Bandit Project

- Name: 朱宇轩
- Student ID：2020531016
- Date：December 21, 2021

Abstract

In probability theory, the multi-armed bandit problem (sometimes called the K- or N-armed bandit problem) is a problem in which a fixed limited set of resources must be allocated between competing (alternative) choices in a way that maximizes their expected gain, when each choice's properties are only partially known at the time of allocation, and may become better understood as time passes or by allocating resources to the choice. This is a classic reinforcement learning problem that exemplifies the exploration–exploitation tradeoff dilemma. The name comes from imagining a gambler at a row of slot machines (sometimes known as "one-armed bandits"), who has to decide which machines to play, how many times to play each machine and in which order to play them, and whether to continue with the current machine or try a different machine. The multi-armed bandit problem also falls into the broad category of stochastic scheduling.[3]

In the problem, each machine provides a random reward from a probability distribution specific to that machine. The objective of the gambler is to maximize the sum of rewards earned through a sequence of lever pulls.The crucial tradeoff the gambler faces at each trial is between "exploitation" of the machine that has the highest expected payoff and "exploration" to get more information about the expected payoffs of the other machines.[3]

Contents

Settings

imports

In [55]:

```
import random
import math
import numpy as np
from scipy.stats import beta
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import cm
from matplotlib.ticker import LinearLocator

plt.style.use('ggplot')
sns.set_style("white")


font1 = {'family': 'serif',
        'color':  'darkred',
        'weight': 'normal',
        'size': 12,
        }

font2 = {'family': 'serif',
        'color':  'dodgerblue',
        'weight': 'normal',
        'size': 12,
        }

colors1 = '#DC143C'
colors2 = '#00CED1'
```

Global background

In [74]:

```
def arms(index, p1 = 0.8, p2 = 0.6, p3 = 0.5):
    '''
    Return the result of the indexth arm being pulled down
```

```
    '''
    if index == 1:
        if random.random() <= p1:
            return 1
        else:
            return 0
    elif index ==2:
        if random.random() <= p2:
            return 1
        else:
            return 0
    elif index == 3:
        if random.random() <= p3:
            return 1
        else:
            return 0
    else:
        raise IndexError

truth = [0.8, 0.6, 0.5]
```

Useful tools

In [57]:

```
def arg_max(list):
    '''
    Returns the index of the maximum value of the list.
    When there are multiple maximum values in the list, equal probability returns one of them.
    '''
    index = 0
    max_value = 0
    max_index = []
    for i in list:
        if i > max_value:
            max_value = i
            max_index = [index]
        elif i == max_value:
            max_index.append(index)
        index += 1
    if len(max_index) == 1:
        return max_index[0]
    else:
        return random.choice(max_index)

def distance_between_truth(theta):
    return (theta[0]-truth[0])**2+(theta[1]-truth[1])**2+(theta[2]-truth[2])**2
```

Main Codes

In [58]:

```
class Result():
    def __init__(self, method, parameter, repeat, output = 'false', N = 6000):
        self.output_flag = output
        self.repeat = repeat #repeat times
        self.count = 0 #experiment times
        self.countarms = [0,0,0]
        self.N = N
        self.theta_total = [0,0,0]
        self.theta_mean = [0,0,0]
        self.gain_total = 0

        # optimal_choices_ratio
        self.optimal_choices_ratio_total = []
        self.ratio_of_optimal_choices_min = [0 for x in range(self.N)]
        self.ratio_of_optimal_choices_max = [0 for x in range(self.N)]
```

```
        # optimal_gain_ratio
        self.optimal_gain_ratio_total = []
        self.ratio_of_optimal_gain_min = [0 for x in range(self.N)]
        self.ratio_of_optimal_gain_max = [0 for x in range(self.N)]

        # distance
        self.distance_total = []
        self.distance_min = [0 for x in range(self.N)]
        self.distance_max = [0 for x in range(self.N)]

        if method in ['greedy', 'UCB', 'TS']:
            self.run(method, parameter)
        else:
            raise IndexError

    def run(self, method, parameter):
        if method == 'greedy':
            self.target = 'epsilon = {}'.format(parameter)
            for index in tqdm(range(self.repeat)):
                optimal_choices_ratio, optimal_gain_ratio, distance = self.greedy(parameter)
                self.optimal_choices_ratio_total.append(optimal_choices_ratio)
                self.optimal_gain_ratio_total.append(optimal_gain_ratio)
                self.distance_total.append(distance)
        elif method == 'UCB':
            self.target = 'c = {}'.format(parameter)
            for index in tqdm(range(self.repeat)):
                optimal_choices_ratio, optimal_gain_ratio, distance = self.UCB(parameter)
                self.optimal_choices_ratio_total.append(optimal_choices_ratio)
                self.optimal_gain_ratio_total.append(optimal_gain_ratio)
                self.distance_total.append(distance)
        elif method == 'TS':
            if parameter == 1:
                self.target = '{(a1,b1)=(1,1),(a2,b2)=(1,1),(a3,b3)=(1,1)}'
            else:
                self.target = '{(a1,b1)=(601,401),(a2,b2)=(401,601),(a3,b3)=(2,3)}'
            for index in tqdm(range(self.repeat)):
                optimal_choices_ratio, optimal_gain_ratio, distance = self.TS(parameter)
                self.optimal_choices_ratio_total.append(optimal_choices_ratio)
                self.optimal_gain_ratio_total.append(optimal_gain_ratio)
                self.distance_total.append(distance)

        self.calculate()
        self.output(self.target)

    def calculate(self):
        for i in range(self.N):
            # optimal_choices_ratio
            self.ratio_of_optimal_choices_min[i] = min([x[i] for x in self.optimal_choices_ratio
            self.ratio_of_optimal_choices_max[i] = max([x[i] for x in self.optimal_choices_ratio
            # optimal_gain_ratio
            self.ratio_of_optimal_gain_min[i] = min([x[i] for x in self.optimal_gain_ratio_total
            self.ratio_of_optimal_gain_max[i] = max([x[i] for x in self.optimal_gain_ratio_total
            # distance
            self.distance_min[i] = min([x[i] for x in self.distance_total])
            self.distance_max[i] = max([x[i] for x in self.distance_total])



    def greedy(self, epsilon):
        optimal_choices_ratio = [] #In the previous index experiments, the ratio of No. 1 arm was
        optimal_gain_ratio = [] #Ratio of income to ideal value after index experiments
        distance = [] #Mean square error between predicted and actual values after index experime
        theta_mean = [0,0,0] # estimation of theta_j
        count = [0,0,0] # count(j)
        gain = 0 # total reward
        result = 0 # the result of the arm of this time
        for t in range(self.N):
            if random.random() <= epsilon:
                I = random.randint(1,3)
```

```
            else:
                I = arg_max(theta_mean) + 1
            result = arms(I)
            self.countarms[I-1] += 1
            gain += result
            count[I-1] += 1
            theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
            # optimal_choices_ratio
            optimal_choices_ratio.append(count[0] / (t+1))
            # optimal_gain_ratio
            optimal_gain_ratio.append(gain / ((t+1)*0.8))
            # distance
            distance.append(distance_between_truth(theta_mean))

        for i in range(3):
            self.theta_total[i] += theta_mean[i]
        self.gain_total += gain

        return optimal_choices_ratio, optimal_gain_ratio, distance

    def UCB(self, c):
        optimal_choices_ratio = [] #In the previous index experiments, the ratio of No. 1 arm wa
        optimal_gain_ratio = [] #Ratio of income to ideal value after index experiments
        distance = [] #Mean square error between predicted and actual values after index experim
        theta_mean = [0,0,0] # estimation of theta_j
        count = [0,0,0] # count(j)
        gain = 0 # total reward
        result = 0 # the result of the arm of this time
        for t in range(3):
            count[t] += 1
            result = arms(t+1)
            gain += result
            theta_mean[t] = result
            # optimal_choices_ratio
            optimal_choices_ratio.append(count[0] / (t+1))
            # optimal_gain_ratio
            optimal_gain_ratio.append(gain / ((t+1)*0.8))
            # distance
            distance.append(distance_between_truth(theta_mean))
        for t in range(3, self.N):
            temp = [0,0,0]
            for j in range(3):
                temp[j] = theta_mean[j] + c * (2*math.log10(t)/count[j])**0.5
            I = arg_max(temp) + 1
            result = arms(I)
            self.countarms[I-1] += 1
            count[I-1] += 1
            gain += result
            theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
            # optimal_choices_ratio
            optimal_choices_ratio.append(count[0] / (t+1))
            # optimal_gain_ratio
            optimal_gain_ratio.append(gain / ((t+1)*0.8))
            # distance
            distance.append(distance_between_truth(theta_mean))

        for i in range(3):
            self.theta_total[i] += theta_mean[i]
        self.gain_total += gain

        return optimal_choices_ratio, optimal_gain_ratio, distance

    def TS(self, index):
        if index == 1:
            parameter = [[1,1],[1,1],[1,1]]
        elif index == 2:
            parameter = [[601,401],[401,601],[2,3]]
        else:
            raise IndexError
```

```
        optimal_choices_ratio = [] #In the previous index experiments, the ratio of No. 1 arm was
        optimal_gain_ratio = [] #Ratio of income to ideal value after index experiments
        distance = [] #Mean square error between predicted and actual values after index experime
        theta_mean = [0,0,0] # estimation of theta_j
        count = [0,0,0] # count(j)
        gain = 0 # total reward
        result = 0 # the result of the arm of this time
        for t in range(self.N):
            sample_theta = [0,0,0]
            for i in range(3):
                sample_theta[i] = beta.rvs(parameter[i][0],parameter[i][1])
            I = arg_max(sample_theta) + 1
            result = arms(I)
            self.countarms[I-1] += 1
            count[I-1] += 1
            parameter[I-1][0] += result
            parameter[I-1][1] += 1-result
            gain += result
            # optimal_choices_ratio
            optimal_choices_ratio.append(count[0] / (t+1))
            # optimal_gain_ratio
            optimal_gain_ratio.append(gain / ((t+1)*0.8))
            # distance
            for i in range(3):
                theta_mean[i] = parameter[i][0] / (parameter[i][0]+parameter[i][1])
            distance.append(distance_between_truth(theta_mean))

        for i in range(3):
            theta_mean[i] = parameter[i][0] / (parameter[i][0]+parameter[i][1])
        for i in range(3):
            self.theta_total[i] += theta_mean[i]
        self.gain_total += gain

        return optimal_choices_ratio, optimal_gain_ratio, distance

    def output(self, parameter):
        self.gain_mean = self.gain_total / self.repeat
        theta_mean = [0,0,0]
        for i in range(3):
            theta_mean[i] = self.theta_total[i] / self.repeat
        self.distance_mean = sum([x[-1] for x in self.distance_total])/self.repeat
        if self.output_flag == 'true':
            print("the average gain of {} is {}".format(parameter, self.gain_mean))
            print("the estimated probability is {}".format(theta_mean))
            print("the times of each arm being pulled is {}".format(self.countarms))
            print("the mean square error between the truth and the estimation is {}".format(self.
```

