Business War Policy Exploration

Ren Hui $\star^1$
ShanghaiTech University
Shanghai, China
renhui@shanghaitech.edu.cn

Xia Kangjie $\star^2$
ShanghaiTech University
Shanghai, China
xiakj@shanghaitech.edu.cn

Lei Huang $\star^3$
ShanghaiTech University
Shanghai, China
huanglei@shanghaitech.edu.cn

Abstract

This paper investigates the strategic interactions between two restaurants that compete for a fixed number of customers in a region. We assume that the restaurants have the same service quality and can only compete on prices. We develop a game-theoretic model that incorporates customer preferences, forgetfulness, and information about the rivals situation. We examine how different models and parameter conditions affect the outcomes of the price war. We compare the pricing strategies, forecasting models, initial funds, and price positioning of the restaurants and their impacts on the market dynamics and welfare. We illustrate our results with graphs and tables and discuss their implications for the restaurant industry and policy makers.

1 Introduction

1.1 Background

The restaurant industry is a highly competitive and dynamic sector in the economy. Restaurants face fierce competition from existing and potential rivals, as well as changing consumer tastes and preferences. To survive and thrive in this environment, restaurants need to constantly innovate and adapt their strategies to attract and retain customers.

One of the most common and effective ways for restaurants to compete is through price wars. Price wars are situations where two or more firms repeatedly lower their prices to gain market share and

$\star$ All authors contribute to this work equally.
1 Code further framework, Approximate Q-Agent, MCQ Agent, Manual Agent, argument command, report-Reinforcement Learning.
2 Code base framework, Expectimax Agent, AlphaBeta Agent, visualization, data excel, presentation slides, report-Introduction, Search tree based agents and Conclusion.
3 Neural Network Pridiction Agent, multiThreading test, presentation speech, report-Neural network Expectimax search algorithm.

deter entry by new competitors. Price wars can have significant effects on the profitability and welfare of the firms and consumers involved, as well as on the overall efficiency and stability of the market. For one restaurant, how to make choice between making sacrifice and making a low price to win over consumers, or making a high price to make more money while loss some preferred consumer? This is a interesting problem.

In this paper, we focus on a simple but realistic scenario where there are only two restaurants in a region, and they have the same service quality. The region has a fixed number of regular customers, each of whom can choose one of the two restaurants to eat. The restaurants know the exact preferences of each customer. The restaurants engage in a price war to compete for customers, maximize their profits, and prevent bankruptcy.

1.2 Market Model

We model the market as follows:

1. Customer acquisition rules

   Each day, each customer enters one of the two restaurants randomly, based on factors such as queue length, restaurant preparation, eating time, customer friends, customer mood, etc. However, customers can switch to another restaurant according to their preferences.

2. Customer preferences

   For the two restaurants A and B, each customer has a preference value for each restaurant, denoted by p_A and p_B. The preference value can be positive, negative or zero, and it reflects how much the customer likes or dislikes the restaurant. The customer's preference state can be one of the following:

   - Prefer A ($p_A > p_B$)
   - No preference ($p_A = p_B$)
   - Prefer B ($p_A < p_B$)

3. Perfect price discrimination

   Each restaurant can offer a specific price to each customer, depending on their preference state and its own strategy. The price can be one of the following levels:

   - High price. This will reduce the customer's preference value for that restaurant by a large amount ($p'_X = p_X + M - H$, where M, H is a constant).
   - Medium price. This will maintain the customer's original preference value for that restaurant ($p'_X = p_X$).
   - Low price. This will increase the customer's preference value for that restaurant by a small amount ($p'_X = p_X + M - L$, where M, L is a constant).
   - Super-low price. This will increase the customer's preference value for that restaurant by a large amount ($p'_X = p_X + M - S$, where M, S is a constant).

4. Customer comparison and being "slaughtered"

   After a customer enters a certain restaurant X according to the customer acquisition rules, he or she will react according to the price offered by X:

   State: No preference or prefer X:

   - The customer will choose to eat at X, and update his or her preference value for X according to the price given by X.

   State: Prefer another restaurant Y:

   - If X offers a low price or super-low price, the customer will choose to eat at X and update his or her preference value for X according to X's given price.
   - If X offers a high price or medium price, the customer will choose to eat at Y.

   Notice: Whatever the consumer decide to eat at a restaurant or not, his preference will always update according to the given price by sellers.

5. Daily update

   - For sellers, they have daily income and cost, which will update their balance at the end of each day.

6. End conditions We have two conditions to determine if the war should come to the end:
    - When the balance of the player seller or all of the rivals come to non-positive(≤ 0), the game ends.
    - When the number index of the day reach 100, the game ends. This is to prevent a single business war continuing for too long.
7. Victory condition

    The player seller wins the war if the game ends and the following two conditions are met:
    - The player seller's balance is positive(> 0).
    - The player seller's final balance is greater than or equal to the highest balance among the rivals.

