Performance Evaluation on Classical Algorithms for Multi-Armed Bandit SI140@2021Fall final project

Cheng Peng*

January 16, 2022

Contents

*pengcheng2@shanghaitech.edu.cn ID=2020533068

1 The Overall Performance Comparison

We made a bar chart1 based on the simulation, where the length of each bar is the gap[1] between the oracle value and the average reward of the corresponding algorithm.
Generally speaking, we have TS > UCB > $\epsilon-$Greedy in the sense of average reward, when the parameter is tuned.

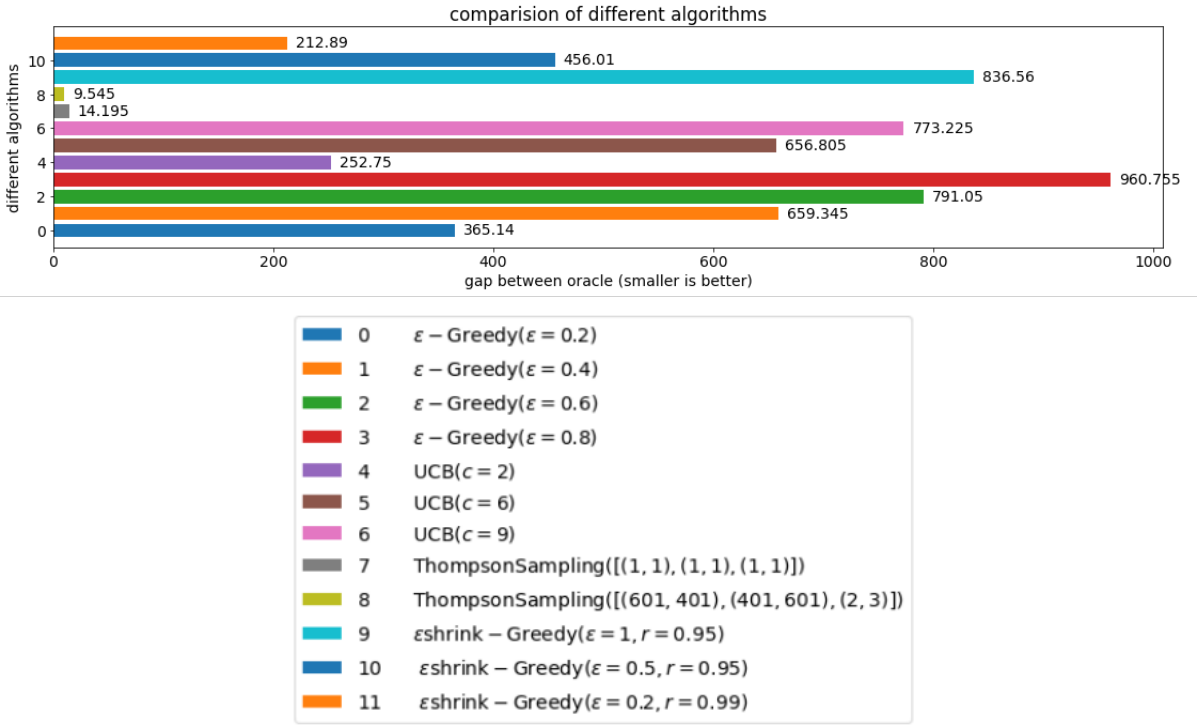


Figure 1: Bar Chart: the performance of each algorithms

To better understand the behavior of each algorithm, the knowledge used to make decision on each step is recorded. We store the estimated θ_js and organize them into scatter plots1.
We can see that *UCB* and *TS* can quickly learn the actual distribution, while ϵ-*Greedy* algorithm takes a long time to converge.
The *Simulated Annealing* algorithm failed to learn the behind distribution. We doubt that it is caused by too short explore phase i.e. ϵ shrinks too fast.

[1] This is also called *regret*, which is the goal to minimize.

Figure 2: The learning curve of different algorithms

2 The Exploration-Exploitation Trade-off and Impact of Parameter Selection

2.1 The E-E trade-off

In MAB problem settings, the agent do not have the real distribution of reward behind each arm, so the optimization and learning process have to be carried out simultaneously.
After some arm pulls, the algorithm has some knowledge on the distributions. We have decide to pull the arm that gives maximal expected reward based on acquired knowledge i.e. exploit or to pull other arms to gain more accurate information of the distributions i.e. explore.
There, we face a dilemma: To maximize the aggregated reward, we want to pull the optimal arm as much as possible. However, to find the optimal arm, we have to spend more chances pulling every arms.