Result

greedy

Test Code

In [59]:

```
G1 = Result('greedy',0.2,200, output = 'true')
G2 = Result('greedy',0.4,200, output = 'true')
G3 = Result('greedy',0.6,200, output = 'true')
G4 = Result('greedy',0.8,200, output = 'true')
```

```
the average gain of epsilon = 0.2 is 4596.475
the estimated probability is [0.8006939011627735, 0.5980637755137636, 0.5013710023742433]
the times of each arm being pulled is [1034084, 83952, 81964]
the mean square error between the truth and the estimation is 0.0012117476357039311
```

```
the average gain of epsilon = 0.4 is 4390.44
the estimated probability is [0.79941312369369, 0.6000279313722066, 0.4963371669077414]
the times of each arm being pulled is [876568, 161974, 161458]
the mean square error between the truth and the estimation is 0.0006413047242263518


the average gain of epsilon = 0.6 is 4197.045
the estimated probability is [0.8001306392006053, 0.6002174596456423, 0.49864235417551717]
the times of each arm being pulled is [718445, 240634, 240921]
the mean square error between the truth and the estimation is 0.00041202817652681703


the average gain of epsilon = 0.8 is 3997.535
the estimated probability is [0.800008310142434, 0.5983546865808536, 0.5005342308780905]
the times of each arm being pulled is [559346, 320602, 320052]
the mean square error between the truth and the estimation is 0.0003464143888422032
```

Trying more epsilon to find the best one

In [60]:

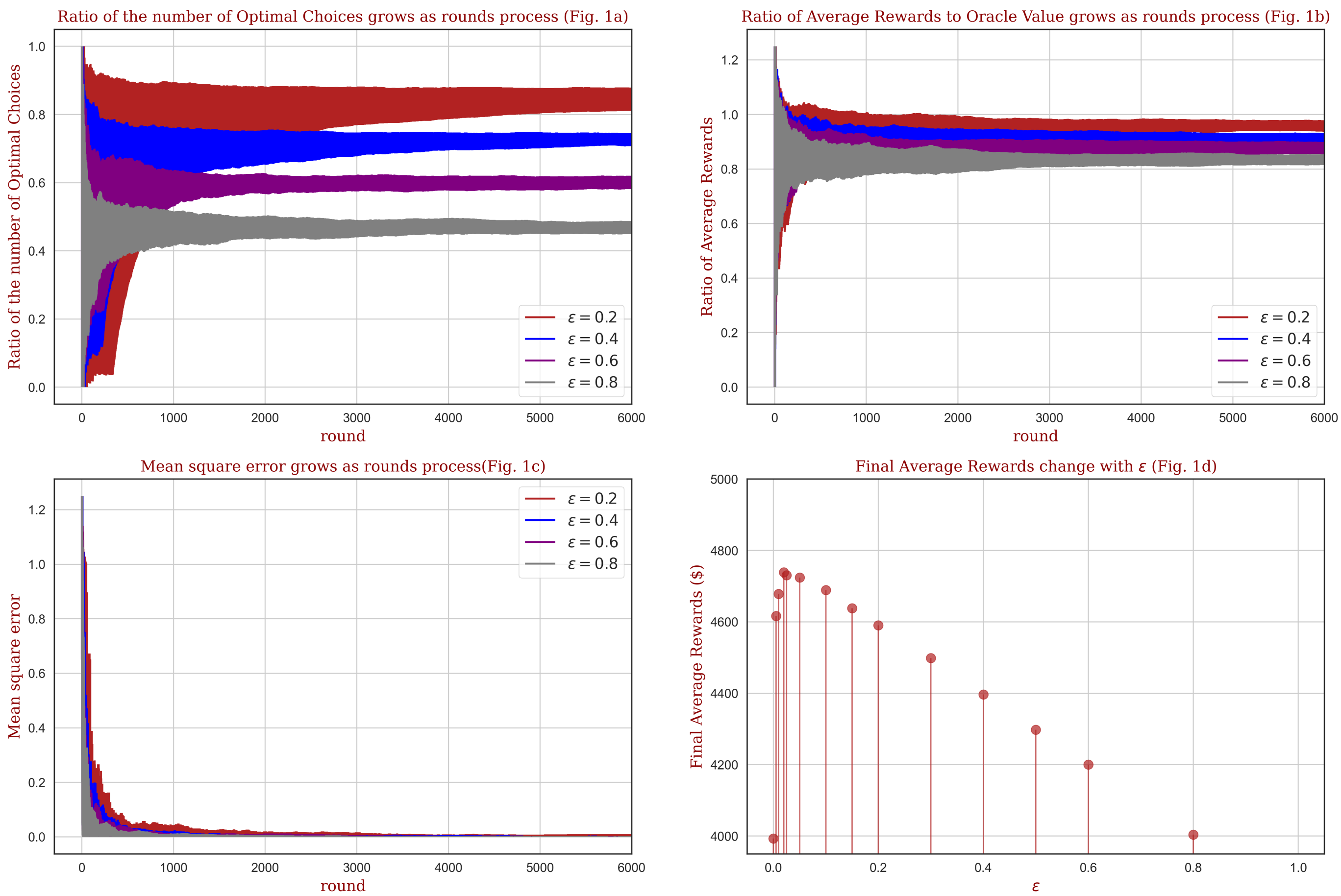
```
greedy_gain = []
epsilon = [0,0.005,0.01,0.02,0.025,0.05,0.1,0.15,0.2,0.3,0.4,0.5,0.6,0.8,1.0]
for i in tqdm(epsilon):
    G = Result('greedy', i, 200)
    greedy_gain.append(G.gain_mean)
```

Code to visualize the data

In [61]:

```
fig1, ax1 = plt.subplots(2, 2, figsize=(18,12), dpi = 400)

ax1_1 = ax1[0][0]
ax1_2 = ax1[0][1]
ax1_3 = ax1[1][0]
ax1_4 = ax1[1][1]

x_rounds1 = np.arange(0,6000,1)

ax1_1.vlines(x=x_rounds1, ymin = G1.ratio_of_optimal_choices_min, ymax=G1.ratio_of_optimal_choic
ax1_1.vlines(x=x_rounds1, ymin = G2.ratio_of_optimal_choices_min, ymax=G2.ratio_of_optimal_choic
ax1_1.vlines(x=x_rounds1, ymin = G3.ratio_of_optimal_choices_min, ymax=G3.ratio_of_optimal_choic
ax1_1.vlines(x=x_rounds1, ymin = G4.ratio_of_optimal_choices_min, ymax=G4.ratio_of_optimal_choic
ax1_1.axis(xmax = 6000)
ax1_1.set_title('Ratio of the number of Optimal Choices grows as rounds process (Fig. 1a)', fontc
ax1_1.set_xlabel('round', fontdict=font1)
ax1_1.set_ylabel('Ratio of the number of Optimal Choices', fontdict=font1)

ax1_2.vlines(x=x_rounds1, ymin =G1.ratio_of_optimal_gain_min, ymax=G1.ratio_of_optimal_gain_max,
ax1_2.vlines(x=x_rounds1, ymin =G2.ratio_of_optimal_gain_min, ymax=G2.ratio_of_optimal_gain_max,
ax1_2.vlines(x=x_rounds1, ymin =G3.ratio_of_optimal_gain_min, ymax=G3.ratio_of_optimal_gain_max,
ax1_2.vlines(x=x_rounds1, ymin =G4.ratio_of_optimal_gain_min, ymax=G4.ratio_of_optimal_gain_max,
ax1_2.axis(xmax = 6000)
ax1_2.set_title('Ratio of Average Rewards to Oracle Value grows as rounds process (Fig. 1b)', fon
ax1_2.set_xlabel('round', fontdict=font1)
ax1_2.set_ylabel('Ratio of Average Rewards', fontdict=font1)

ax1_3.vlines(x=x_rounds1, ymin =G1.distance_min, ymax=G1.distance_max, color='firebrick')
ax1_3.vlines(x=x_rounds1, ymin =G2.distance_min, ymax=G2.distance_max, color='blue')
ax1_3.vlines(x=x_rounds1, ymin =G3.distance_min, ymax=G3.distance_max, color='purple')
ax1_3.vlines(x=x_rounds1, ymin =G4.distance_min, ymax=G4.distance_max, color='gray')
ax1_3.axis(xmax = 6000)
ax1_3.set_title('Mean square error grows as rounds process(Fig. 1c)', fontdict=font1)
```

```
ax1_3.set_xlabel('round', fontdict=font1)
ax1_3.set_ylabel('Mean square error', fontdict=font1)

ax1_4.scatter(epsilon, greedy_gain, s=55, color='firebrick', alpha=0.7)
ax1_4.axis(ymin = 3950, ymax = 5000)
ax1_4.vlines(x=epsilon, ymin=3950, ymax=greedy_gain, color='firebrick', alpha=0.7, linewidth=1)
ax1_4.set_title(r'Final Average Rewards change with $\epsilon$ (Fig. 1d)', fontdict=font1)
ax1_4.set_xlabel(r'$\epsilon$', fontdict=font1)
ax1_4.set_ylabel('Final Average Rewards ($)', fontdict=font1)

ax1_1.legend((r'$\epsilon = 0.2$',r'$\epsilon = 0.4$',r'$\epsilon = 0.6$',r'$\epsilon = 0.8$'),
ax1_2.legend((r'$\epsilon = 0.2$',r'$\epsilon = 0.4$',r'$\epsilon = 0.6$',r'$\epsilon = 0.8$'),
ax1_3.legend((r'$\epsilon = 0.2$',r'$\epsilon = 0.4$',r'$\epsilon = 0.6$',r'$\epsilon = 0.8$'),
ax1_1.grid()
ax1_2.grid()
ax1_3.grid()
ax1_4.grid()

plt.show()
```