    If the player seller does not meet these conditions, he or she loses the war.

1.3 Important Assumptions

We use a game-theoretic framework to model the strategic interactions between the two restaurants over time. We assume that customer preferences are influenced by past experiences and decrease by 10% every day (simulating forgetfulness). We also assume that each restaurant can obtain some information about the economic situation of the other restaurant by some means (such as word-of-mouth, advertising, or even spying).

1.4 Research Methods

We study the outcomes under different models and parameter settings. Our model employs different strategies to analyze how restaurants set their prices in each period, based on customers' exact preferences for restaurants and expectations of competitors' prices. We compare the average outcomes of the price war under different assumptions about the pricing strategies, expectation models, initial funds, and price levels. We plot these outcomes as graphs with time as the horizontal axis and funds as the vertical axis, using the method of controlled variables. From this, we draw conclusions and compare the differences between the various models. We also discuss the implications of our findings for the restaurant industry and policy makers.

2 Rival Agents

2.1 Default Agent

We design four default strategies, namely random strategy and greedy strategies. Among them, the greedy strategies are divided into high price, low price and super low price strategy. These four agents are implemented as rival agents for testing different methods.

The reason why We do not design a greedy medium strategy that always bids the medium price is that it would never make any profit. The medium price is equal to the average cost of the agent, so bidding the medium price would only break even at best. Therefore, a greedy medium strategy has no discussion value.

2.1.1 Random Agent

The random agent is a strategy to randomly select a price from the price range, that is, the probability of choosing high price, medium price, low price and super low price is 0.25.

2.1.2 Greedy High Agent

The greedy high agent is a strategy to always choose the highest price from the price range.

2.1.3 Greedy Low Agent

The greedy low agent is a strategy that pays a high price when consumers prefer to it, and always pays a low price otherwise.

2.1.4 Greedy Super Low Agent

The greedy super low agent is a strategy that pays a high price when consumers prefer to it, and always pays a super low price otherwise.

3 Search tree based agents

3.1 Expectimax Agent

In this section, we explore how to design an optimal strategy for a business in a competitive market, assuming that we know the opponents strategy. We use the Expectimax Agent, which is a type of decision-making agent that maximizes the expected utility of each action.

3.1.1 Expectimax Search Tree

The Expectimax Agent builds a search tree to evaluate each possible state of the game. The search tree has two types of nodes: max nodes and chance nodes. Max nodes represent the states where the agent chooses an action, and chance nodes represent the states where the opponent or a random event affects the outcome. The agent assigns a utility value to each leaf node using an evaluation function, and then propagates these values up the tree using the expectimax algorithm. The algorithm works as follows:

- At a max node, the agent chooses the action that has the highest utility value among its children.
- At a chance node, the agent computes the expected utility value of each child, weighted by its probability, and then sums them up.

The agent selects the action that corresponds to the roots best child. Figure 1 illustrates an example of an expectimax search tree.

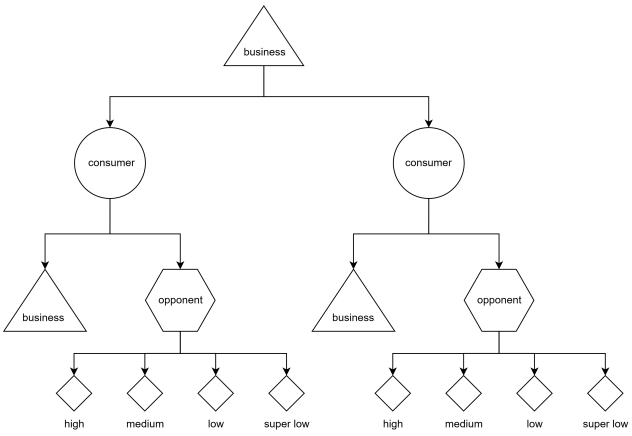


Figure 1: **Expectimax Search Tree**

3.1.2 Evaluation Functions

We tested four different evaluation functions to measure the utility of each state for the agent. The evaluation functions are based on different features of the state, such as:

1. The agents own assets.
2. The difference between the agents and the opponents assets.
3. The difference between the agents assets and an estimate of the opponents assets, with a Gaussian error of 0.05 standard deviation of the total assets.

4. The agents assets plus $0.5\times$the sum of all consumers preferences. This function also considers the market demand and adjusts the weight of the two features with a coefficient of 0.5.

We compared the average winning rate of each evaluation function against various opponents, as shown in Figure 2. The results show that evaluation function 4 performs the best, followed by 2, 3 and 1.