Clearly, a good balance between exploration and exploitation is the key to ideal performance. We will see how the classical algorithms strives to find a proper balance.

2.2 ϵ in ϵ-Greedy algorithm

The ϵ-Greedy algorithm, which is shown in 2.2, is the most straight-forward way to balance exploration and exploitation.

Algorithm 1 $\epsilon-$greedy Algorithm

Initialize $\hat{\theta}(j) = 0, \text{count}(j) = 0, j \in \{1, 2, 3\}$

1: **for** $t = 1, 2 \ldots, N$ **do**

2:

$$I(t) \leftarrow \begin{cases} \underset{j\in\{1,2,3\}}{\arg\max}\ \hat{\theta}(j) & w.p.\ 1-\epsilon \\ \text{randomly chosen from}\{1,2,3\} & w.p.\ \epsilon \end{cases}$$

3: $\text{count}(I(t)) \leftarrow \text{count}(I(t)) + 1$

4: $\hat{\theta}(I(t)) \leftarrow \hat{\theta}(I(t)) + \frac{1}{\text{count}(I(t))}\left[r_{I(t)} - \hat{\theta}(I(t))\right]$

5: **end for**

Roughly, we make $N\epsilon$ trials to learn the distribution and $N(1-\epsilon)$ pulls to maximize the rewards.
With larger ϵ, the algorithm tends to explore more while it prefers to make exploitative decisions.
Finding a proper value of ϵ is somewhat hard in real-world applications. This makes the algorithm less flexible since it can not change the balance between exploration and exploitation adaptively.

We can enhance the algorithm by combining ϵ-Greedy with Simulated Annealing, a traditional algorithm framework for discrete/continuous optimization problems.
We make ϵ decrease as we gain more information of the distributions, this is implemented by making $\epsilon_t = \epsilon_0 r^t$ where r is a real number between 0 and 1. Typically, we choose $r = 0.95$ or $r = 0.99$.
The parameter r controlls the time that the algorithm enter a pure-exploitation phase. The agent can learn more about the distribution when having smaller r.

2.3 c in Upper Confidence Bound algorithm

UCB algorithm, first introduce in [ACF02], can be described by the following pseudo code2.3.

Algorithm 2 UCB Algorithm

for $t = 1, 2, 3$ **do**
    $I(t) \leftarrow t$
    $\text{count}(I(t)) \leftarrow 1$
    $\hat{\theta}(I(t)) \leftarrow r_{I(t)}$
end for
for $t = 4, \ldots, N$ **do**

$$I(t) \leftarrow \underset{j \in \{1,2,3\}}{\arg\max} \left(\hat{\theta}(j) + c \cdot \sqrt{\frac{2 \log t}{\text{count}(j)}} \right)$$

    $\text{count}(I(t)) \leftarrow \text{count}(I(t)) + 1$
    $\hat{\theta}(I(t)) \leftarrow \hat{\theta}(I(t)) + \frac{1}{\text{count}(I(t))} \left[r_{I(t)} - \hat{\theta}(I(t)) \right]$
end for

UCB algorithm employes interval estimation to find the parameter θ_js. We can use Hoeffding bound to find the confidence interval once the confidence level is fixed.
In time slot t, $\hat{\theta}_j + c\sqrt{\frac{2 \log t}{\text{count}_j}}$ is the upper bound of the confidence interval $[\hat{\theta} - \delta, \hat{\theta} + \delta]$. The parameter c is determined by the confidence level.

Consider the scenario where arm k has lower $\hat{\theta}_k$ than other arms while count_k is small. This indicate we haven't gain much information on that arm, we can pull arm k for exploration. In this case, $\sqrt{\frac{2 \log t}{\text{count}_k}}$ is lager than every other arm, allowing UCB algorithm to pick k with greater chances.
As time proceeds and the confidence interval shrinks, the algorithm has enough knowledge on the distributions of arms and makes decisions mainly based on $\hat{\theta}$ i.e. tends to exploit the best arm. This is the way that UCB resolves E-E dilemma.

c correspondes to the confidence level. With larger c, we can enable the algorithm to make more exploration even t is relatively large. This cause the algorithm to converge slowly as 1 shows.
In constrast, with smaller c, we are more confident for the estimation so we tend to exploit more. In the learning history curve, we can observe that $\hat{\theta}$ converges to θ rapidly.

