Probability & Statistics for EECS: Final Project

Due on June 10, 2023 at 23:59

Name: **SiHan Lu, XinHe Cheng**
Student ID: 2021533086, 2021533093

Part 1.1: Oracle value

The oracle value is the theoretically maximized expectation of aggregate rewards over N timeslots. In this case, since Arm1 has the biggest rewarded probability, we should choose this arm apparently. Then the oracle value is

$$oracle = 0.7 \times 5000 = 3500$$

Part 1.2: Implement three classical bandit algorithms

Implement three classical bandit algorithms with following settings:
$N = 5000$
ϵ-greedy with $\epsilon \in \{0.1, 0.5, 0.9\}$.
UCB with $c \in \{1, 5, 10\}$.
TS with
$\{(\alpha_1, \beta_1) = (1, 1), (\alpha_2, \beta_2) = (1, 1), (\alpha_3, \beta_3) = (1, 1)\}$
$\{(\alpha_1, \beta_1) = (601, 401), (\alpha_2, \beta_2) = (401, 601), (\alpha_3, \beta_3) = (2, 3)\}$

(a) ϵ-greedy Algorithm
We can get the result of the reward:
When $\epsilon = 0.1$, Total Reward is 3408.
When $\epsilon = 0.5$, Total Reward is 3125.
When $\epsilon = 0.9$, Total Reward is 2753.

(b) UCB Algorithm
We can get the result of the reward:
UCB with c = 1: Total Reward = 3492
UCB with c = 5: Total Reward = 3137.
UCB with c = 10: Total Reward = 2831.

(c) Thompson sampling Algorithm
We can get the result of the reward:
TS with parameters (a1, b1) = (1, 1), (a2, b2) = (1, 1), (a3, b3) = (1, 1): Total Reward = 3476.
TS with parameters (a1, b1) = (601, 401), (a2, b2) = (401, 601), (a3, b3) = (2, 3): Total Reward = 3507.

Part 1.3: Average Reward and Average Regret

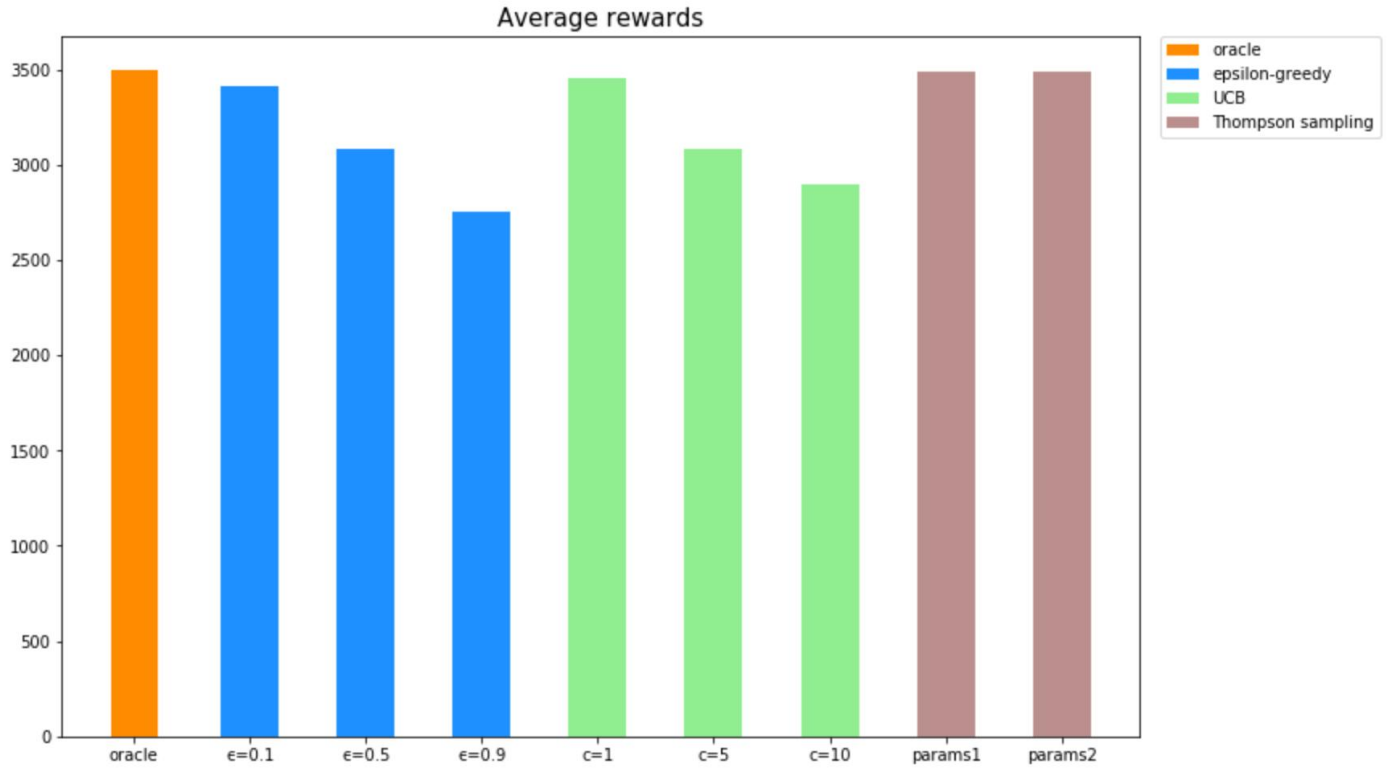
Regard each of the above setting as an experiment. Run each experiment 200 independent trials.