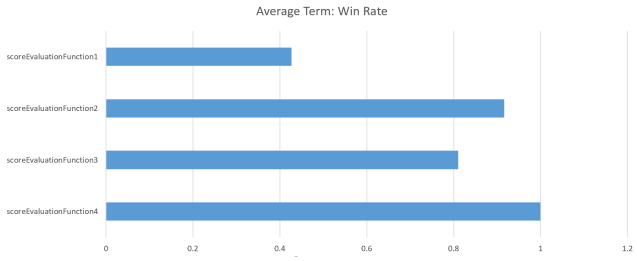


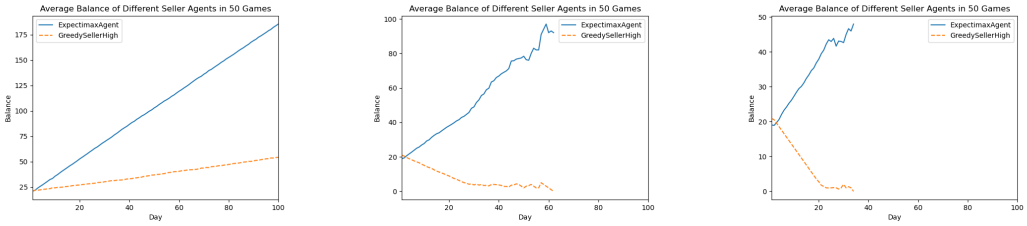
Figure 2: **Average Winning Rate of Different Evaluation Functions**

Evaluation function 4 outperforms the others because it takes into account both the agents profit and the consumers preferences, which are essential for the agents success. It also balances the weight of these two features with a coefficient of 0.5. Evaluation functions 2 and 3 rank second and third because they account for the difference between the agents and the opponents assets, which indicates the market competition. However, evaluation function 3 suffers from a Gaussian error that reduces its accuracy and reliability. Evaluation function 1 performs the worst because it only relies on the agents own assets, which neglects the market demand and competition.

3.1.3 Depth Influence

We also investigated how increasing the search depth affects the performance of the agent. We found that increasing the search depth does not significantly change the average winning rate, but it decreases the final balance.

To illustrate this effect, we used evaluation function 4 (which is the best function we found) and plotted the final balance against time for different search depths in Figure 3. The slope of each curve represents the amount of money earned per unit time by the agent. The simulation results show that as the search depth increases, the agent tends to earn less money but bankrupts the opponent faster.



(a) **k = 1.6(depth = 1)** (b) **k = 1.0(depth = 2)** (c) **k = 0.6(depth = 3)**

Figure 3: **Final Balance of Expectimax Agent with Evaluation Function 4 against GreedyHigh**

This result can be explained by the fact that increasing the search depth makes the agent more farsighted and aggressive, but also more conservative and risk-averse. The agent can foresee the opponents moves and plan ahead to win the game, but it also sacrifices some short-term profit to avoid losing in the long run. Therefore, increasing the search depth may not always be beneficial for the agent, depending on the situation and the goal.

3.2 AlphaBeta Agent

Before we discuss how to predict the opponents behavior, we tried a new model: alpha agent, that is, it is assumed that the opponent always chooses the best price.

3.2.1 AlphaBeta Pruning

The Expectimax Agent can be very computationally expensive, especially if the search tree is deep and branching. AlphaBeta Pruning is an optimization technique that eliminates parts of the tree that are provably irrelevant. It keeps track of two values, alpha and beta, that represent the best and worst utility values for the agent and the opponent at any node. The algorithm works as follows:

- At a max node, alpha is updated to be the maximum of the current alpha and the best childs utility value. If alpha is greater than or equal to beta, the remaining children are pruned, since they cannot improve the utility value for the agent.

- At a min node, beta is updated to be the minimum of the current beta and the best childs utility value. If beta is less than or equal to alpha, the remaining children are pruned, since they cannot decrease the utility value for the max node above.

The agent selects the action that corresponds to the roots best child. Figure 4 illustrates an example of an alphabeta search tree with pruning (where upper triangles represent max nodes and lower triangles represent min nodes).

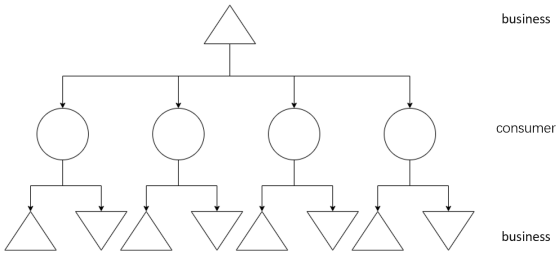
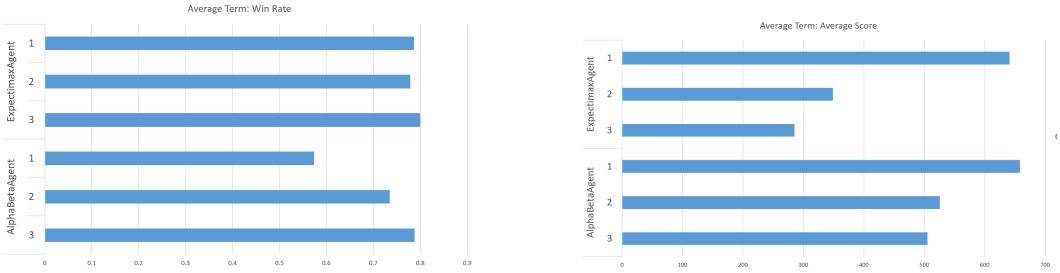