2.4 α_i, β_is in Thompson Sampling algorithm

We found a paper [AG12] that covers the theoretical analysis of Thompson Sampling algorithm2.4.

Algorithm 3 Thompson sampling Algorithm

Initialize Beta parameter $(\alpha_j, \beta_j), j \in \{1, 2, 3\}$

1: **for** $t = 1, 2 \ldots, N$ **do**
2: *# Sample model*
3: **for** $j \in \{1, 2, 3\}$ **do**
4: Sample $\hat{\theta}(j) \sim \text{Beta}(\alpha_j, \beta_j)$
5: **end for**
6: *# Select and pull the arm*

$$I(t) \leftarrow \arg\max_{j \in \{1,2,3\}} \hat{\theta}(j)$$

7: *# Update the distribution*

$$\alpha_{I(t)} \leftarrow \alpha_{I(t)} + r_{I(t)}$$
$$\beta_{I(t)} \leftarrow \beta_{I(t)} + 1 - r_{I(t)}$$

8: **end for**

The main idea of Thompson Sampling is quite simple: use beta-binomial conjugacy to learn and estimate the parameter θ_js. The (α_j, β_j)s stands for the pseudo-count priors of each arm.

Intuitively, if we run TS for enough iterations, the $\hat{\theta}_j = \frac{a_j}{a_j + b_j}$ should eventually converge to θ_j. This can be justified by the learning curve 1. As we can see, the curves plateau and stays around the true value of θ.

However, the prior knowledge still play a crucial role in short runs. For some j, if the prior $\alpha_j + \beta_j$ is large, then we rely more on the prior knowledge rather than the actual rewards. We call this a strong prior.
When we have a strong prior that is far from the true distribution, Thompson Sampling takes great amount of time to learn the actual value of θ. See 1 TS with $\alpha_1, \beta_1 = 601, 401$.
When we have a prior that is close the true distribution, TS performs best. In that case, TS converges at a significant spped and outperforms every other algorithsm. See 1 the first plot for TS.

3 Extension: Dependent Arms

Now we drop the assumption that the reward distribution of arms are independent, which is not realistic. We found several research papers that cover this variant, one with most citation is [PCA07]. We will use the same model

In this variant, we face a MAB with N arms which are grouped into k clusters, where dependencies are allow within a group.

Let $[i]$ be the cluster of arm i and $C[i]$ be the cluster containing i. Let $s_i(t)$ be the number of times arm i gives raeward when pulled "success", and $f_i(t)$ be the number of "failures". We will have

$$s_i(t) \mid \theta_i \sim \text{Bin}(s_i(t) + f_i(t), \theta_i) \quad \theta_i \sim \eta(\pi_{[}i])$$

Where η is the probability distribution of θ and the parameter $\pi_{[}i]$ contains the information of the dependencies within $C[i]$.

We can apply a two-level UCB to exploit the dependencies. In each step, we compute the expected reward and the variance of each group and use UCB to decide the group. Then, we apply UCB again to pick the optimal arms within that group.

To estimate the reward and variance of one group, we take mean among all the arms within that group.

Our two-level UCB is better than plain UCB since it avoids exploring arms in the same group. This gives more "effective" pulls and should generate better performance.

Not Having enough time, I can not implement and run benchmark for this variant.

4 Extension: Bounded Cost

To make the MAB model more practical, we introduce bounded random cost on each arm pull. Multiple research projects have digged into this variant, we picked a popular one [Din+13]. We will use the cost model in this paper and demonstrate the strategy developed from it.

This variant is call MAB-BV[2]. In this problem pulling arm i in time slot t gives reward $r_i(t) \sim \text{Bern}(\theta_i)$ and cost $c_i(t) \sim \text{DUnif}(\frac{1}{m}, \frac{2}{m} \dots \frac{m}{m})$. Moreover, the agent has a limited budget B, which satisfies $m \mid B$.

We modify the UCB algorithm to take the cost and budget into account. In time step t before out-of-budget, we pick

$$\arg\max_j \left(\hat{\theta}_j + \frac{1 + \frac{1}{\lambda}\sqrt{\frac{\log t}{\text{count}_j}}}{\lambda - \sqrt{\frac{\log t}{\text{count}_j}}} \right)$$

where $\lambda \leq \min_j \mathbb{E}(c_j)$ charaterize the lower bound of expected cost.