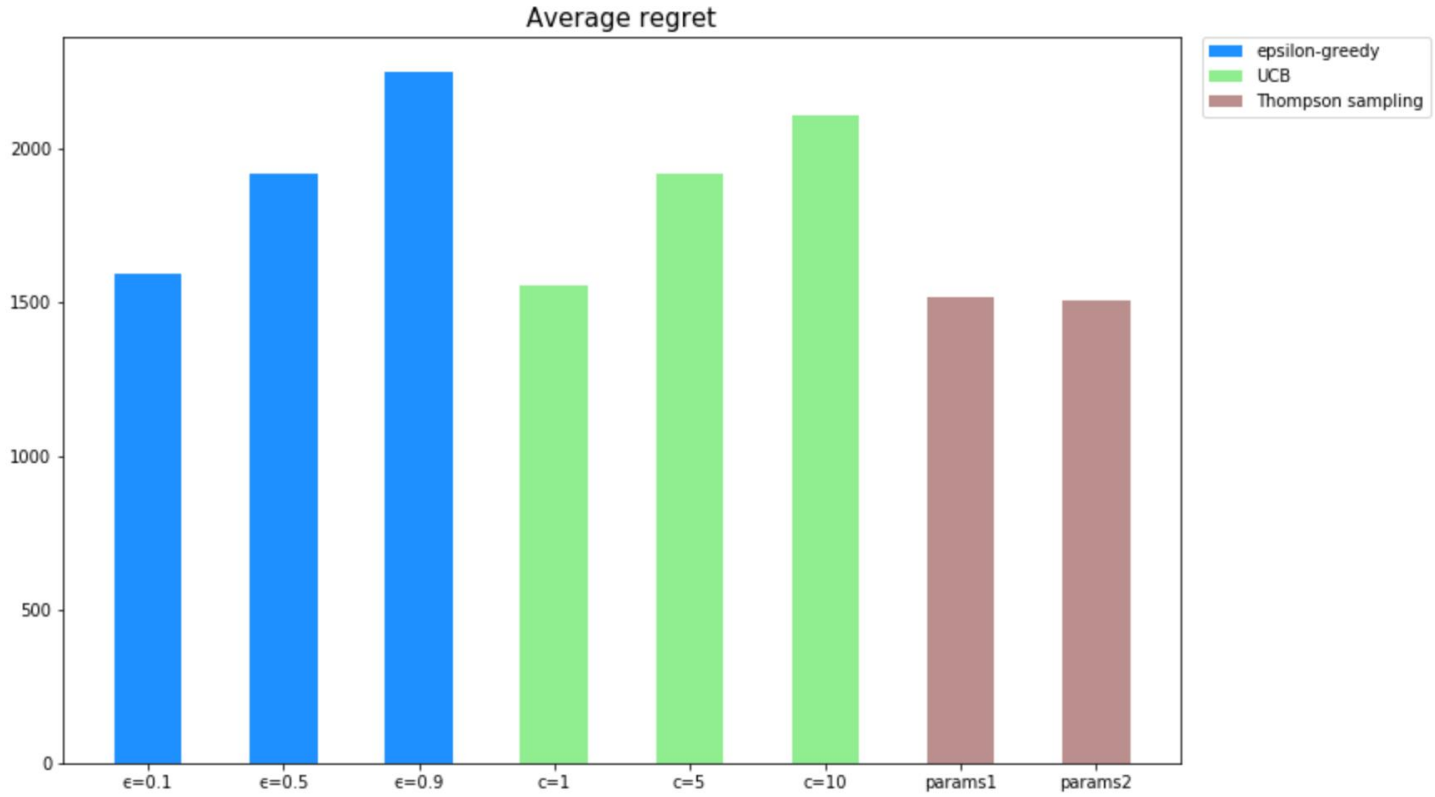
(a) ϵ-greedy Algorithm
When $\epsilon = 0.1$, ϵ-greedy Algorithm's average total-reward is 3083.19, average total-regret is 1920.11.
When $\epsilon = 0.5$, ϵ-greedy Algorithm's average total-reward is 3078.78, average total-regret is 1924.81.
When $\epsilon = 0.9$, ϵ-greedy Algorithm's average total-reward is 2751.93, average total-regret is 2252.355.