Figure 4: **AlphaBeta Search Tree**

3.2.2 Comparison with Expectimax Agent

We compared the performance of the AlphaBeta Agent with the Expectimax Agent. We controlled the variable of the search tree depth, which is the y-axis of the figure. We found that the AlphaBeta Agent performs better than the Expectimax Agent in terms of average score, but worse in terms of average winning rate, as shown in Figure 5. This is because the AlphaBeta Agent uses pruning technique, which can ignore some unnecessary nodes, thereby improving the search efficiency, but also possibly missing some favorable nodes. While the Expectimax Agent considers all possible nodes and calculates their weighted average, thus closer to the true expected value.

Moreover, Figure 5 also shows the trend of the AlphaBeta Agents performance as the search tree depth increases. The average winning rate increases, while the average score decreases. This is because as the search tree depth increases, the AlphaBeta Agent can see more moves ahead and avoid losing situations, which improves its winning rate. However, as the search tree depth increases, the AlphaBeta Agent also becomes more conservative and avoids risky moves that could lead to high scores, which lowers its average score.



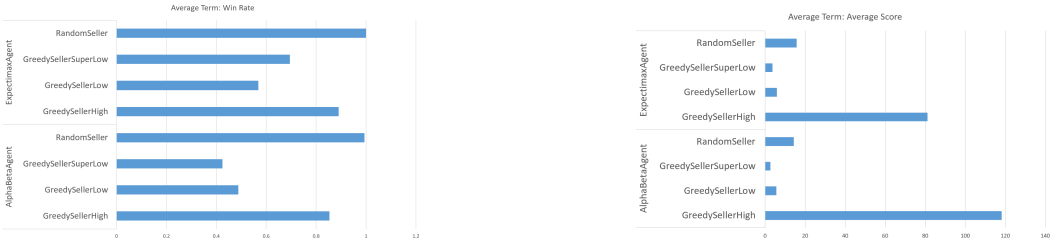
(a) **Average Win Rate** (b) **Average Score**

Figure 5: **Comparison with Expectimax Agent**

3.3 Comparison of Expectimax Agent and AlphaBeta Agent with Different Rival Agents

3.3.1 Overall Performance

To evaluate the effectiveness of Expectimax Agent and AlphaBeta Agent, we test them against four different rival agents with varying levels of intelligence and strategy: Random Agent, Greedy High Agent, Greedy Low Agent, and Greedy Super Low Agent. We use two metrics to measure the performance: average win rate and average score. The average win rate is the ratio of games that the agent wins out of 50 games. The average score is the mean of the final balance that the agent gets in each game. Figure 6 illustrates the outcomes of our comparison.



(a) **Average Win Rate** (b) **Average Score**

Figure 6: **Overall Performance of Expectimax and AlphaBeta With Different Rival Agents**

3.3.2 Difference in Random Agent

Both Expectimax Agent and AlphaBeta Agent can easily defeat the Random Agent, as they have similar average scores and win rates, as shown in Figure 6. This is because the Random Agent does not follow any strategy or preference in choosing its actions, so it acts randomly and unpredictably. This makes it a weak and unreliable opponent for either agent.

3.3.3 Difference in Greedy High Agent

AlphaBeta Agent can achieve a higher average score than Expectimax Agent, but with a slightly lower win rate. Therefore, AlphaBeta Agent is more suitable when facing a Greedy High Agent. This is because the Greedy High Agent always chooses the action that maximizes its immediate reward, without considering the future consequences. This makes it greedy and short-sighted, so it can be easily exploited by AlphaBeta Agent's pruning technique, which can avoid its traps and take advantage of its mistakes. On the other hand, Expectimax Agent does not prune any branches, so it explores all possible actions equally. This may lead to some unnecessary computation and suboptimal decisions.

3.3.4 Difference in Greedy Low Agent

Expectimax Agent has a slightly higher win rate than AlphaBeta Agent, but with the same average score. Therefore, Expectimax Agent has a slight advantage when facing a Greedy Low Agent.

This is because the Greedy Low Agent sometimes chooses the action that minimizes its immediate loss, without considering the future opportunities. This makes it cautious and risk-averse, so it can be sometimes avoided by Expectimax Agent's expected value technique, which can estimate the probability and utility of each action based on all possible outcomes. AlphaBeta Agent, on the other hand, may prune away some branches that have low immediate value but high future value, based on its alpha-beta values. This may lead to some missed opportunities and suboptimal decisions.