UCB-BV drops the parameter c in orginal UCB. Instead we now use λ, which corresponds to the average cost, to balance exploration and exploitation.
If we are expecting high cost on each pull i.e. large λ, then we tend to exploit the best arm. On the contary, if we assume that the cost for pulling one arm is low, then we prefer to pull an arm that has less count.

Not Having enough time, I can not implement and run benchmark for this variant.

[2] BV stands for *budget constraint and variable costs*

reference

[ACF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multiarmed bandit problem". In: *Machine learning* 47.2 (2002), pp. 235–256.

[PCA07] Sandeep Pandey, Deepayan Chakrabarti, and Deepak Agarwal. "Multi-armed bandit problems with dependent arms". In: *Proceedings of the 24th international conference on Machine learning*. 2007, pp. 721–728.

[AG12] Shipra Agrawal and Navin Goyal. "Analysis of thompson sampling for the multi-armed bandit problem". In: *Conference on learning theory*. JMLR Workshop and Conference Proceedings. 2012, pp. 39–1.

[Din+13] Wenkui Ding et al. "Multi-armed bandit with budget constraint and variable costs". In: *Twenty-Seventh AAAI Conference on Artificial Intelligence*. 2013.

MAB

January 16, 2022

1 SI140@Fall2021 Final Project: Multi-armed Bandit

1.1 metadata

- abstract: performance evalutaion of classical MAB algorithms
- due date: 2022/01/16 11:59am
- author: spinach/hehelego (彭程 pengcheng2@shanghaitech.edu.cn 2020533068)

1.2 environment

- OS: arch linux
- kernel: 5.16.0-arch1-1
- Arch: x86-64 (amd64)
- python: version **3.10.1** (require 3.10 for convenient type annotation, see PEP 484 and new features in python 3.10)
- ipython: version 7.31.0
- numpy: version 1.21.3
- matplotlib: version 3.5.0

```
# output of "jupyter --version" command
Selected Jupyter core packages...
IPython          : 7.31.0
ipykernel        : 6.6.0
ipywidgets       : 7.6.5
jupyter_client   : 7.1.0
jupyter_core     : 4.9.1
jupyter_server   : not installed
jupyterlab       : not installed
nbclient         : 0.5.4
nbconvert        : 6.1.0
nbformat         : 5.1.3
notebook         : 6.4.4
qtconsole        : not installed
traitlets        : 5.1.0
```

1.3 table of contents

- multi-armed bandit framework setup
- implementation of the three appointed classical bandit learning algorithms
- implementation of several more algorithms

- simulation and visualization

The above sections can be found in this jupyter notebook, while the discussion section can be found in `report.pdf`.

- performance analysis and comparison
- insight on exploration-exploitation trade-off

And we have included two multi-armed bandit variants.

- algorithm for MAB problem with dependent arms
- algorithm for MAB problem with bounded cost

1.4 code style

- multiple statement in one line is allowed
- identifier for variable `snake_case`
- identifier for constant `SANKE_CASE`
- identifier for function `snake_case`
- identifier for class `CamelCase`
- identifier for method `snake_case`
- identifier for property `snake_case`
- identifier for class method `snake_case`
- identifier for static method `snake_case`

1.5 section: MAB setup

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import numpy.random as npr
import numpy.typing as npt
import abc
import typing

# the (pesudo) random number generator
rng = npr.Generator(npr.MT19937(19260817))


def natural_number_stream(start: int = 0) -> typing.Iterator[int]:
    r'''
    generate a natural number stream [start, start+1, start+2, ...]
    '''
    while True:
        yield start
        start += 1


def argmax(xs: typing.Iterable) -> int:
    r'''
    index of the maximum element
```

```
    \text{GetFirstComponent}( max_{\text{CompareSecondComponent}}( index,value␣
↪) )
    '''
    return max(zip(natural_number_stream(), xs), key=lambda kv: kv[1])[0]


def argmin(xs: typing.Iterable) -> int:
    r'''
    index of the minimum element
    \text{GetFirstComponent}( min_{\text{CompareSecondComponent}}( index,value␣
↪) )
    '''
    return max(zip(natural_number_stream(), xs), key=lambda kv: -kv[1])[0]


class Sampling:
    r'''
    sampling named distribution

    methods for single sample generation:
    - uniform(a,b):         $\mathrm{Unif}(a,b)$
    - bernoulli(p):         $\mathrm{Bern}(p)$
    - exponential(rate):    $\mathrm{Expo}(\lambda)$, where rate is the value of␣
↪$\lambda$ parameter.
    - gamma(a,rate):        $\mathrm{Gamma}(a,\lambda)$, where $the value of␣
↪parameter $\lambda$ is rate.
    '''