Result evaluation

- From the data results of the selected epsilon, it can be seen that the smaller the epsilon, the easier it is to select the best arm (the arm with the highest probability of obtaining 1), that is, it is easier to obtain high income.

- From the number of times each arm is pulled, the greedy algorithm will have a few trial times and a few times to obtain benefits. The proportion of this quantity depends on the epsilon. When the epsilon is smaller, the algorithm is more inclined to obtain reward. On the contrary, it will test more times and gain a more accurate estimated probability and a smaller gain. However, in the epsilon selected by the topic, it can be seen from Figure 3 that the real probability can always be accurately estimated.

- It can be seen from Figure 4 that for the arm probability given in the topic, the best epsilon should be around 0.025. At this time, the greedy algorithm can balance the number allocation of exploration and expansion, so as to determine the best arm within the reliable range with the minimum number of times, and use the remaining times for expansion, so as to obtain the maximum reward.

UCB

Test Code

In [62]:
```
U1 = Result('UCB', 2, 200, output='true')
U2 = Result('UCB', 6, 200, output='true')
U3 = Result('UCB', 9, 200, output='true')
```

```
the average gain of c = 2 is 4658.25
the estimated probability is [0.8003895342207455, 0.5932559451177447, 0.49543157024383616]
the times of each arm being pulled is [1078099, 78303, 42998]
the mean square error between the truth and the estimation is 0.001879080472771918


the average gain of c = 6 is 4289.265
the estimated probability is [0.7993568195204707, 0.6007755695162659, 0.49977794934503933]
the times of each arm being pulled is [774567, 253773, 171060]
the mean square error between the truth and the estimation is 0.0004976827868275872


the average gain of c = 9 is 4145.4
the estimated probability is [0.7994425134272266, 0.599623737507218, 0.5007682181940206]
the times of each arm being pulled is [660605, 308346, 230449]
the mean square error between the truth and the estimation is 0.00043000889208959393
```

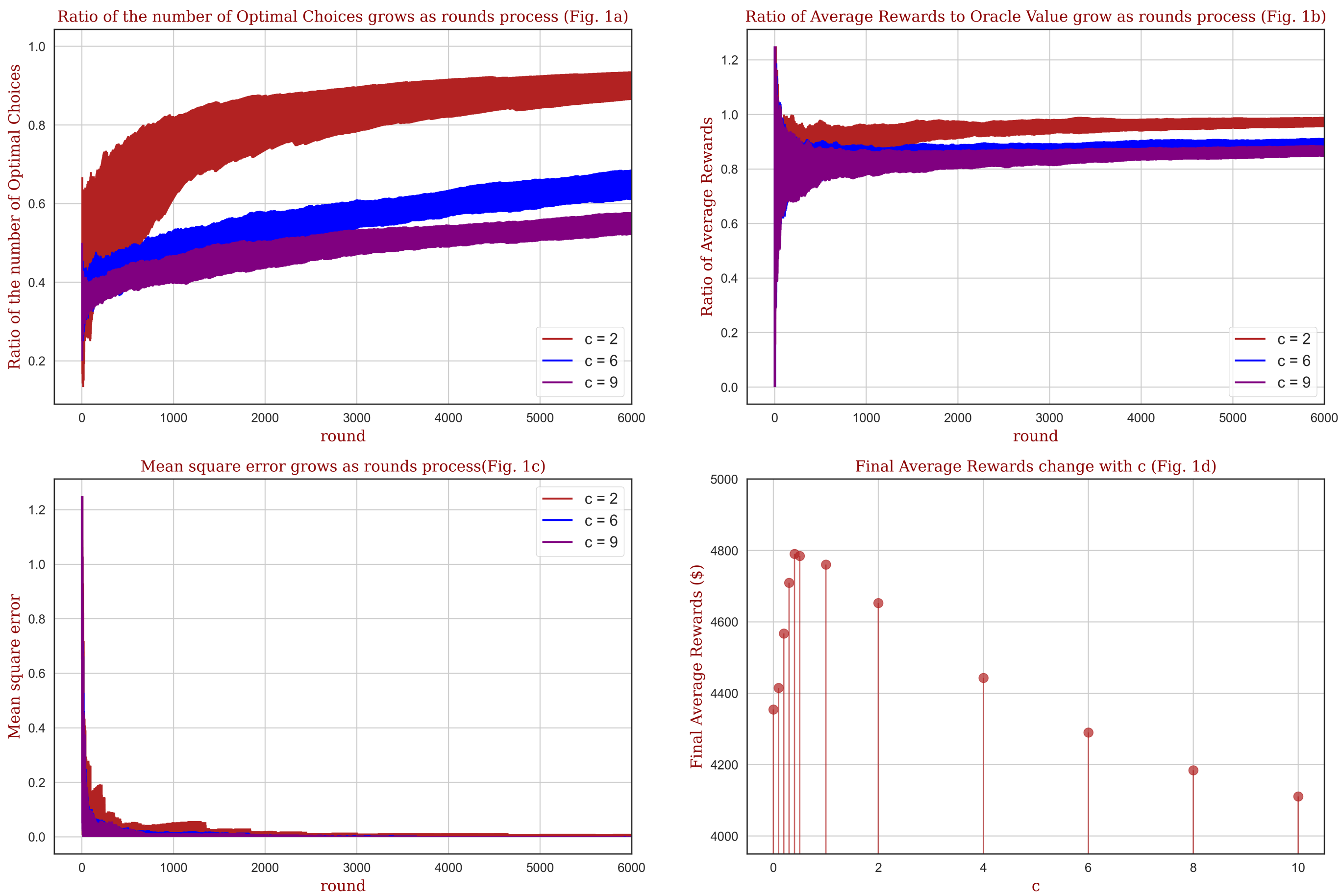
Trying more c to find the best one

In [63]:
```
UCB_gain = []
c = [0,0.1,0.2,0.3,0.4,0.5,1,2,4,6,8,10]
for i in tqdm(c):
    U = Result('UCB', i, 100)
    UCB_gain.append(U.gain_mean)
```