3.3.5 Difference in Greedy Super Low Agent

Expectimax Agent has a higher win rate and average score than AlphaBeta Agent. Therefore, Expectimax Agent is more suitable when facing a Greedy Super Low Agent. This is because the Greedy Super Low Agent always chooses the action that minimizes its immediate loss by a large margin, without considering the future opportunities. This makes it very conservative and passive, so it often misses out on good actions that could improve its situation. Expectimax Agent can take advantage of this by estimating the expected value of each action based on a probability distribution over all possible outcomes, and choosing the action that maximizes its expected utility. AlphaBeta Agent, on the other hand, may be too optimistic or pessimistic about some actions based on its alpha-beta values, and miss some better actions.

3.3.6 Trend in Average Score

The average score of the agent depends on the distribution of customers price tolerance and the opponents bidding strategy. When the opponent bids higher, the agent can make more money by attracting more premium customers who have higher price tolerance. However, when the opponent bids lower, the agent has to lower its bid as well to compete for customers. This results in a lower average score for both agents.

As shown in Figure 7, the Expectimax agents optimal strategy is to bid lower when facing a low-bidding opponent. This leads to a situation where both agents are competing for customers by lowering their prices. Therefore, the Expectimax agent cannot make much money in this scenario.

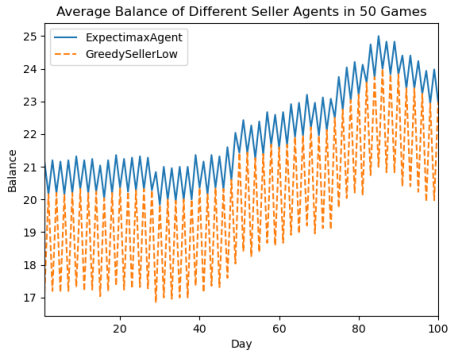


Figure 7: **Competing for customers**

However, if both agents cooperate and bid higher, they can increase their average score by sharing the market and avoiding price wars. This is a more efficient and sustainable outcome for both agents. Figure 8 shows how the average score of both agents increases when they cooperate and bid higher.

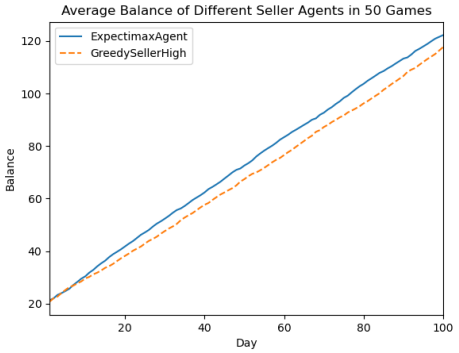


Figure 8: **Cooperating for mutual benefit**

3.4 Conclusion of Search Tree Based Agents

Based on our comparison, we conclude that Expectimax Agent and AlphaBeta Agent have different strengths and weaknesses when facing different rival agents. Expectimax Agent is more robust and flexible, as it can handle uncertainty and randomness better than AlphaBeta Agent. However, Expectimax Agent is also more computationally expensive, as it does not prune any branches in its search tree. AlphaBeta Agent is more efficient and aggressive, as it can prune away branches that are not worth exploring based on its alpha-beta values. However, AlphaBeta Agent is also more sensitive and brittle, as it may make wrong decisions based on inaccurate or incomplete information. Therefore, the choice of which agent to use depends on the characteristics and preferences of the rival agent, as well as the time and space constraints of the game.

4 Learning Opponent's Strategy without prior knowledge

A key assumption in our previous analysis is that we know the opponents strategy in advance. However, this is rarely the case in real-world scenarios, where the agent can only observe the opponents actions and infer its strategy. Therefore, a crucial challenge for our agent is to predict the opponents behavior based on limited observations online. An then use this model to make better decisions and improve its performance. In this section, we will propose a multilayer perceptron approach to learn the opponents strategy from its actions online, and show how our agent can use this approach to adapt to different opponents.

4.1 Observation of opponent's action

A observation $(\mathbf{x}, y)$. $\mathbf{x}$ is composed of (i) costumer's preference $\mathbf{p}$ and (ii) rival's balance b. $\mathbf{x}$ is the concatenation of $\mathbf{p}$ and b. And y is the action the rival take regarding this costumer.