(b) UCB Algorithm
When c = 1, UCB Algorithm's average total-reward is 3451.01,average total-regret is 1553.17.
When c = 5, UCB Algorithm's average total-reward is 3081.66,average total-regret is 1916.23.
When c = 10, UCB Algorithm's average total-reward is 2894.25,average total-regret is 2106.67.

(c) Thompson sampling Algorithm
When (α1, β1) = (1, 1), (α1, β1) = (1, 1), (α3, β3) = (1, 1),
TS Algorithm's average total-reward is 3485.045, average total-regret is 1516.67.

When $(\alpha 1, \beta 1) = (601, 401)$, $(\alpha 3, \beta 3) = (401, 601)$, $(\alpha 3, \beta 3) = (2, 3)$,
TS Algorithm's average total-reward is 3488.375,average total-regret is 1504.975.





Part 1.4: Comparison

Part 1.4.1: Gap

Compute the gaps between the algorithm outputs and the oracle value.

(a) ϵ-greedy Algorithm
When ϵ is 0.1, the gap between the algorithm output and the oracle value is 85.96500000000015.
When ϵ is 0.5, the gap between the algorithm output and the oracle value is 416.80999999999995.
When ϵ is 0.9, the gap between the algorithm output and the oracle value is 748.0700000000002.

(b) UCB Algorithm
When c is 1, the gap between the algorithm output and the oracle value is 48.98999999999978.
When c is 5, the gap between the algorithm output and the oracle value is 418.34000000000015.
When c is 10, the gap between the algorithm output and the oracle value is 605.75.

(c) Thompson Sampling Algorithm
When TS with parameters ((1,1),(1,1),(1,1)),
the gap between the algorithm output and the oracle value is 14.954999999999927.
When TS with parameters ((601, 401),(401, 601),(2,3)),
the gap between the algorithm output and the oracle value is 11.625.

Part 1.4.2: Best algorithm

Since Thompson Sampling Algorithm has the minimum regret and maximum reward among three algorithms and the results of Thompson Sampling Algorithm are relatively stable, Thompson Sampling Algorithm is the best.