Code to visualize the data

In [64]:
```
fig2, ax2 = plt.subplots(2, 2, figsize=(18,12), dpi = 400)

ax2_1 = ax2[0][0]
ax2_2 = ax2[0][1]
ax2_3 = ax2[1][0]
ax2_4 = ax2[1][1]

x_rounds1 = np.arange(0,6000,1)

ax2_1.vlines(x=x_rounds1, ymin = U1.ratio_of_optimal_choices_min, ymax=U1.ratio_of_optimal_choic
ax2_1.vlines(x=x_rounds1, ymin = U2.ratio_of_optimal_choices_min, ymax=U2.ratio_of_optimal_choic
ax2_1.vlines(x=x_rounds1, ymin = U3.ratio_of_optimal_choices_min, ymax=U3.ratio_of_optimal_choic
ax2_1.axis(xmax = 6000)
ax2_1.set_title('Ratio of the number of Optimal Choices grows as rounds process (Fig. 1a)', fontd
ax2_1.set_xlabel('round', fontdict=font1)
ax2_1.set_ylabel('Ratio of the number of Optimal Choices', fontdict=font1)

ax2_2.vlines(x=x_rounds1, ymin =U1.ratio_of_optimal_gain_min, ymax=U1.ratio_of_optimal_gain_max,
ax2_2.vlines(x=x_rounds1, ymin =U2.ratio_of_optimal_gain_min, ymax=U2.ratio_of_optimal_gain_max,
ax2_2.vlines(x=x_rounds1, ymin =U3.ratio_of_optimal_gain_min, ymax=U3.ratio_of_optimal_gain_max,
ax2_2.axis(xmax = 6000)
```

```
ax2_2.set_title('Ratio of Average Rewards to Oracle Value grow as rounds process (Fig. 1b)', fon
ax2_2.set_xlabel('round', fontdict=font1)
ax2_2.set_ylabel('Ratio of Average Rewards', fontdict=font1)

ax2_3.vlines(x=x_rounds1, ymin =U1.distance_min, ymax=U1.distance_max, color='firebrick')
ax2_3.vlines(x=x_rounds1, ymin =U2.distance_min, ymax=U2.distance_max, color='blue')
ax2_3.vlines(x=x_rounds1, ymin =U3.distance_min, ymax=U3.distance_max, color='purple')
ax2_3.axis(xmax = 6000)
ax2_3.set_title('Mean square error grows as rounds process(Fig. 1c)', fontdict=font1)
ax2_3.set_xlabel('round', fontdict=font1)
ax2_3.set_ylabel('Mean square error', fontdict=font1)

ax2_4.scatter(c, UCB_gain, s=55, color='firebrick', alpha=0.7)
ax2_4.axis(ymin = 3950, ymax = 5000)
ax2_4.vlines(x=c, ymin=3950, ymax=UCB_gain, color='firebrick', alpha=0.7, linewidth=1)
ax2_4.set_title(r'Final Average Rewards change with c (Fig. 1d)', fontdict=font1)
ax2_4.set_xlabel(r'c', fontdict=font1)
ax2_4.set_ylabel('Final Average Rewards ($)', fontdict=font1)

ax2_1.legend(('c = 2','c = 6','c = 9'), loc='lower right', fontsize = 'large')
ax2_2.legend(('c = 2','c = 6','c = 9'), loc='lower right', fontsize = 'large')
ax2_3.legend(('c = 2','c = 6','c = 9'), loc='upper right', fontsize = 'large')
ax2_1.grid()
ax2_2.grid()
ax2_3.grid()
ax2_4.grid()

plt.show()
```



Result evaluation

- From the data results of the selected c, it can be seen that the smaller the c is, the easier it is to select the best arm (the arm with the highest probability of obtaining 1), that is, it is easier to obtain high income.
- From the number of times each arm is pulled, the UCB algorithm will have a few trial times and a few times to obtain benefits. The proportion of this quantity depends on c. When c is smaller, the algorithm

is more inclined to obtain reward. On the contrary, it will test more times and gain a more accurate estimated probability and a smaller gain. However, with c selected by the topic, it can be seen from Figure 3 that the real probability can always be accurately estimated.

- It can be seen from Figure 4 that for the arm probability given in the topic, the best c should be around 0.4. At this time, the UCB algorithm can balance the number allocation of exploration and expansion, so as to determine the best arm within the reliable range with the minimum number of times, and use the remaining times for expansion, so as to obtain the maximum reward.

TS

Test Code

In [65]:

```
T1 = Result('TS',1,200,output='true')
T2 = Result('TS',2,200,output='true')
```

```
the average gain of {(a1,b1)=(1,1),(a2,b2)=(1,1),(a3,b3)=(1,1)} is 4784.15
the estimated probability is [0.7998868798136948, 0.5492306611849983, 0.4635101480110133]
the times of each arm being pulled is [1186150, 9093, 4757]
the mean square error between the truth and the estimation is 0.025845522775866554
```

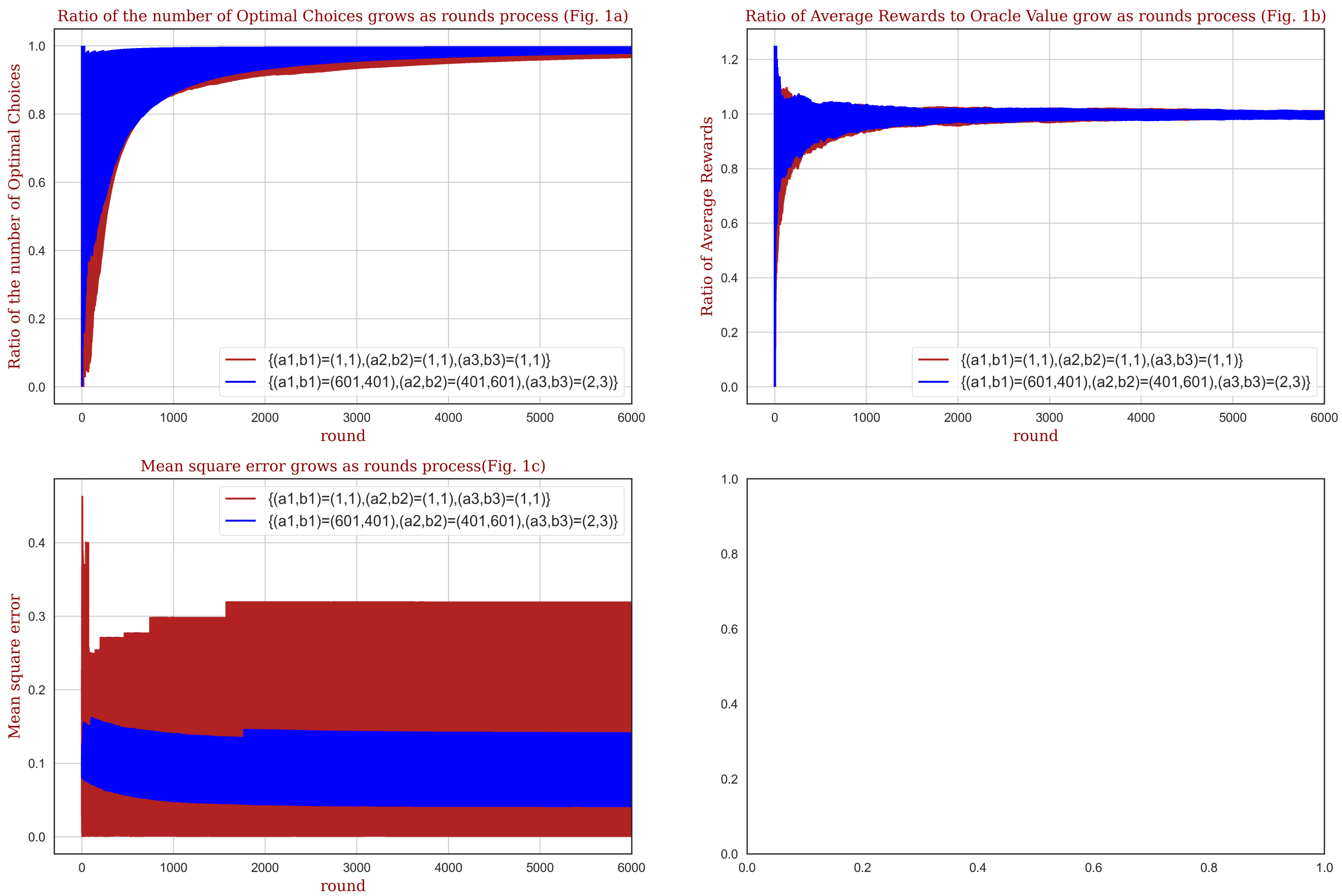
```
the average gain of {(a1,b1)=(601,401),(a2,b2)=(401,601),(a3,b3)=(2,3)} is 4788.345
the estimated probability is [0.7711773642848471, 0.4001996007984022, 0.4435354673374803]
the times of each arm being pulled is [1192316, 0, 7684]
the mean square error between the truth and the estimation is 0.05221613954382462
```

Code to visualize the data

In [66]:

```
fig3, ax3 = plt.subplots(2, 2, figsize=(18,12), dpi = 400)

ax3_1 = ax3[0][0]
ax3_2 = ax3[0][1]
ax3_3 = ax3[1][0]

x_rounds1 = np.arange(0,6000,1)

ax3_1.vlines(x=x_rounds1, ymin = T1.ratio_of_optimal_choices_min, ymax=T1.ratio_of_optimal_choic
ax3_1.vlines(x=x_rounds1, ymin = T2.ratio_of_optimal_choices_min, ymax=T2.ratio_of_optimal_choic
ax3_1.axis(xmax = 6000)
ax3_1.set_title('Ratio of the number of Optimal Choices grows as rounds process (Fig. 1a)', font
ax3_1.set_xlabel('round', fontdict=font1)
ax3_1.set_ylabel('Ratio of the number of Optimal Choices', fontdict=font1)

ax3_2.vlines(x=x_rounds1, ymin =T1.ratio_of_optimal_gain_min, ymax=T1.ratio_of_optimal_gain_max,
ax3_2.vlines(x=x_rounds1, ymin =T2.ratio_of_optimal_gain_min, ymax=T2.ratio_of_optimal_gain_max,
ax3_2.axis(xmax = 6000)
ax3_2.set_title('Ratio of Average Rewards to Oracle Value grow as rounds process (Fig. 1b)', fon
ax3_2.set_xlabel('round', fontdict=font1)
ax3_2.set_ylabel('Ratio of Average Rewards', fontdict=font1)

ax3_3.vlines(x=x_rounds1, ymin =T1.distance_min, ymax=T1.distance_max, color='firebrick')
ax3_3.vlines(x=x_rounds1, ymin =T2.distance_min, ymax=T2.distance_max, color='blue')
ax3_3.axis(xmax = 6000)
ax3_3.set_title('Mean square error grows as rounds process(Fig. 1c)', fontdict=font1)
ax3_3.set_xlabel('round', fontdict=font1)
ax3_3.set_ylabel('Mean square error', fontdict=font1)

ax3_1.legend(('{(a1,b1)=(1,1),(a2,b2)=(1,1),(a3,b3)=(1,1)}','{(a1,b1)=(601,401),(a2,b2)=(401,601)
ax3_2.legend(('{(a1,b1)=(1,1),(a2,b2)=(1,1),(a3,b3)=(1,1)}','{(a1,b1)=(601,401),(a2,b2)=(401,601)
ax3_3.legend(('{(a1,b1)=(1,1),(a2,b2)=(1,1),(a3,b3)=(1,1)}','{(a1,b1)=(601,401),(a2,b2)=(401,601)
```

```
ax3_1.grid()
ax3_2.grid()
ax3_3.grid()

plt.show()
```



Result evaluation

- From Figure 1, we can know that the second set of parameters (blue) gives a higher a priori probability to the optimal arm, so that it can select the optimal arm faster and use more times for expansion.