    @staticmethod
    def uniform(a: float = 0, b: float = 1) -> float:
        return a+(b-a)*rng.random()

    @staticmethod
    def uniform_array(size: int, a: float = 0, b: float = 1) -> npt.NDArray[np.
↪float_]:
        return a+(b-a)*rng.random(size)

    @staticmethod
    def bernoulli(prob: float) -> int:
        return 1 if Sampling.uniform() < prob else 0

    @staticmethod
    def exponential(rate: float = 1) -> float:
        return rng.exponential(1/rate)

    @staticmethod
    def gamma(a: float, rate: float = 1) -> float:
```

```
        return rng.gamma(a, rate)

    @staticmethod
    def beta(a: float, b: float) -> float:
        return rng.beta(a, b)


class MAB:
    r'''
    the multi-armed bandit
    '''

    def __init__(self, theta: list[float]):
        self.theta = theta[:]
        self.arms = len(theta)

    def pull(self, i: int) -> int:
        return Sampling.bernoulli(self.theta[i])

    def oracle_value(self, n: int) -> float:
        return n*max(self.theta)


class Strategy(abc.ABC):
    r'''
    the abstract base class for bandit algorithms
    '''

    def __init__(self, mab: MAB, n: int):
        self.mab = mab
        self.n = n

    @abc.abstractmethod
    def run(self) -> (int, list[list[float]]):
        '''
        perform one simulation: n pulls.
        return a tuple, where the first component is the total rewards
        and the second component is the learned distribution on each step.
        '''
        ...

    @property
    @abc.abstractmethod
    def profile(self) -> str:
        '''
        return the strategy/algorithm name and value of parameters
        '''
```

```
        ...
```

1.6 section: implementation of classical bandit learning algorithms

[2]:
```
class EpsilonGreedy(Strategy):
    def __init__(self, mab: MAB, n: int, eps: float):
        super().__init__(mab, n)
        self._profile = f'$\\epsilon-\mathrm{{Greedy}}(\\epsilon={eps})$'
        self.eps = eps
        self.count = [int(0) for _ in range(mab.arms)]
        self.theta_hat = [float(0) for _ in range(mab.arms)]
        self.estimation: list[list[float]] = [[] for _ in range(n)]

    def run(self) -> tuple[int, list[list[float]]]:
        earn = 0
        for t in range(self.n):
            arm = argmax(self.count)
            if Sampling.bernoulli(self.eps):
                arm = rng.integers(self.mab.arms)
            reward = self.mab.pull(arm)
            earn += reward

            self.count[arm] += 1
            self.theta_hat[arm] += (reward-self.theta_hat[arm])/self.count[arm]
            self.estimation[t] = self.theta_hat[:]

        return (earn, self.estimation)

    @property
    def profile(self) -> str:
        return self._profile


class UpperConfidenceBound(Strategy):
    def __init__(self, mab: MAB, n: int, c: float):
        super().__init__(mab, n)
        self._profile = f'$\mathrm{{UCB}}(c={c})$'
        self.c = c
        self.count = [int(0) for _ in range(mab.arms)]
        self.theta_hat = [float(0) for _ in range(mab.arms)]
        self.estimation: list[list[float]] = [[] for _ in range(n)]

    def run(self) -> tuple[int, list[list[int]]]:
        from math import log, sqrt
        earn = 0
        for t in range(self.mab.arms):
            reward = self.mab.pull(t)
```

```
            earn += reward

            self.count[t] = 1
            self.theta_hat[t] = reward
            self.estimation[t] = self.theta_hat[:]

        for t in range(self.mab.arms, self.n):
            arm = argmax(self.theta_hat[i] + self.c*sqrt(2*log(t+1)/self.
↪count[i])
                         for i in range(self.mab.arms))
            reward = self.mab.pull(arm)
            earn += reward

            self.count[arm] += 1
            self.theta_hat[arm] += (reward-self.theta_hat[arm])/self.count[arm]
            self.estimation[t] = self.theta_hat[:]

        return (earn, self.estimation)