4.2 Neural-network-expectimax search algorithm

Algorithm 1 A framework for neural-network-expectimax search algorithm

Require: $N(\mathbf{x})$: A 3-layer perceptron , initialized empty, returns opponent's predicted action given costumer's information $\mathbf{x}$.
Require: $\mathcal{X}$: A Data set of opponent's behaviors, initialized empty.
Require: $f(N, \mathcal{X}) = N'$: A proper perceptron online training update algorithm.
Require: $\sigma = search(belief = N)$: A multiagent search function returns speculated best action σ under the prior belief N the opponent's speculated behaviours.
while not GameOver do
  $(\mathbf{x}, y) \leftarrow$ newly observed opponent's behavior ▷ Observe the opponent's action in reality
  $\mathcal{X} \leftarrow \mathcal{X} \cup \{(\mathbf{x}, y)\}$ ▷ Add to data set
  $N \leftarrow f(N, \mathcal{X})$ ▷ Online Training
  $myBestAction \leftarrow search(belief = N)$ ▷ Find the best action using multiagent search
  Perform $myBestAction$ ▷ Perform the best action in reality
end while

The algorithm 1 is an online algorithm that tries to capture the opponent's policy using a 3-layer network. Note that algorithm 1 is a online algorithm.

4.3 Experiment

For simplicity, we choose to make the online update algorithm as simple as possible. $f(N, \mathcal{X})$ simply means training a new network from the data set $\mathcal{X}$. Further advancement can be made but not discussed in this article. So there is no need to discuss with the online learning loss descent.

Compared with baseline (randomly guess component's behavior), our algorithm achieves good results.

Figure 9: neural-network-expectimax search algorithm score(mid of every part)

5 Reinforcement Learning

For Reinforcement Learning, we define the transition as from last state that one consumer come to the player seller and the player seller has the chance to give out one price to the next state that he again has the chance.

One state contains the information about who is current consumer, the preferences of each consumer, the balance of each seller and the number index of the day.

For the preferences, since they are vectors of floats, we take round of each element as the state recorded to prevent the state space become too large. Furthermore, we just record the difference between the balance of each seller and the balance of the player seller, as the preference is relative and this can also reduce the state space.

However, the state space is still very large. Consider a simple situation, suppose there are two sellers and two consumers, the balance of each seller $\in [0, 100)$, preference difference of each consumer $\in (-50, 50)$, balance and preference change steps are multiplies of 2, and there are 100 days, then the size of state space is already $(100/2)^2 * (100/2)^2 * 100 * 2 = 1.25 * 10^9$
So we mainly focus on how to approximately get the Q-value.

5.1 Approximate Q-Learning

Recall the method of Approximate Q-learning:

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + ... + w_n f_n(s,a)$$

$$\text{difference} = (r + y * \max_{a'} Q(s', a')) - Q(s,a)$$

$$w_i = w_i + \alpha * \text{difference} * f_i(s,a)$$

Where $f_i(s,a)$ is the feature function, w_i is the weight of each feature, α is the learning rate, r is the reward, y is the discount factor, s' is the next state, a' is the next action, s is the current state, a is the current action.

5.1.1 Training reward

For the reward r, we can not just use the change of balance of the seller as the reward, or the agent will only aim to get a high balance at last, not how to defeat others. So we use the difference of balance between the player seller and the other seller as the reward:

$$D = \sum_{i=1}^{n} (balance_{player} - balance_i)\, i \in \{sellers \setminus player\}$$

$$r = D_{next} - D_{current}$$

So that the agent will try to defeat others.

5.1.2 Training discount

In our problem, the balance sellers get does not have a discount i.e. the effect that one seller earn 1 dollar today is the same as he earns 1 dollar tomorrow, which is near to reality. So the discount factor $y = 1$.
However, this lead to a tricky problem that during the training, the weights will not converge. They will go to infinity or negative infinity as the training episodes increase. This can be explained from the Q-value iteration formula:

$$Q(s, a) = r + y * \max_{a'} Q(s', a')$$

When $|y| < 1$, this recursive formula is guaranteed to converge to some finite values. However, when $y = 1$, the Q-value will be the sum of all the rewards of all episodes. This will lead to the weights go to infinity.
So to make this method running, we have to let $y < 1$, about 0.9. But this will decrease its performance. So we turn to try Monte Carlo Method

5.2 Monte Carlo Q-learning

We want to find a method that can conduct Q-learning without weights diverging to infinity. We learnt that On-Policy First-Visit MC Control, which takes each whole business war as an episode, and update Q value at the end of it. This avoid the problem that weights may accumulate to infinity as $y = 1$.

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
  $\pi \leftarrow$ an arbitrary ε-soft policy
  $Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
  $Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$
      $A^* \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken arbitrarily)
      For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

However, we should not just take the average to be Q-value since the ones long time ago may not be correct. So we apply Exponential moving average. Notice that this method has the disadvantage that information about state connections is wasted, and only when one war end, the weights will be updated, so the weight converge speed may be slow. To speedup the converge speed, we use Upper confidence bound as exploration function. Thus, we propose the algorithm as 2 and 0.