- From Fig. 2, we can see that no matter what group of parameters are used, TS algorithm can approach the theoretical optimal return.

    - The reason for the success of the first set of parameters is that it does not set a strong priori probability, which can be approximated as no priori probability. Therefore, the results of each exploration can effectively contribute to the posteriori probability. Therefore, after limited and less exploration, it can quickly select the best arm. However, it is also for this reason that its experiment has a great impact on the posterior probability, that is, this group of parameters can not effectively estimate the probability value. As can be seen from Figure 3, its mean square error changes greatly with repeated experiments and cannot approach the theoretical value.

    - The second set of parameters is successful because its a priori probability ensures the selection of the best arm. The subsequent 6000 experiments can affect the posterior probability, but the strength is smaller than the first group of parameters. Therefore, this group of parameters will be used for expansion more times, so that it has a great error for the non optimal probability prediction value, which is the reason for its large overall mean square error.

- In general, TS algorithm tends to expand rather than explore. After it determines the best arm, it hardly explores other arms. Therefore, behind its absolute advantage of reward, its disadvantage is that if the wrong and high weighted a priori probability is given (i.e., the value of (a, b) is much greater than the number of experiments 6000), the result will be far worse than expected.

Comparison and Understanding

- Compared with the three algorithms, TS algorithm is easier to obtain higher income, greedy algorithm is more inclined to predict the probability values of the three arms, and UCB algorithm is in a compromise position. Therefore, choosing which algorithm needs to consider the actual needs of the task.
- As mentioned above, the work of different algorithms can be abstracted as a trade-off between exploration and expansion. From the output results of each experiment, we can also clearly see the number of times each arm is pulled.
  - The greedy algorithm uses the parameter epsilon to limit the proportion of exploration and expansion, which is a certain value. Even after confirming the best arm, some test times are still used to explore the other two arms. Therefore, the greedy algorithm can accurately predict the three probabilities, so there is a smaller mean square error. But its benefits are relatively small.
  - The trade-off process of UCB algorithm is affected by c and the number of experiments. Academically, the UCB algorithm uses housing boundaries and try to limit the solution to the optimal mathematically. It is dynamic since the adjustment item contains variables that change over time, and this is automatically self adjustment without humans although we still need to find the optimal c. In general, this dynamic adjustment is similar to the connotation of machine learning.
  - TS algorithm, using Beta distribution and Bayesian theory, can determine the best arm as quickly as possible, and stick to it to obtain the best reward. However, the disadvantage is that the misleading caused by the great a priori probability of an error cannot be offset within a limited number of tests, and the probability of the remaining two arms cannot be predicted, although this is sometimes not necessary.

Dependent Case

Case 1: dependent on the other two arms

Take a certain case as an example:

- P(A=1) = 0.8
- P(B=1|A=1) = 0.7
- P(B=1|A=0) = 0.2
- P(C=1|A=1) = 0.9
- P(C=1|A=0) = 0.1

In [67]:

```
def arms(index):
    '''
    Return the result of the indexth arm being pulled down with dependent case 1
    '''
    if random.random() <= 0.8:
        A = 1
        if random.random() <= 0.7:
            B = 1
        else:
            B = 0
        if random.random() <= 0.9:
            C = 1
        else:
            C = 0
    else:
        A = 0
        if random.random() <= 0.2:
            B = 1
        else:
            B = 0
```

```
        if random.random() <= 0.1:
            C = 1
        else:
            C = 0
    return [A,B,C][index-1]


truth = [0.8, 0.6, 0.74]
```

test for dependent case 1

In [68]:
```
G_case1_1 = Result('greedy',0.2,200,'true')
G_case1_2 = Result('greedy',0.4,200,'true')
G_case1_3 = Result('greedy',0.6,200,'true')
G_case1_4 = Result('greedy',0.8,200,'true')
```

```
the average gain of epsilon = 0.2 is 4688.275
the estimated probability is [0.8000195517544162, 0.6028266906848937, 0.7392983953482735]
the times of each arm being pulled is [1013019, 81753, 105228]
the mean square error between the truth and the estimation is 0.0009125533440285036


the average gain of epsilon = 0.4 is 4588.13
the estimated probability is [0.8002257187198657, 0.6008144405296185, 0.739819490278206]
the times of each arm being pulled is [864722, 161653, 173625]
the mean square error between the truth and the estimation is 0.0005518685018192101


the average gain of epsilon = 0.6 is 4478.855
the estimated probability is [0.798836070035971, 0.5994326815421721, 0.7394003852142171]
the times of each arm being pulled is [708947, 240516, 250537]
the mean square error between the truth and the estimation is 0.000406407374861149


the average gain of epsilon = 0.8 is 4380.23
the estimated probability is [0.7994777832145035, 0.6000183797204033, 0.7391521217345156]
the times of each arm being pulled is [555259, 319606, 325135]
the mean square error between the truth and the estimation is 0.00030585114637995216
```

In [69]:
```
U_case1_1 = Result('UCB', 2, 200, output='true')
U_case1_2 = Result('UCB', 6, 200, output='true')
U_case1_3 = Result('UCB', 9, 200, output='true')
```

```
the average gain of c = 2 is 4645.605
the estimated probability is [0.8009601659660076, 0.593288608887031, 0.7384167022719227]
the times of each arm being pulled is [842535, 72685, 284180]
the mean square error between the truth and the estimation is 0.0009615562326787674


the average gain of c = 6 is 4466.035
the estimated probability is [0.8001069151116619, 0.6002256952666838, 0.7391653922485769]
the times of each arm being pulled is [581565, 213126, 404709]
the mean square error between the truth and the estimation is 0.00035920737553000515


the average gain of c = 9 is 4413.18
the estimated probability is [0.7996732174768476, 0.5993001579900571, 0.7399236080539509]
the times of each arm being pulled is [523140, 261701, 414559]
the mean square error between the truth and the estimation is 0.0003574893934630359
```

In [70]:

```
T_case1_1 = Result('TS',1,200,output='true')
T_case1_2 = Result('TS',2,200,output='true')
```

```
the average gain of {(a1,b1)=(1,1),(a2,b2)=(1,1),(a3,b3)=(1,1)} is 4769.565
the estimated probability is [0.7992281698267963, 0.547038473880975, 0.7087819415330869]
the times of each arm being pulled is [1132231, 8449, 59320]
the mean square error between the truth and the estimation is 0.016452144725152985


the average gain of {(a1,b1)=(601,401),(a2,b2)=(401,601),(a3,b3)=(2,3)} is 4458.055
the estimated probability is [0.6156154979850971, 0.4001996007984022, 0.7337755765573064]
the times of each arm being pulled is [48952, 0, 1151048]
the mean square error between the truth and the estimation is 0.07688699682899439
```

Case 2: dependent on the last pull

We use single quotation marks to represent the last return value of the arm, and then we choose a specific occasion, given A=B=C=1 at the 0 pull, and we have the following probability

- P(A=1|A'=0) = 0.9
- P(A=1|A'=1) = 0.1
- P(B=1|B'=0) = 0.1
- P(B=1|B'=1) = 0.2
- P(C=1) = 0.5

In [71]:

```
arms_result = [1,1,1]

def arms(index):
    '''
    Return the result of the indexth arm being pulled down
    '''
    global arms_result
    if arms_result[0] == 0:
        if random.random() <= 0.9:
            A = 1
        else:
            A = 0
    else:
        if random.random() <= 0.1:
            A = 1
        else:
            A = 0

    if arms_result[0] == 0:
        if random.random() <= 0.1:
            B = 1
        else:
            B = 0
    else:
        if random.random() <= 0.2:
            B = 1
        else:
            B = 0

    if random.random() <= 0.5:
        C = 1
    else:
        C = 0

    arms_result = [A,B,C]
    return arms_result[index-1]
```

```
truth = [0.5,0.11,0.5]
```

test for dependent case 2

In [72]:
```
G_case2_1 = Result('greedy',0.2,200,'true')
G_case2_2 = Result('greedy',0.4,200,'true')
G_case2_3 = Result('greedy',0.6,200,'true')
G_case2_4 = Result('greedy',0.8,200,'true')
```

```
the average gain of epsilon = 0.2 is 2855.585
the estimated probability is [0.4935282251500525, 0.14860485868971596, 0.49460801657992975]
the times of each arm being pulled is [583180, 80215, 536605]
the mean square error between the truth and the estimation is 0.002198454623380822


the average gain of epsilon = 0.4 is 2715.515
the estimated probability is [0.4960148735334606, 0.14900026093361624, 0.4968790517699806]
the times of each arm being pulled is [522445, 160744, 516811]
the mean square error between the truth and the estimation is 0.0019417703164275393


the average gain of epsilon = 0.6 is 2582.79
the estimated probability is [0.49715427890422786, 0.15048848958810937, 0.49817387076771197]
the times of each arm being pulled is [472352, 239360, 488288]
the mean square error between the truth and the estimation is 0.0019845205607314956


the average gain of epsilon = 0.8 is 2438.965
the estimated probability is [0.4984241638516215, 0.14937605411086044, 0.49943158449726405]
the times of each arm being pulled is [432424, 320291, 447285]
the mean square error between the truth and the estimation is 0.001826213241508024
```