    @property
    def profile(self) -> str:
        return self._profile


class ThompsonSampling(Strategy):
    def __init__(self, mab: MAB, n: int, prior: list[tuple[int, int]]):
        super().__init__(mab, n)
        self.beta_parameters = [list(i) for i in prior]
        self.theta_hat = [float(0) for _ in range(mab.arms)]
        self.estimation: list[list[float]] = [[] for _ in range(n)]
        self._profile = f'$\mathrm{{ThompsonSampling}}({prior})$'

    def run(self) -> tuple[int, list[list[float]]]:
        earn = 0
        for t in range(self.n):
            self.theta_hat = [Sampling.beta(a, b)
                              for (a, b) in self.beta_parameters]
            arm = argmax(self.theta_hat)
            reward = self.mab.pull(arm)
            earn += reward

            self.beta_parameters[arm][0] += reward
            self.beta_parameters[arm][1] += 1-reward
            self.estimation[t] = [a/(a+b) for (a,b) in  self.beta_parameters]

        return (earn, self.estimation)
```

```
    @property
    def profile(self) -> str:
        return self._profile
```

1.7 section: several more algorithms

[3]:
```
class EpsilonDecreaseGreedy(Strategy):
    def __init__(self, mab: MAB, n: int, eps: float, shrink_factor: float):
        super().__init__(mab, n)
        self._profile = f'$\\epsilon\mathrm{{shrink-Greedy}}(\\epsilon={eps},␣
↪r={shrink_factor})$'
        self.eps = eps
        self.shrink_factor = shrink_factor
        self.count = [int(0) for _ in range(mab.arms)]
        self.theta_hat = [float(0) for _ in range(mab.arms)]
        self.estimation: list[list[float]] = [[] for _ in range(n)]

    def run(self) -> tuple[int, list[list[float]]]:
        earn = 0
        for t in range(self.n):
            arm = argmax(self.count)
            if Sampling.bernoulli(self.eps):
                arm = rng.integers(self.mab.arms)
            reward = self.mab.pull(arm)
            earn += reward

            self.count[arm] += 1
            self.theta_hat[arm] += (reward-self.theta_hat[arm])/self.count[arm]
            self.eps *= self.shrink_factor
            self.estimation[t] = self.theta_hat[:]

        return (earn, self.estimation)

    @property
    def profile(self) -> str:
        return self._profile
```

1.8 section: simulation

Benchmark settings:

– $N = 6000$.
– ϵ-greedy with $\epsilon = 0.2, 0.4, 0.6, 0.8$.
– UCB with $c = 2, 6, 9$.
– Thompson Sampling with

$$\{(\alpha_1, \beta_1) = (1, 1), (\alpha_2, \beta_2) = (1, 1), (\alpha_3, \beta_3) = (1, 1)\}$$
$$\{(\alpha_1, \beta_1) = (601, 401), (\alpha_2, \beta_2) = (401, 601), (\alpha_3, \beta_3) = (2, 3)\}$$

Arm j	1	2	3
θ_j	0.8	0.6	0.5

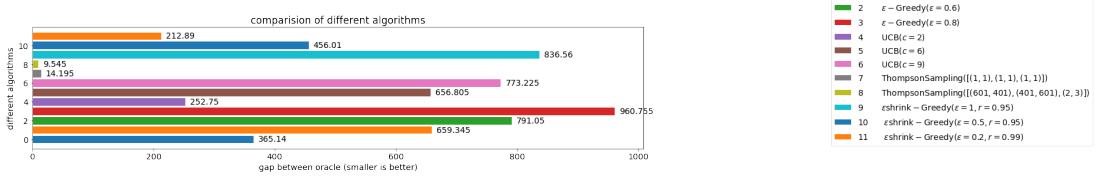
note: the simulation (runs on single CPU core) takes about 1.5min to finish on `AMD Ryzen 7 4800U @ 4.2GHz`

[4]:
```
plt.rcParams['font.size'] = 14
plt.rcParams["figure.figsize"] = [20, 6]
plt.rcParams["figure.autolayout"] = True