Part 1.4.3: The impacts of ϵ,c,α_j,β_j

(a) ϵ in greedy:

In the ϵ-greedy algorithm, the value of ϵ determines the trade-off between exploration and exploitation. A higher value of ϵ indicates a higher exploration probability, where the system is more inclined to try new actions to gather more information. On the other hand, a lower value of ϵ indicates a higher exploitation probability, where the system is more inclined to choose the action it currently believes to be the best for immediate rewards. The choice of directly affects the balance between exploration and exploitation in the algorithm. A higher ϵ value facilitates the exploration of new actions but may miss the current optimal solution, while a lower ϵ value focuses more on exploiting the action believed to be the best but may miss other potentially better solutions. Selecting an appropriate ϵ value aims to maximize the long-term cumulative reward. The importance of ϵ lies in balancing the process of exploration and exploitation, where a higher ϵ value leans towards exploration, and a lower ϵ value leans towards exploitation.

(b) c in UCB(Upper Confidence Bound):

The value of parameter c affects the balance between exploration and exploitation. A larger value of c will make the algorithm more inclined to explore actions with higher uncertainty, while a smaller value of c will prioritize exploiting actions with higher current estimates. Additionally, a larger value of c encourages the algorithm to emphasize exploration, attempting a greater variety of actions to gather more information. However, excessively large values of c may lead to over-exploration, reducing the efficiency of utilizing known

information. Furthermore, c also influences the initial upper confidence bounds for each action. A higher value of c encourages more exploration in the initial stages. As the algorithm executes and data accumulates, the confidence bounds gradually shrink, and the algorithm becomes more inclined to exploit better-known actions.

(c) α,β in Thompson Sampling:

The Thompson sampling algorithm utilizes random sampling and posterior probability updates for action selection with the aim of maximizing cumulative rewards in uncertain environments. By performing random sampling based on the posterior probability distribution of each action, the Thompson sampling algorithm explores in a probabilistic manner and continuously updates the probability distribution based on observed outcomes.This allows the algorithm to gradually adjust the preference for action selection, achieving a better balance between exploration and exploitation to maximize long-term cumulative rewards.

Part 1.5: trade-off in bandit algorithms

The exploration-exploitation trade-off is the core idea in bandit algorithms. It involves initially conducting a certain number of experiments to gather sufficient information. Then, based on this information, the algorithm identifies the optimal reward choice and subsequently exploits it in future selections until the algorithm concludes. During the exploration process, the posterior probabilities need to be updated after each trial. This gradually brings the probabilities of each action closer to their true values, resulting in maximizing the expected return. Exploration and exploitation compete with each other as objectives. Excessive exploration may cause missed opportunities with known better actions, while excessive exploitation may overlook the chance to explore unknown actions. The design of the algorithm requires a balancing act between exploration and exploitation to maximize long-term cumulative rewards. In the initial stages, emphasis should be on exploration to gather more information and data. As the algorithm progresses and data accumulates, it should gradually shift towards exploiting known better actions to improve the efficiency of cumulative rewards.

Part 1.6: The dependent case

Since we have learned that Thompson Sampling Algorithm is the best of the three algorithms, we would like to build an algorithm based on this conclusion that is suitable for the dependent case.Every time we pull one arm, in the original algorithm, we only update the corresponding arm each time.But in the dependent case,we also update the other two α and β.We add reward to other two αs and add (1-reward) to other two βs each time.

Part 2.1: design simulations to check the behavior of this policy

```
def bayesian_bandit(N, gamma, a1, a2, b1, b2):
    # 初始化参数和变量
    alpha=[a1, a2]
    beta=[b1, b2]
    total_reward = 0
    count=[0, 0]
    # 执行拉动并更新参数
    for t in range(N):
        theta1 = alpha[0] / (alpha[0] + beta[0])
        theta2 = alpha[1] / (alpha[1] + beta[1])
        arm = 1 if theta1 >= theta2 else 2
        # 根据所选的臂进行采样
        if arm == 1:
            probabilities = [1 - theta1, theta1]
        else:
            probabilities = [1 - theta2, theta2]
        result = random.choices([0, 1], probabilities)[0]
        # 根据结果更新Beta分布的参数
        if arm == 1:
            alpha[0] += result
            beta[0] += 1 - result
            count[0]+=1
        else:
            alpha[1] += result
            beta[1] += 1 - result
            count[1]+=1

        # 如果结果为1（表示成功拉动），根据时间步骤更新总奖励
        if result == 1:
            total_reward += gamma ** t
#             if arm == 1:
#                 alpha[0] += result
#                 beta[0] += 1 - result
#                 count[0]+=1
#             else:
#                 alpha[1] += result
#                 beta[1] += 1 - result
#                 count[1]+=1

    return total_reward, count

print("total_reward, count")
total=0
for i in range(200):
    total_reward, count=bayesian_bandit(5000, 0.9, 1, 1, 1, 1)
    print(total_reward, count)
    total+=total_reward
average_reward=total/200
print(average_reward)
```

```
6.327703758488667
```