In [73]:
```
U_case2_1 = Result('UCB', 2, 200, output='true')
U_case2_2 = Result('UCB', 6, 200, output='true')
U_case2_3 = Result('UCB', 9, 200, output='true')
```

```
the average gain of c = 2 is 2863.06
the estimated probability is [0.44885493994259884, 0.13762207928458334, 0.4998726846984058]
the times of each arm being pulled is [334790, 30618, 833992]
the mean square error between the truth and the estimation is 0.004466433352587635


the average gain of c = 6 is 2542.95
the estimated probability is [0.3788336292624894, 0.13856385721786008, 0.49898860548725826]
the times of each arm being pulled is [348058, 134662, 716680]
the mean square error between the truth and the estimation is 0.01578504909869662


the average gain of c = 9 is 2425.305
the estimated probability is [0.37936748151663674, 0.14009330986371837, 0.4997592981514283]
the times of each arm being pulled is [385758, 190005, 623637]
the mean square error between the truth and the estimation is 0.015728260995636425
```

In [38]:
```
T_case2_1 = Result('TS',1,200,output='true')
T_case2_2 = Result('TS',2,200,output='true')
```

```
the average gain of {(a1,b1)=(1,1),(a2,b2)=(1,1),(a3,b3)=(1,1)} is 2988.3
```

```
the estimated probability is [0.4889633558667175, 0.15725831395316553, 0.4921184085101536]
the times of each arm being pulled is [590764, 4024, 605212]
the mean square error between the truth and the estimation is 0.007537995362406057


the average gain of {(a1,b1)=(601,401),(a2,b2)=(401,601),(a3,b3)=(2,3)} is 2996.82
the estimated probability is [0.5156638478443862, 0.4001996007984022, 0.478661626584705]
the times of each arm being pulled is [1037675, 0, 162325]
the mean square error between the truth and the estimation is 0.08720426936317761
```

Evaluation and analysis of the results of independence

No matter what independent case we set, we can always calculate the margin probability of each arm. This makes the independence or not have no impact on the operation of the algorithm, because after a certain amount of exploration experiments, the algorithm can always approach the real margin probability. At this time, treating the arm as independent has no impact on the expected results.

That is, taking the dependent case is equivalent to replacing the arm probability given by the problem, and then still simulate it according to the independent case. The results obtained by each algorithm are consistent with the above discussion.

Moreover, in the actual situation, we will not know whether the arm is independent or not, so the sub simulation of non independent situation is also appropriate to the reality.

With Constraints

It is assumed that each time the arm is pulled, it costs $m, which makes it possible for us to go bankrupt due to too much exploration during the experiment, thus losing the possibility of expansion. Therefore, we need an index to limit when it is appropriate to explore and when it is appropriate to expand.

In an extreme case, participants were given 50 dollars at the beginning of the experiment, and it cost 0.775 dollar to pull the arm each time. Thus, the main code is updated as follows:

In [75]:

```
class Constraints(Result):
    def __init__(self, method, parameter, repeat, output='false', N=6000):
        super().__init__(method, parameter, repeat, output=output, N=N)

    def money_output(self):
        self.money = []
        outputed = 0
        for i in range(self.N):
            temp = 0
            for j in range(self.repeat):
                temp += self.optimal_gain_ratio_total[j][i] *(i+1)*0.8
            temp = temp / self.repeat+50-0.775*(i+1)
            if temp >= 0:
                self.money.append(temp)
            else:
                self.money.append(0)
            if outputed == 0 and temp <= 0:
                print("with {} , bankrupts at the {} time".format(self.target, i))
                outputed = 1
        if outputed ==0:
            print("with {} , does not bankrupt".format(self.target))
```

Bankrupt Examples

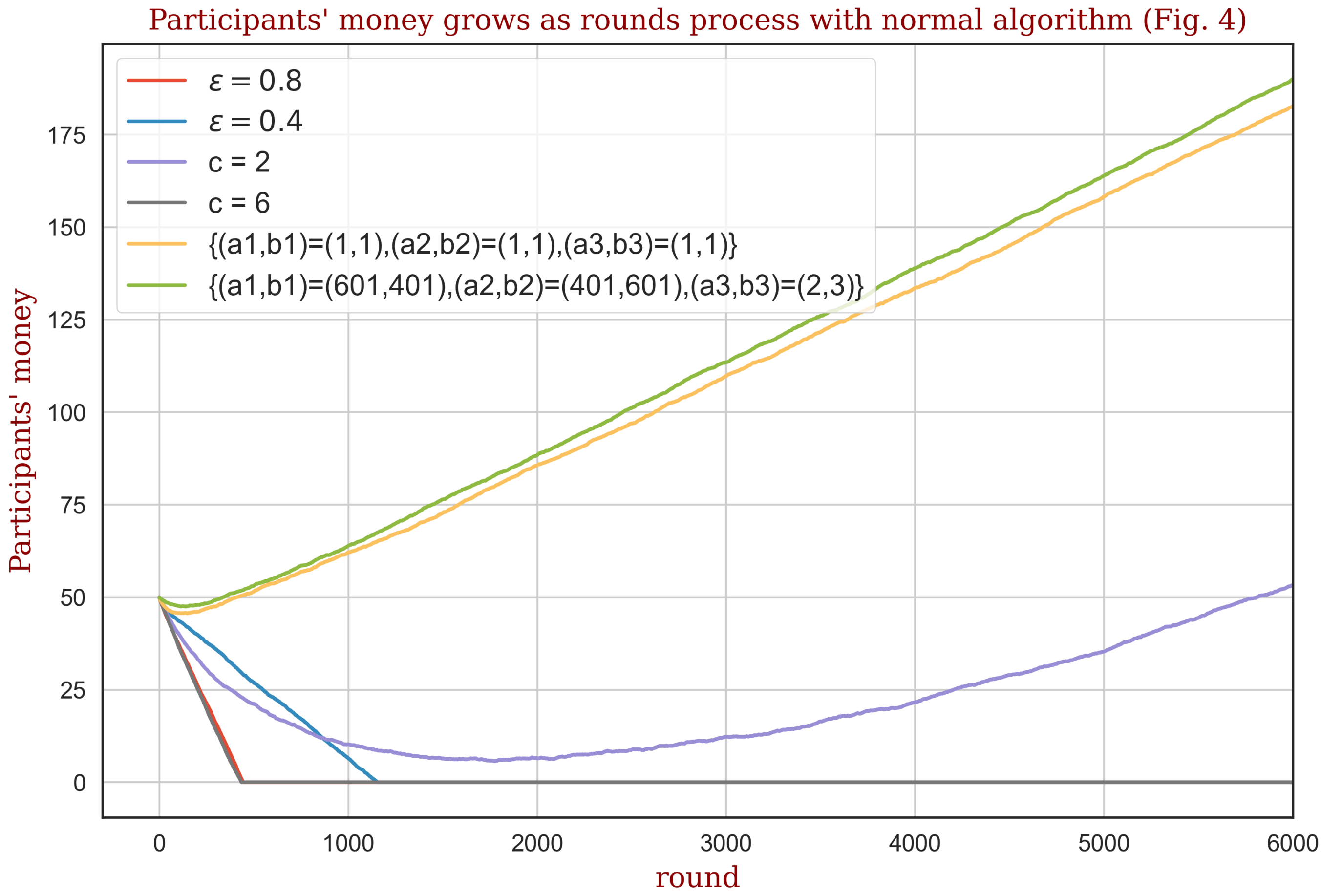
Taking several specific algorithms and parameters as examples, we can see that the algorithm has probability of bankruptcy after adding the cost of pulling arm.

In [76]:
```
G1_0 = Constraints('greedy',0.8,200)
G1_0.money_output()

G2_0 = Constraints('greedy',0.4,200)
G2_0.money_output()

U1_0 = Constraints('UCB',2,200)
U1_0.money_output()

U2_0 = Constraints('UCB',6,200)
U2_0.money_output()

T1_0 = Constraints('TS',1,200)
T1_0.money_output()

T2_0 = Constraints('TS',2,200)
T2_0.money_output()
```

```
with epsilon = 0.8 , bankrupts at the 444 time

with epsilon = 0.4 , bankrupts at the 1157 time

with c = 2 , does not bankrupt

with c = 6 , bankrupts at the 435 time

with {(a1,b1)=(1,1),(a2,b2)=(1,1),(a3,b3)=(1,1)} , does not bankrupt

with {(a1,b1)=(601,401),(a2,b2)=(401,601),(a3,b3)=(2,3)} , does not bankrupt
```

Visualization

In [77]:
```
fig4, ax4_1 = plt.subplots(1, 1, figsize=(9,6), dpi = 400)

x_rounds1 = np.arange(0,6000,1)

ax4_1.plot(x_rounds1, G1_0.money)
ax4_1.plot(x_rounds1, G2_0.money)
ax4_1.plot(x_rounds1, U1_0.money)
ax4_1.plot(x_rounds1, U2_0.money)
ax4_1.plot(x_rounds1, T1_0.money)
ax4_1.plot(x_rounds1, T2_0.money)
ax4_1.axis(xmax = 6000)
ax4_1.set_title('Participants\' money grows as rounds process with normal algorithm (Fig. 4)', fo
ax4_1.set_xlabel('round', fontdict=font1)
ax4_1.set_ylabel('Participants\' money', fontdict=font1)
ax4_1.legend((r'$\epsilon = 0.8$',r'$\epsilon = 0.4$','c = 2','c = 6','{(a1,b1)=(1,1),(a2,b2)=(1,
ax4_1.grid()

plt.show()
```



Update the algorithm

Based on the above discussion, we look for a parameter to measure our resistance to bankruptcy.