5.2.1 Approximate Monte Carlo Q-learning

Since the state space of our problem is very large, we have to apply approximate method so that the training time is acceptable. We use the same feature extractor as before, while the time for (state, action) pair is still kept to apply UCB exploration function. Therefore, the algorithm become as 4 and 0:

Algorithm 2 Episode update for Monte Carlo Q-learning

Require: Get an episode: $S_0, A_0, D_0, ..., S_{T-1}, A_{T-1}, D_{T-1}, S_T, D_T$
finalScore=D_T
for each S_t, A_t, D_t in episode **do**
$Q' =$ finalScore $- D_t$
value,time=QTable[(S_t, A_t)]
time+=1
QTable[(S_t, A_t)]=[value*(1-α)+$Q' * \alpha$,time]
end for
Where α is the learning rate and time is about how many times the certain state was explored.

Algorithm 3 Make choice using UCB for Monte Carlo Q-learning

Require: QTable, state S
for each valid pair S, A' **do**
value,time=QTable[(S, A')]
$Q_{tmp}[A']$=value + $\frac{\theta}{time+1}$
end for
return argmax(Q_{tmp})
Where θ is a hyper parameter controlling the extent of exploration

Algorithm 4 Episode update for Approximate Monte Carlo Q-learning

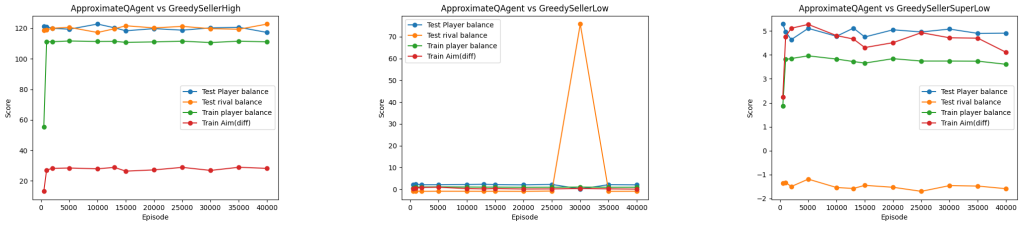
Require: Get an episode: $S_0, A_0, D_0, ..., S_{T-1}, A_{T-1}, D_{T-1}, S_T, D_T$
finalScore=D_T
for each S_t, A_t, D_t in episode **do**
$Q' = finalScore$ - D_t
time=timeTable[(S_t, A_t)]+=1
features=featureExtractor(S_t, A_t)
value=dot(weights,features)
difference=Q'-value
for f in features **do**
weights[f]+= α*difference * f.value
end for
end for

Algorithm 5 Make choice using UCB for Approximate Monte Carlo Q-learning

Require: QTable, state S
for each valid pair S, A' **do**
time=timeTable[(S, A')]
value= dot(weights,featureExtractor(S, A'))
$Q_{tmp}[A']$=value + $\frac{\theta}{time+1}$
end for
return argmax(Q_{tmp})

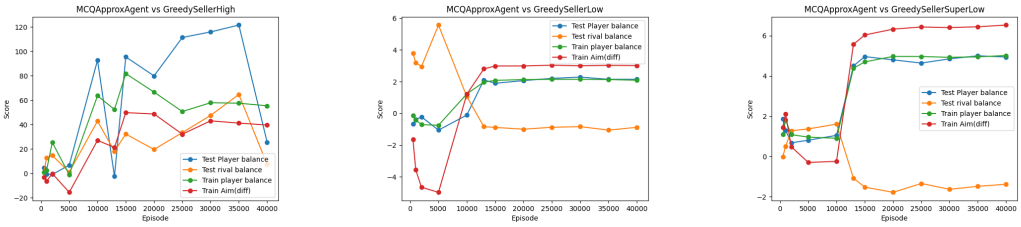
5.3 Experiment

For experiment, we compare Approximate Q-learning and Monte Carlo Approximate one. For Approximate Q-learning, we use epsilon-greedy exploration function and $\epsilon = 0.1, \gamma = 0.91, \alpha = 0.2$. For Monte Carlo Approximate Q-learning, we use UCB exploration function and $\theta = 10, \gamma = 1, \alpha = 0.2$. Our experiments for this part are done in the situation of: initial balance=20, daily cost=1, medium price=10, low price=8, super low price=6, high price=12, consumer number=2.



(a) Win Rate:0.54 (b) Win Rate:0.93 (c) Win Rate:1.00

Figure 10: Approximate Q-learning



(a) Win Rate:0.52 (b) Win Rate:0.93 (c) Win Rate:1.00

Figure 11: Monte Carlo Approximate Q-learning

While training, we insert test phases among episodes. Every test phases contain 100 rounds and take the average to get the data points. After training, another test phase containing 2000 rounds is taken to compute the final wining rate.