RUNS = 200
N = 6000
mab = MAB([0.8, 0.6, 0.5])

oracle_value = mab.oracle_value(N)
average = lambda xs: sum(xs) / len(xs)


def strategies() -> list[Strategy]:
    eps_gre: list[Strategy] = [
        EpsilonGreedy(mab, N, 0.2),
        EpsilonGreedy(mab, N, 0.4),
        EpsilonGreedy(mab, N, 0.6),
        EpsilonGreedy(mab, N, 0.8),
    ]
    ucb: list[Strategy] = [
        UpperConfidenceBound(mab, N, 2),
        UpperConfidenceBound(mab, N, 6),
        UpperConfidenceBound(mab, N, 9),
    ]
    ts: list[Strategy] = [
        ThompsonSampling(mab, N, [(1, 1), (1, 1), (1, 1)]),
        ThompsonSampling(mab, N, [(601, 401), (401, 601), (2, 3)]),
    ]
    eps_dec_gre: list[Strategy] = [
        EpsilonDecreaseGreedy(mab, N, 1, 0.95),
        EpsilonDecreaseGreedy(mab, N, 0.5, 0.95),
        EpsilonDecreaseGreedy(mab, N, 0.2, 0.99),
    ]
    return eps_gre + ucb + ts + eps_dec_gre

def once_reward() -> dict[str, int]:
```

```
    return {
        i.profile : i.run()[0]
        for i in strategies()
    }

benchmark_reward: dict[str,list[int]] = {i.profile : [] for i in strategies()}
for _ in range(RUNS):
    out = once_reward()
    for (k,v) in out.items():
        benchmark_reward[k].append(v)
```

1.8.1 sub section: regret comparison

[5]:
```
# comparing average reward

fig, ax = plt.subplots()
ax.set_title('comparison of different algorithms')
ax.set_xlabel('gap between oracle (smaller is better)')
ax.set_ylabel('different algorithms')
for (i, (algo, rewards)) in enumerate(benchmark_reward.items()):
    avg = average(rewards)
    bar = ax.barh(i, oracle_value-avg, label=f'{i}\t{algo}')
    ax.bar_label(bar, padding=8)
ax.legend(bbox_to_anchor=(1.3, 1.5))

plt.show()
```



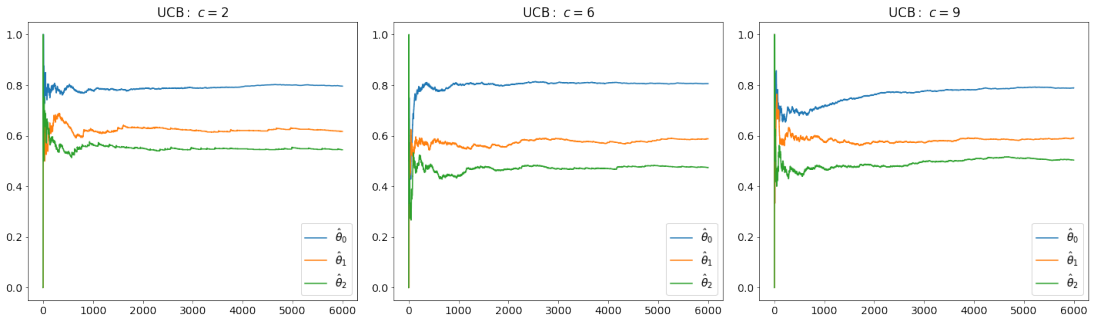
Based on the simulation result, we can conclude that `Thomposon Sampling` outperforms the `epsilon Greedy` and `Upper Confidence Bound`.

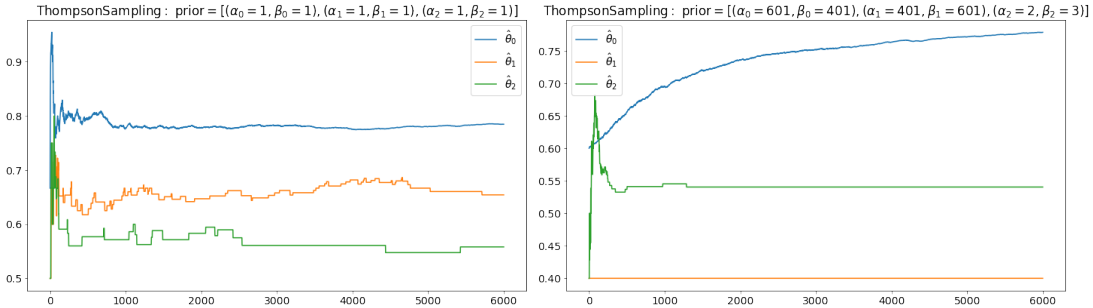
1.8.2 sub section: learning curve

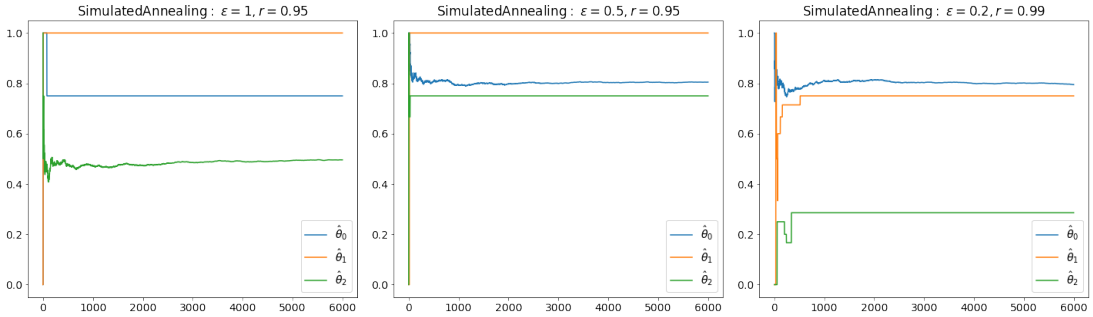
[6]:
```
# compare learning accuracy and speed of convergence
eps_gre: list[Strategy] = [
    EpsilonGreedy(mab, N, 0.2),
    EpsilonGreedy(mab, N, 0.4),
    EpsilonGreedy(mab, N, 0.6),
    EpsilonGreedy(mab, N, 0.8),
```