Part 2.2: provide an example to show why above policy is not optimal

If only the above strategy is used, that is, the arm with the highest current estimated probability is selected for pulling each time, the better arm may be missed because the success probability of the arm is not estimated accurately. For example, if we have two arms, we take $\theta 1$ and $\theta 2$ as 0.5 and 0.6 respectively, but take $\alpha 1$, $\alpha 2$, $\beta 1$, $\beta 2$ as 5, 1, 1, 5 respectively, which means we will pull the arm in the first few trials with a low probability of success $\theta 1 = 0.5$. By always choosing the first arm to pull, we miss the opportunity to explore the second arm and get a bigger reward.So that, such policy is not optimal.

Part 2.3: proof

If $\frac{\alpha_i}{\alpha_i+\beta_i}$ represents the probability of success and $\frac{\beta_i}{\alpha_i+\beta_i}$ represents the probability of failure, we can denote the reward as $\gamma 0 = 1$ if the first attempt is successful. If it fails, the reward is 0. Additionally, the parameters $\alpha 1$, $\alpha 2$, $\beta 1$, $\beta 2$ are updated after each success or failure. Therefore, the equation can be written as follows:

$$R_1(\alpha_1, \beta_1) = \frac{\alpha_1}{\alpha_1+\beta_1}[1 + \gamma R_1(\alpha_1+1, \beta_1)] + \frac{\beta_1}{\alpha_1+\beta_1}[\gamma R_1(\alpha_1, \beta_1+1)]$$

$$\Rightarrow R_1(\alpha_1, \beta_1) = \frac{\alpha_1}{\alpha_1+\beta_1}[1 + \gamma R_1(\alpha_1, \beta_1)] + \frac{\beta_1}{\alpha_1+\beta_1}[\gamma R_1(\alpha_1, \beta_1)]$$

$$R_2(\alpha_2, \beta_2) = \frac{\alpha_2}{\alpha_2+\beta_2}[1 + \gamma R_2(\alpha_2+1, \beta_2)] + \frac{\beta_2}{\alpha_2+\beta_2}[\gamma R_2(\alpha_2, \beta_2+1)]$$

$$\Rightarrow R_2(\alpha_2, \beta_2) = \frac{\alpha_2}{\alpha_2+\beta_2}[1 + \gamma R_2(\alpha_2, \beta_2)] + \frac{\beta_2}{\alpha_2+\beta_2}[\gamma R_2(\alpha_2, \beta_2)]$$

To maximize our reward, we should choose the larger value between R1(α_1, β_1) and R2(α_2, β_2) given the parametersα_1,β_1,α_2, and β_2. Therefore:

$$R(\alpha_1, \beta_1, \alpha_2, \beta_2) = \max\{R_1(\alpha_1, \beta_1), R_2(\alpha_2, \beta_2)\}$$

Part 2.4: solve it exactly or approximately

When number of trials are large, the addition reward of γ^n doesn't influence the consequence.If $\alpha_i+\beta_i$¿100, its R equals to 0. $\alpha_i+\beta_i$=100, its R equals to $\frac{\alpha_i}{\alpha_i+\beta_i}$. Let $0 < gamma < 1$, where $gamma^{100} \approx 0$, so we can ignore it.
Thus, when $\alpha_i+\beta_i$ ¿ 100, if $R_1 > R_2$, we select the larger value of R1.