Recall that our Training aim is the difference between the player seller and his rivals. So we observe this line to see if the training converges.

From the experiment figures, We first can conclude that reinforcement learning can easily defeat low and super-low greedy agent, while hard to defeat greedy-high agent. Monte Carlo Approximate-Q agent just can not converge, while Approximate Q-agent can converge but the win rate is still only 0.54. Intuitively explaining, compare to other two rivals, who tends to often give out "low price", but this will lead his balance reduce quickly, which is like suicide, so it is easier to find out a strategy to win. Although the greedy-high rival is always giving out the highest price and make no preferred consumers, he is hard to die since the only cost is the daily cost, which make the player agent hard to defeat him since every low-price decision make lead to higher risk. In addition, from the view of method, We think there are two main reasons:

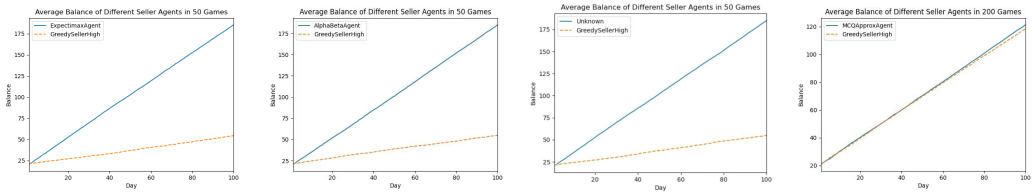
- The exploration in training makes the setting between training and testing different, then the model can not perform as well as training during testing. From the figure we can see that although the line of "Train Aim" goes relatively high, which show that the agent is already able to win in training, but this is not reflected in testing. Further, we study the operation records that the agent learnt, which shows that the agent only learns to always choose to give out "high" price. And in training, it just use the random property of exploration function to give out relatively low price so it can utilize this to win during training stage.
- We have tried to let the exploration rate gradually decrease as the episode goes, but this does no help for the agent to learn about the optimal strategy. We found that, for the two approximate agents, we use linear weights to approximate Q-values. However, this will perform worse when the Q-value function is actually non-linear. This can be seen in the figure(a) of two Approximate Q-learning. Even though the training episode goes to 40000,

the outcome of Monte Carlo one still vibrate. While for the other one, although it converges, it only converge to a local maxima, because the training aim is far below than Monte Carlo one. So we suppose using multiple layer of weights and introduce non-linear projections will be of help, which relates to the knowledge of deep-learning.

What's more, we found that the problem of gamma does not influence much of Approximate Q-agent. We suppose this is because the rivals are relatively "simple", which means that it is not necessary for the player agent to think long-term. So even though we set $\gamma = 0.91$ for Approximate Q-agent, its performance is not influenced much.

6 Comparison of Methods

Our results show that Expectimax Agent is more robust and flexible, but also computationally expensive. AlphaBeta Agent is more efficient and aggressive, but is less robust. Neural-network Expectimax Agent is an online algorithm with a nice performance in score, which is also more adaptive and intelligent. As for reinforcement learning agents, the results also show that it fails to defeat greedy-high rival agent due to exploration and linear weights issues. Overall, Neural-network Expectimax Agent is generally the most suitable solution to explore an acceptable policy. Figure 12 shows more details of different agents' behaviors in a particular game.



(a) **Expectimax** (b) **AlphaBeta** (c) **NN-Expectimax** (d) **MCQ Approx**

Figure 12: **Comparison of different agents' beheaviors**

7 Conclusions

We investigate the performance of several classical AI methods under the setting of "Business War". It simulates a market scenario where multiple monopoly sellers compete for consumers by setting prices for their products. The consumers have heterogeneous preferences. The sellers have different costs and profits, and they will attempt to survive and maximize their balance by the end of the game. Overall, we suggest the family of Expectimax algorithms is the best solution among all.

We also propose possible improvements for future work, such as using a more sophisticated function approximator or a different exploration strategy. Also, online algorithms are more feasible in real practice and should be paid more attention to.

8 Appendix

8.1 Open Source Policy

We have published our source code under the open-source license of MIT. The repo is at `https://github.com/rhfeiyang/CS181-AI-Project` .

8.2 Fun Facts

We intuitively found that when two rivals tend to give out low prices and take defeating others as their main aim, they will earn relatively less money in the end. While if both rivals tend to cooperate and give out relatively high prices, they will hold the market and both will make more money. In turn, when companies are trying to compete on price and win over consumers, consumers will gain the most. Since consumers far outnumber businesses, if one government wants to bring happiness to its people, it should sanction monopolies, preventing them from holding the market and making very high prices.

9 Reference

[1] G. Lugosi and S. Mendelson, Mean estimation and regression under heavy-tailed distributions: A survey, Foundations of Computational Mathematics, vol. 19, no. 5, pp. 11451190, 2019