset j, ordered by time
for i ← 0 to |T| − 1 do
    /* Since T only grows, in practice we can store the data persistently and just
       run the for-loop for the new trades in T.                                  */
    Let b be the buyer in trade T[i], and s be the seller
    Let v be the volume
    if L[s.num] < v then
        Add v to L[b.num]
        Set L[s.num] to 0
    else
        Add v to L[b.num]
        Subtract v from L[s.num]
Return L
```

Fundamentalist – Trading

- Uses expected values of trades in order-book to make decisions on trades
- Attempts to maximize “goal suit” by buying and selling cards with highest expected returns

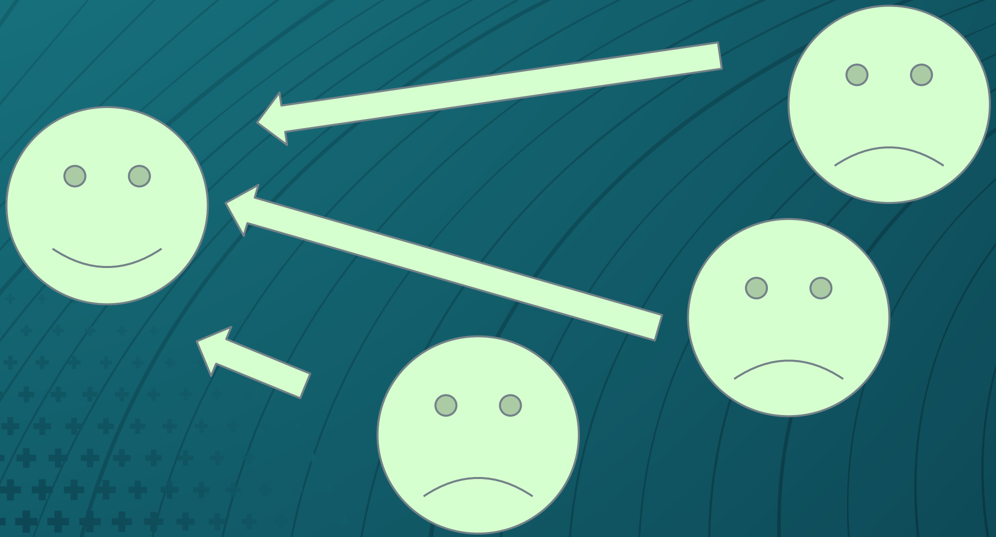
Algorithm 4: Fundamentalist trading algorithm for asset j

Let m be the multinomial distribution obtained by card counting and the methods detailed above
Initialize a two element list $L = [e_b(j, j_n, m), e_s(j, j_n, m)]$
Let O_b and O_s be the agent's own order books for buy and sell orders respectively for asset j
for $i \leftarrow 0$ **to** $|O_b| - 1$ **do**
    Let o be the order $O_b[i]$
    if $o.price > L[0]$ **then**
        Let o.deleted be $True$
for $i \leftarrow 0$ **to** $|O_s| - 1$ **do**
    Let o be the order $O_b[i]$
    if $o.price < L[1]$ **then**
        Let o.deleted be $True$
Return L

$$p_j^* = \frac{1}{|A|}\sum_{i\in A}\frac{1}{2}\left(\frac{1}{k}\sum_{p_b\in B_{ji}(k)} p_b + \frac{1}{k}\sum_{p_s\in S_{ji}(k)} p_s\right),$$

Bottom-Feeder

Outputting expected value based on past order history



Order Sending

- Send orders based on expected values and market prices

Algorithm 2: Order-sending based on expected value for asset j

Get two expected values p_b and p_s, for buying and selling asset j respectively.
Randomly choose to buy or sell, with probability 0.5
if *buying* **then**
    Get a price p according to $\text{Uniform}(0, p_b)$
    Let s be the lowest sell order price in the market for asset j
    Send a limit buy order for asset j with price $\min(p, s)$
else
    Get a price p according to $\text{Uniform}(p_s, 2p_s)$
    Let b be the highest buy order price in the market for asset j
    Send a limit sell order for asset j with price $\max(p, b)$

Random Agent

- Randomly places a bid or offer every few seconds
- Mostly used for testing