For greedy algorithms, when there is less money, we should expand more and explore less, that is, epsilon should decrease with the decrease of money.

Similarly, for UCB algorithm, the value of c should decrease with the decrease of money. Therefore, in these two algorithms, we try to find our parameters in linear and exponential ways respectively. Then there is the following formula:

$$flag_1 = \frac{money}{expected\ reward}$$

$$flag_2 = 1 - e^{-x*money}$$

$$\epsilon' = \epsilon * flag$$

$$c' = c * flag$$

where we take expected value as 200, x as 0.001 for an example.

For TS algorithm, we should adjust the a priori probability before each experiment according to the amount of money, so that the arm with high probability accounts for a higher possibility of pulling. However, unless the a priori probability deviation is very large, TS algorithm will probably choose the best arm, making the possibility of bankruptcy very low. When the prior probability deviation is great, we should not be limited to adjusting the TS algorithm, but should replace a new algorithm to ignore this deviation.

Linear Way to update the algorithm

In [78]:

```
class Constraints_updated_1(Constraints):
    def __init__(self, method, parameter, repeat, output='false', N=6000):
        super().__init__(method, parameter, repeat, output=output, N=N)
```

```
    def greedy(self, epsilon):
        true_epsilon = epsilon
        money = 50
        optimal_choices_ratio = [] #In the previous index experiments, the ratio of No. 1 arm was
        optimal_gain_ratio = [] #Ratio of income to ideal value after index experiments
        distance = [] #Mean square error between predicted and actual values after index experime
        theta_mean = [0,0,0] # estimation of theta_j
        count = [0,0,0] # count(j)
        gain = 0 # total reward
        result = 0 # the result of the arm of this time
        for t in range(self.N):
            if money >= 0:
                epsilon = true_epsilon * money / 150
                if random.random() <= epsilon:
                    I = random.randint(1,3)
                else:
                    I = arg_max(theta_mean) + 1
                result = arms(I)
                self.countarms[I-1] += 1
                gain += result
                money = money + result - 0.775
                count[I-1] += 1
                theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
                # optimal_choices_ratio
                optimal_choices_ratio.append(count[0] / (t+1))
                # optimal_gain_ratio
                optimal_gain_ratio.append(gain / ((t+1)*0.8))
                # distance
                distance.append(distance_between_truth(theta_mean))
            else:
                result = 0 # The arm cannot be pulled after bankruptcy
                I = 1 # To avoid program errors, set a virtual value
                self.countarms[I-1] += 1
                gain += result
                count[I-1] += 1
                theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
                # optimal_choices_ratio
                optimal_choices_ratio.append(count[0] / (t+1))
                # optimal_gain_ratio
                optimal_gain_ratio.append(gain / ((t+1)*0.8))
                # distance
                distance.append(distance_between_truth(theta_mean))

        for i in range(3):
            self.theta_total[i] += theta_mean[i]
        self.gain_total += gain

        return optimal_choices_ratio, optimal_gain_ratio, distance

    def UCB(self, c):
        true_c = c
        money = 50
        optimal_choices_ratio = [] #In the previous index experiments, the ratio of No. 1 arm was
        optimal_gain_ratio = [] #Ratio of income to ideal value after index experiments
        distance = [] #Mean square error between predicted and actual values after index experime
        theta_mean = [0,0,0] # estimation of theta_j
        count = [0,0,0] # count(j)
        gain = 0 # total reward
        result = 0 # the result of the arm of this time
        for t in range(3):
            count[t] += 1
            result = arms(t+1)
            gain += result
            money = money + result - 0.775
            theta_mean[t] = result
            # optimal_choices_ratio
            optimal_choices_ratio.append(count[0] / (t+1))
            # optimal_gain_ratio
```

```
            optimal_gain_ratio.append(gain / ((t+1)*0.8))
            # distance
            distance.append(distance_between_truth(theta_mean))
        for t in range(3, self.N):
            if money >= 0:
                c = true_c * money / 150
                temp = [0,0,0]
                for j in range(3):
                    temp[j] = theta_mean[j] + c * (2*math.log10(t)/count[j])**0.5
                I = arg_max(temp) + 1
                result = arms(I)
                self.countarms[I-1] += 1
                count[I-1] += 1
                gain += result
                money = money + result - 0.775
                theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
                # optimal_choices_ratio
                optimal_choices_ratio.append(count[0] / (t+1))
                # optimal_gain_ratio
                optimal_gain_ratio.append(gain / ((t+1)*0.8))
                # distance
                distance.append(distance_between_truth(theta_mean))
            else:
                result = 0 # The arm cannot be pulled after bankruptcy
                I = 1 # To avoid program errors, set a virtual value
                self.countarms[I-1] += 1
                gain += result
                count[I-1] += 1
                theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
                # optimal_choices_ratio
                optimal_choices_ratio.append(count[0] / (t+1))
                # optimal_gain_ratio
                optimal_gain_ratio.append(gain / ((t+1)*0.8))
                # distance
                distance.append(distance_between_truth(theta_mean))

        for i in range(3):
            self.theta_total[i] += theta_mean[i]
        self.gain_total += gain

        return optimal_choices_ratio, optimal_gain_ratio, distance
```

Test Code

In [79]:

```
G1_1 = Constraints_updated_1('greedy',0.8,200)
G1_1.money_output()

G2_1 = Constraints_updated_1('greedy',0.4,200)
G2_1.money_output()

U1_1 = Constraints_updated_1('UCB',2,200)
U1_1.money_output()

U2_1 = Constraints_updated_1('UCB',6,200)
U2_1.money_output()
```

```
with epsilon = 0.8 , bankrupts at the 1955 time


with epsilon = 0.4 , bankrupts at the 884 time


with c = 2 , bankrupts at the 2167 time
```

with c = 6 , does not bankrupt

Exponential way to update the algotirhm

In [80]:

```
class Constraints_updated_2(Constraints):
    def __init__(self, method, parameter, repeat, output='false', N=6000):
        super().__init__(method, parameter, repeat, output=output, N=N)

    def greedy(self, epsilon):
        true_epsilon = epsilon
        money = 50
        optimal_choices_ratio = [] #In the previous index experiments, the ratio of No. 1 arm was
        optimal_gain_ratio = [] #Ratio of income to ideal value after index experiments
        distance = [] #Mean square error between predicted and actual values after index experim
        theta_mean = [0,0,0] # estimation of theta_j
        count = [0,0,0] # count(j)
        gain = 0 # total reward
        result = 0 # the result of the arm of this time
        for t in range(self.N):
            if money >= 0:
                epsilon = true_epsilon * (1-math.exp(-0.001*money))
                if random.random() <= epsilon:
                    I = random.randint(1,3)
                else:
                    I = arg_max(theta_mean) + 1
                result = arms(I)
                self.countarms[I-1] += 1
                gain += result
                money = money + result - 0.775
                count[I-1] += 1
                theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
                # optimal_choices_ratio
                optimal_choices_ratio.append(count[0] / (t+1))
                # optimal_gain_ratio
                optimal_gain_ratio.append(gain / ((t+1)*0.8))
                # distance
                distance.append(distance_between_truth(theta_mean))
            else:
                result = 0 # The arm cannot be pulled after bankruptcy
                I = 1 # To avoid program errors, set a virtual value
                self.countarms[I-1] += 1
                gain += result
                count[I-1] += 1
                theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
                # optimal_choices_ratio
                optimal_choices_ratio.append(count[0] / (t+1))
                # optimal_gain_ratio
                optimal_gain_ratio.append(gain / ((t+1)*0.8))
                # distance
                distance.append(distance_between_truth(theta_mean))

        for i in range(3):
            self.theta_total[i] += theta_mean[i]
        self.gain_total += gain

        return optimal_choices_ratio, optimal_gain_ratio, distance

    def UCB(self, c):
        true_c = c
        money = 50
        optimal_choices_ratio = [] #In the previous index experiments, the ratio of No. 1 arm was
        optimal_gain_ratio = [] #Ratio of income to ideal value after index experiments
        distance = [] #Mean square error between predicted and actual values after index experim
        theta_mean = [0,0,0] # estimation of theta_j
        count = [0,0,0] # count(j)
        gain = 0 # total reward
        result = 0 # the result of the arm of this time
```

```
        for t in range(3):
            count[t] += 1
            result = arms(t+1)
            gain += result
            money = money + result - 0.775
            theta_mean[t] = result
            # optimal_choices_ratio
            optimal_choices_ratio.append(count[0] / (t+1))
            # optimal_gain_ratio
            optimal_gain_ratio.append(gain / ((t+1)*0.8))
            # distance
            distance.append(distance_between_truth(theta_mean))
        for t in range(3, self.N):
            if money >= 0:
                c = true_c * (1-math.exp(-0.0001*money))
                temp = [0,0,0]
                for j in range(3):
                    temp[j] = theta_mean[j] + c * (2*math.log10(t)/count[j])**0.5
                I = arg_max(temp) + 1
                result = arms(I)
                self.countarms[I-1] += 1
                count[I-1] += 1
                gain += result
                money = money + result - 0.775
                theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
                # optimal_choices_ratio
                optimal_choices_ratio.append(count[0] / (t+1))
                # optimal_gain_ratio
                optimal_gain_ratio.append(gain / ((t+1)*0.8))
                # distance
                distance.append(distance_between_truth(theta_mean))
            else:
                result = 0 # The arm cannot be pulled after bankruptcy
                I = 1 # To avoid program errors, set a virtual value
                self.countarms[I-1] += 1
                gain += result
                count[I-1] += 1
                theta_mean[I-1] += (result-theta_mean[I-1])/count[I-1]
                # optimal_choices_ratio
                optimal_choices_ratio.append(count[0] / (t+1))
                # optimal_gain_ratio
                optimal_gain_ratio.append(gain / ((t+1)*0.8))
                # distance
                distance.append(distance_between_truth(theta_mean))

        for i in range(3):
            self.theta_total[i] += theta_mean[i]
        self.gain_total += gain

        return optimal_choices_ratio, optimal_gain_ratio, distance
```