$$R_1(\alpha_1, \beta_1) = \frac{\alpha_1}{\alpha_1+\beta_1}[1 + \gamma R_1(\alpha_1+1, \beta_1)] + \frac{\beta_1}{\alpha_1+\beta_1}[\gamma R_1(\alpha_1, \beta_1+1)]$$

$$\Rightarrow R_1(\alpha_1, \beta_1) = \frac{\alpha_1}{\alpha_1+\beta_1}[1 + \gamma R_1(\alpha_1, \beta_1)] + \frac{\beta_1}{\alpha_1+\beta_1}[\gamma R_1(\alpha_1, \beta_1)]$$

$$\Rightarrow R(\alpha_1, \beta_1, \alpha_2, \beta_2) = R_1(\alpha_1, \beta_1) = \frac{\alpha_1}{(\alpha_1+\beta_1)(1-\gamma)}$$

and,if $R_1 < R_2$, we select the larger value of R2.

$$R_2(\alpha_2, \beta_2) = \frac{\alpha_2}{\alpha_2+\beta_2}[1 + \gamma R_2(\alpha_2+1, \beta_2)] + \frac{\beta_2}{\alpha_2+\beta_2}[\gamma R_2(\alpha_2, \beta_2+1)]$$

$$\Rightarrow R_2(\alpha_2, \beta_2) = \frac{\alpha_2}{\alpha_2+\beta_2}[1 + \gamma R_2(\alpha_2, \beta_2)] + \frac{\beta_2}{\alpha_2+\beta_2}[\gamma R_2(\alpha_2, \beta_2)]$$

$$\Rightarrow R(\alpha_1, \beta_1, \alpha_2, \beta_2) = R_2(\alpha_2, \beta_2) = \frac{\alpha_2}{(\alpha_2+\beta_2)(1-\gamma)}$$

If $\alpha_i+\beta_i$¡100, we can initialize it using a recursive equation.

Part 2.5

```
def TS(n, a, b, gamma, times):
    bandit_means = [0.0, 0.0]
    num_arms = 2
    arm_counts = np.zeros(num_arms)
    arm_successes = np.ones(num_arms)
    arm_successes[0] = a[0]
    arm_successes[1] = a[1]
    arm_failures = np.ones(num_arms)
    arm_failures[0] = b[0]
    arm_failures[1] = b[1]
    total_reward = 0
    max_reward = 0
    total_regret = 0
    step = 0  # 迭代步数

    for t in range(times):
        arm_probs = np.random.beta(arm_successes, arm_failures)
        chosen_arm = np.argmax(arm_probs)
        reward = 0
        if np.random.rand() <= chosen_arm:
            reward = 1
        arm_counts[chosen_arm] += 1
        if reward == 1:
            arm_successes[chosen_arm] += 1
            total_reward += gamma ** step
        else:
            arm_failures[chosen_arm] += 1
        if reward > max_reward:
            max_reward = reward
        total_regret += max_reward - gamma ** step
        step += 1

    arm_counts = np.zeros(num_arms)  # 重置计数器
```

```
average_reward: 8.465576230128812
```

```
    for i in range(times, n):
        arm_probs = np.random.beta(arm_successes, arm_failures)
        chosen_arm = np.argmax(arm_probs)
        arm_counts[chosen_arm] += 1
        reward = 0
        if np.random.rand() <= arm_probs[chosen_arm]:
            reward = 1
        if reward == 1:
            arm_successes[chosen_arm] += 1
            total_reward += gamma ** step
        else:
            arm_failures[chosen_arm] += 1
        if reward > max_reward:
            max_reward = reward
        total_regret += max_reward - gamma ** step
        step += 1

    return total_reward, total_regret, arm_counts

N = 5000
param_settings = [((1, 1), (1, 1))]

for A, B in param_settings:
    total = 0
    for i in range(200):
        a = [A[0], B[0]]
        b = [A[1], B[1]]
        total_reward, total_regret, arm_counts = TS(N, a, b, 0.9, 10)
        total += total_reward
        print(f"TS with parameters (a1, b1) = {A}, (a2, b2) = {B}: Total Reward = {total_reward}, arm_counts: {arm_counts}")
    average_reward = total / 200
    print("average_reward:", average_reward)
```

Default Policy

The first step is to implement the default policy, which plays through an entire game session. It chooses actions at random, applying them to the board until the game is over. It then returns the reward vector of the finished board.