```
Successfully added the order to the database.
Player Random Player offers 14 for clubs.
Successfully added the order to the database.
Player Random Player bids 4 for hearts.
Successfully added the order to the database.
Player Random Player bids 2 for diamonds.
Successfully added the order to the database.
Player Random Player offers 7 for clubs.
```

Human Player - Command Line Interface

- Parse & interpret input, make corresponding action
- Print current state in a readable way
  - Use pretty printer
- Limitation: Can’t flush user input, user has to manually fetch

```
Time remaining: 180
Your current hand:
- hearts: 2
- diamonds: 4
- clubs: 2
- spades: 1
Current players:
- Human Player James, balance: 308
- Random Player, balance: 292
Current Order Book:
- Random Player bids diamonds at price 5
- Random Player bids spades at price 10
- Random Player offers hearts at price 12
- Human Player James offers diamonds at price 2
Make an action (type h or help for help): f
```

Controller - Linking Agent and Backend

- Each agent has their own websocket
- Agents pass in their websocket as parameter
- Controller performs action
  - Add players
  - Start a round
  - Place/accept/cancel orders
  - Get game updates

Game Engine

WebSocket

- Essential for multiplayer games
- Two-way connection between user and server
- Broadcasts game updates to user

Database

- NoSQL structure
  - Flexible and scalable
- Logs game data
  - Can be used for future analysis or model training

Testing

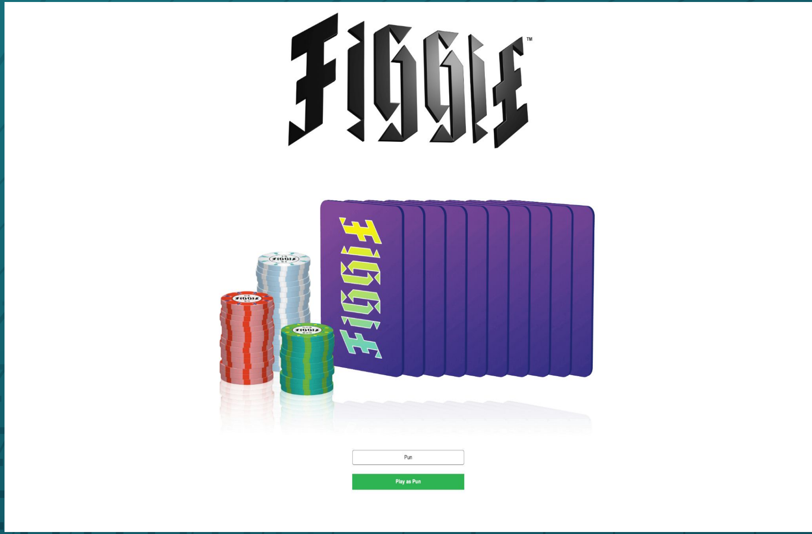
- Benefits
  - Code quality
  - Reliability
  - Saves time

```python
def test_place_order(self):
      # successful bid
      self.assertEqual(get_book()["bids"]["clubs"].order_id, EMPTY_BID.order_id)
      place_order("1", True, "clubs", 5)
      self.assertEqual(get_book()["bids"]["clubs"].player_id, "1")

      # unsuccessful bid
      place_order("2", True, "clubs", 1)
      self.assertEqual(get_book()["bids"]["clubs"].player_id, "1")

      # successful offer
      self.assertEqual(get_book()["offers"]["clubs"].order_id, EMPTY_OFFER.order_id)
      place_order("1", False, "clubs", 5)
      self.assertEqual(get_book()["offers"]["clubs"].player_id, "1")

      # unsuccessful offer
      place_order("2", False, "clubs", 7)
      self.assertEqual(get_book()["offers"]["clubs"].player_id, "1")
```

User Interface

Login Page

- Quick UI using React.js



Key Components

- Game can start when 4 players have connected
- View highest bid/lowest offer
- Place/Accept bids and offers
- View balance of each suit

Game Page - Demo

Conclusion

What we learned

- Figgie
- Full-stack applications
- When to scale-back overly ambitious goals

Next steps

- Deployed backend
  - Clients from different computers
- More advanced agents
  - Learn parameters
  - Imitation Learning
  - Full E2E Deep Model
- Cleaner Frontend UI

Questions?