```
]
ucb: list[Strategy] = [
    UpperConfidenceBound(mab, N, 2),
    UpperConfidenceBound(mab, N, 6),
    UpperConfidenceBound(mab, N, 9),
]
ts: list[Strategy] = [
    ThompsonSampling(mab, N, [(1, 1), (1, 1), (1, 1)]),
    ThompsonSampling(mab, N, [(601, 401), (401, 601), (2, 3)]),
]
eps_dec_gre: list[Strategy] = [
    EpsilonDecreaseGreedy(mab, N, 1, 0.95),
    EpsilonDecreaseGreedy(mab, N, 0.5, 0.95),
    EpsilonDecreaseGreedy(mab, N, 0.2, 0.99),
]

# epsilon greedy
fig, ax = plt.subplots(1,len(eps_gre))
for (i, algo) in enumerate(eps_gre):
    ax[i].set_title(f'$\\epsilon-\\mathrm{{Greedy}}:\\ \\epsilon={algo.eps}$')
    history = np.array(algo.run()[1]).transpose()
    for (j, data) in enumerate(history):
        ax[i].plot(data, label=f'$\\hat\\theta_{j}$')
    ax[i].legend()

# UCB
fig, ax = plt.subplots(1,len(ucb))
for (i, algo) in enumerate(ucb):
    ax[i].set_title(f'$\\mathrm{{UCB}}:\\ c={algo.c}$')
    history = np.array(algo.run()[1]).transpose()
    for (j, data) in enumerate(history):
        ax[i].plot(data, label=f'$\\hat\\theta_{j}$')
    ax[i].legend()

# Thompson Sampling
fig, ax = plt.subplots(1,len(ts))
for (i, algo) in enumerate(ts):
    prior = ', '.join(f'(\\alpha_{i}={a},\\beta_{i}={b})' for (i,(a,b)) in␣
 ↪enumerate(algo.beta_parameters))
    ax[i].set_title(f'$\\mathrm{{ThompsonSampling}}:\\␣
 ↪\\mathrm{{prior}}=\\left[{prior}\\right]$')
    history = np.array(algo.run()[1]).transpose()
    for (j, data) in enumerate(history):
        ax[i].plot(data, label=f'$\\hat\\theta_{j}$')
    ax[i].legend()

# epsilon decrease greedy
```

```
fig, ax = plt.subplots(1,len(eps_dec_gre))
for (i, algo) in enumerate(eps_dec_gre):
    ax[i].set_title(f'$\\mathrm{{SimulatedAnnealing}}:\\ \\epsilon={algo.
 ↪eps},r={algo.shrink_factor}$')
    history = np.array(algo.run()[1]).transpose()
    for (j, data) in enumerate(history):
        ax[i].plot(data, label=f'$\\hat\\theta_{j}$')
    ax[i].legend()

# show all the figures
plt.show()
```







1.9 section: discussion, extension and generalization

In this section, we will analysis the impact of each (hyper-)parameters and demonstrate our insight on the exploration-exploitation tradeoff.
After that, we are to introduce two extended variant of the multi-armed bandit problem and modify the UCB algorithm for the new problem settings.

see `report.pdf`