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

Browne, et al.

Note: You can use `random.choice(my_list)` to select a random item from `my_list`

In [156]:
```
######################################################################
# randomly picking moves to reach the end game
# Input: BOARD, the board that we want to start randomly picking moves
# Output: the reward vector when the game terminates
######################################################################
def default_policy(board):
    #while s is not terminal
#     state_s_reward = None
    while not board.is_terminal():
        action_set = board.get_legal_actions()
        #chooose a from A(s) uniformly at random
        my_list=list(action_set)
        action = random.choice(my_list)
        #s<-f(s,a)
        board = action.apply(board)
        state_s_reward=board.reward_vector()
    #return reward of state s
    return state_s_reward if state_s_reward is not None else 0
#     return state_s_reward
```

In [157]:
```
test_default_policy(default_policy)
```

```
test passed
test passed
test passed
```

Tests passed!!

Best Child

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

Browne, et al.

Note: For convenience, we've implemented a function that returns the heuristic inside the max operator. Look at the function `node.value(c)` for the `NODE` class API and save yourself the headache.

In [182]:
```
##############################################################
# get the best child from this node (using heuristic)
# Input: NODE, the node we want to find the best child of
#        C,    the exploitation constant
# Output: the child node
##############################################################
def best_child(node, c):
    #v'from children of v
    all_children = node.get_children()
    max_data = -1000000
    best_child_node = None  # 初始化 best_child_node 变量
    for current_child in all_children:
        current_data= current_child.value(c)#get the calculated UCT value
        if max_data <= current_data :
            max_data = current_data
            best_child_node = current_child
        else:
            continue
    return best_child_node
```

In [183]:
```
test_best_child(best_child)
```

```
test passed
```

Tests passed!!

Expand

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

Browne, et al.

In [46]:
```
############################################################
# expand a node since it is not fully expanded
# Input: NODE, a node that want to be expanded
# Output: the child node
############################################################
# def expand(node):
def expand(node):
    board = node.get_board()
    action_set = board.get_legal_actions()
    all_children = node.get_children()
    untried_actions = set()

    for action in action_set:
        if action not in all_children:
            untried_actions.add(action)

    # 选择一个未探索的动作
    action = random.choice(list(untried_actions))
    new_board = action.apply(board)
    new_node = Node(new_board, action, node)
    node.add_child(new_node)

    return new_node
```

In [47]:
```
test_expand(expand)
```

Tests passed!!

Tree Policy

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

Browne, et al.

In [50]:
```
############################################################
# heuristically search to the leaf level
# Input: NODE, a node that want to search down
#        C, the exploitation value
# Output: the leaf node that we expand till
############################################################

def tree_policy(node, c):
    i =0;
    while not node.get_board().is_terminal():
        i+=1
        if not node.is_fully_expanded():
            return expand(node)
        else:
            node = best_child(node, c)
        if(i>6):
            return node
    return node
```

In [51]:
```
test_tree_policy(tree_policy, expand, best_child)
```

```
test passed
```

Tests passed!!

Backup

Now its time to make a way to turn the reward from `default_policy` into the information that `tree_policy` needs. `backup` should take the terminal state and reward from `default_policy` and proceed up the tree, updating the nodes on its path based on the reward.

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

Browne, et al.

In [200]:
```
############################################################
# reward update for the tree after one simulation
# Input: NODE, the node that we want to backup from
#        REWARD_VECTOR, the reward vector of this exploration
# Output: nothing
############################################################

def backup(node, reward_vector):

        # Update the Q-value of the node based on the reward vector
        player_id = node.get_player_id()
        print(player_id)
        delta = reward_vector[player_id]
        print("delta",delta)
#         print("node.q1",node.q)
        node.q -=delta
        print("node.q2",node.q)
        if(node.get_parent()is not None):
#             node.q =node.q_value()+delta  # Update the Q-value directly
            node = node.get_parent()  # Move up to the parent node
        else:
            return
```

In [201]:
```
test_backup(backup)
```

```
(1, -1)
1
delta -1
node.q2 1.0
0
delta 1
node.q2 -1.0
1
```