Test Code

In [81]:

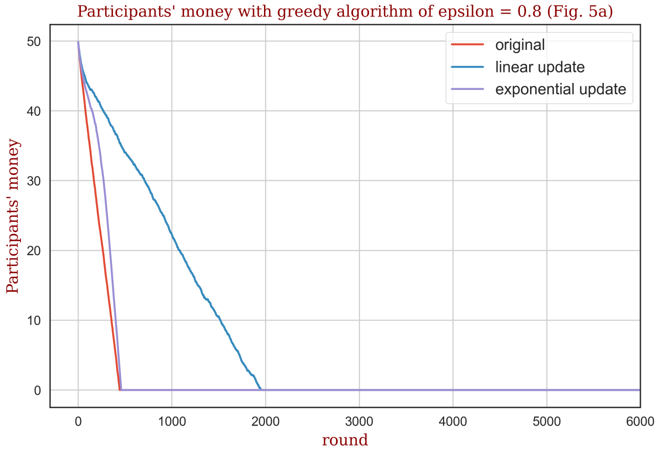
```
G1_2 = Constraints_updated_2('greedy',0.8,200)
G1_2.money_output()

G2_2 = Constraints_updated_2('greedy',0.4,200)
G2_2.money_output()

U1_2 = Constraints_updated_2('UCB',2,200)
U1_2.money_output()

U2_2 = Constraints_updated_2('UCB',6,200)
U2_2.money_output()
```

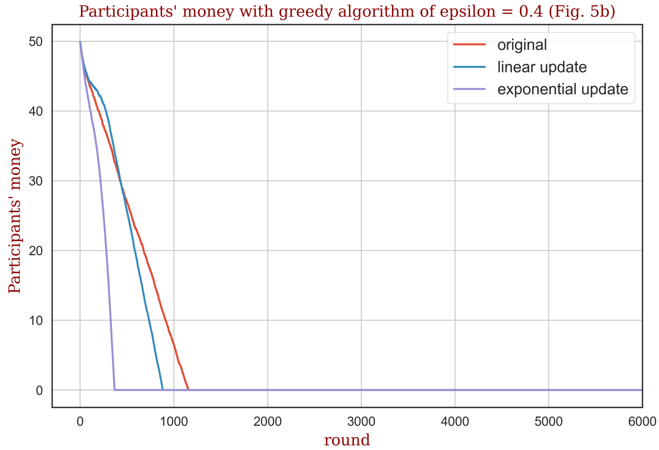
```
with epsilon = 0.8 , bankrupts at the 460 time


with epsilon = 0.4 , bankrupts at the 367 time


with c = 2 , bankrupts at the 500 time


with c = 6 , bankrupts at the 467 time
```

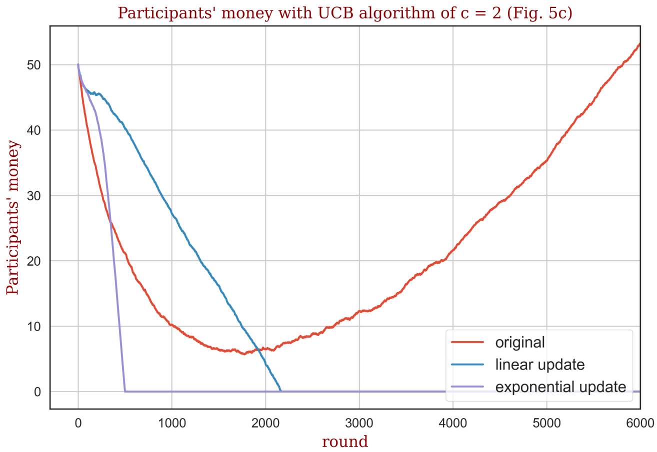
Visualize

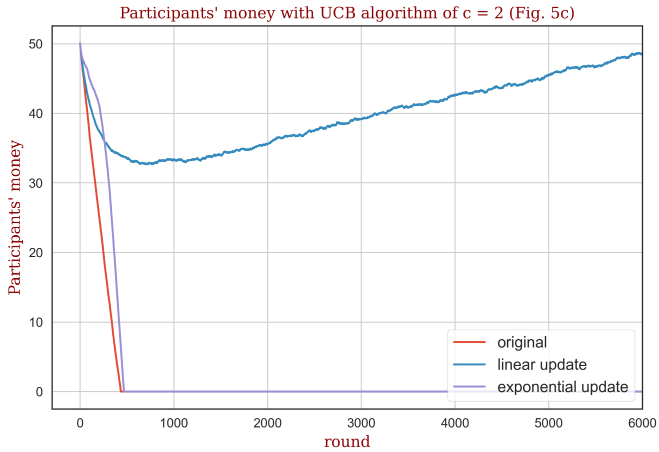
In [82]:

```
fig4, ax4 = plt.subplots(2, 2, figsize=(18,12), dpi = 400)

ax4_1 = ax4[0][0]
ax4_2 = ax4[0][1]
ax4_3 = ax4[1][0]
ax4_4 = ax4[1][1]

x_rounds1 = np.arange(0,6000,1)

ax4_1.plot(x_rounds1, G1_0.money)
ax4_1.plot(x_rounds1, G1_1.money)
ax4_1.plot(x_rounds1, G1_2.money)
ax4_1.axis(xmax = 6000)
ax4_1.set_title('Participants\' money with greedy algorithm of epsilon = 0.8 (Fig. 5a)', fontdic
ax4_1.set_xlabel('round', fontdict=font1)
ax4_1.set_ylabel('Participants\' money', fontdict=font1)
ax4_1.legend(('original','linear update','exponential update'), loc='upper right', fontsize = 'l
ax4_1.grid()

ax4_2.plot(x_rounds1, G2_0.money)
ax4_2.plot(x_rounds1, G2_1.money)
ax4_2.plot(x_rounds1, G2_2.money)
ax4_2.axis(xmax = 6000)
ax4_2.set_title('Participants\' money with greedy algorithm of epsilon = 0.4 (Fig. 5b)', fontdic
ax4_2.set_xlabel('round', fontdict=font1)
ax4_2.set_ylabel('Participants\' money', fontdict=font1)
ax4_2.legend(('original','linear update','exponential update'), loc='upper right', fontsize = 'l
ax4_2.grid()

ax4_3.plot(x_rounds1, U1_0.money)
ax4_3.plot(x_rounds1, U1_1.money)
ax4_3.plot(x_rounds1, U1_2.money)
ax4_3.axis(xmax = 6000)
ax4_3.set_title('Participants\' money with UCB algorithm of c = 2 (Fig. 5c)', fontdict=font1)
ax4_3.set_xlabel('round', fontdict=font1)
ax4_3.set_ylabel('Participants\' money', fontdict=font1)
ax4_3.legend(('original','linear update','exponential update'), loc='lower right', fontsize = 'l
ax4_3.grid()

ax4_4.plot(x_rounds1, U2_0.money)
ax4_4.plot(x_rounds1, U2_1.money)
ax4_4.plot(x_rounds1, U2_2.money)
ax4_4.axis(xmax = 6000)
ax4_4.set_title('Participants\' money with UCB algorithm of c = 2 (Fig. 5c)', fontdict=font1)
ax4_4.set_xlabel('round', fontdict=font1)
ax4_4.set_ylabel('Participants\' money', fontdict=font1)
ax4_4.legend(('original','linear update','exponential update'), loc='lower right', fontsize = 'l
ax4_4.grid()

plt.show()
```









Evaluation

It can be seen from the figure that different update methods have different effects on different parameters and algorithms. This forces us to set different resistance to bankruptcy according to the algorithm when facing the cost arm, so as to obtain better reward.

References

[1] Reinforcement Learning: An Introduction (second edition), R. Sutton & A. Barto, 2018.

[2] Project-Slides (SI140 Fall 2020), Ziyu Shao (ShanghaiTech), 2020.

[3] https://en.wikipedia.org/wiki/Multi-armed_bandit