Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Bachelor Thesis

Intelligent Algorithms for Optimal Exploration and Exploitation in Dynamic environments

Petr Marek

Supervisor: MSc. Štěpán Kopřiva, MSc.

Study Programme: Open Informatics

Specialisation: Computer and Information Science

May 21, 2015

Poděkování

Zde bych rád poděkoval všem, kteří mě podporovali při psaní mé bakalářské práce.
Především děkuji MSc. Štěpánu Kopřivovi, MSc. za vedení, konzultace a cenné rady, které mi poskytl při psaní tohoto textu v angličtině.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 21.5.2015 Petr Marek

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	Petr M a r e k
Studijní program:	Otevřená informatika (bakalářský)
Obor:	Informatika a počítačové vědy
Název tématu:	Algoritmy pro optimální exploraci a exploitaci v dynamických prostředí

Pokyny pro vypracování:

1. Nastudujte problém optimální explorace a exploitace v dynamických prostředích.
2. Nastudujte koncept multi-armed bandit, s ním spojené algoritmy a použití pro počítání optimální explorace a exploatace.
3. Formulujte problém optimální explorace a exploitace pro jednoduchou online hru s volitelnými herními parametry.
4. Naprogramujte jednoduchou online hru s volitelnými herními parametry.
5. Navrhněte vhodný algoritmus pro problém explorace a exploitace v prostoru herních parametrů výše jmenované hry.
6. Naimplementujte algoritmus z bodu 5.
7. Otestujte algoritmus na online hře z bodu 4.

Seznam odborné literatury:

[1] Joannès Vermorel , Mehryar Mohri - Multi-armed bandit algorithms and empirical evaluation - 2005
[2] Peter Auer , Nicolò Cesa-Bianchi , Paul Fischer – Finite-time Analysis of the Multi-armed Bandit Problem- 2000

Vedoucí bakalářské práce: MSc. Štěpán Kopřiva, MSc.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
vedoucí katedry	**děkan**

V Praze dne 14. 1. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student:	Petr M a r e k
Study programme:	Open Informatics
Specialisation:	Computer and Information Science
Title of Bachelor Project:	Intelligent Algorithms for Optimal Exploration and Exploitation in Dynamic environments

Guidelines:

1. Study the problem of optimal exploration & exploitation in dynamic environments.
2. Study the multi - armed bandit problem and its relation to the optimal exploration & exploitation.
3. Formalize the problem of optimal exploration and exploitation for a simple online game.
4. Design and implement simple online game with variable game parameters.
5. Design an algorithm for exploration and exploitation in the space of game parameters of the game in 4.
6. Implement the algorithm designed in 5.
7. Evaluate the algorithm on the game in 4.

Bibliography/Sources:

[1] Joannès Vermorel , Mehryar Mohri - Multi-armed bandit algorithms and empirical evaluation - 2005

[2] Peter Auer , Nicolò Cesa-Bianchi , Paul Fischer – Finite-time Analysis of the Multi-armed Bandit Problem- 2000

Bachelor Project Supervisor: MSc. Štěpán Kopřiva, MSc.

Valid until: the end of the summer semester of academic year 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
Head of Department	**Dean**

Prague, January 14, 2015

Abstrakt

V předložené práci studujeme algoritmy, které se zabývají udržováním rovnováhy mezi exploration (průzkum prostředí ve snaze najít optimální strategii) a exploitation (využívání strategie, která přináší největší užitek). Dilema mezi exploration a exploitation je součástí teorie posilovaného učení (reinforcement learning) a jeden z jeho nejjednodušších příkladů je u tzv. multi-armed bandit problému, který je v práci využit pro modelování konkrétního prostředí jednoduché online hry. Hlavním cílem práce je navrhnout algoritmus, který bude optimálně vyvažovat exploration a exploitation a tak přinese největší užitek. V případě jednoduché online hry je algoritmus schopen najít nejoblíbenější variantu této hry. K tomu účelu je navržena hra Speeder s volitelnými herními parametry a algoritmus UCBCustom, který je uzpůsobený použití v daném prostředí a jeho efektivita analyzována na hře Speeder. Algoritmus umí za běhu dynamicky přepočítávat užitky parametrů. V práci také popisujeme, jak se místo herních parametrů dají použít ceny v případě freemium her. Skrze testování práce ukazuje, že navržený algoritmus UCBCustom je schopný identifikovat parametry jedné z nejoblíbenějších variant hry a to již po malém počtu odehraných her.
Klíčová slova: Multi-armed bandit, exploration versus exploitation, freemium

Abstract

In the thesis we study algorithms that balance exploration (searching the environment for optimal strategy) and exploitation (using the most beneficial strategy). The exploration vs. exploitation dilemma is the main aspect of various problems in theory of reinforcement learning and one of its simplest instances is the multi-armed bandit problem, which we use to model a specific environment of a simple online game. The main objective of the thesis is to develop an algorithm that will optimally balance exploration and exploitation and thus will result in the best possible profit. In the case of a simple online game, the algorithm is able to find the most enjoyable variant of the game. To that end, we have designed a game with variable game parameters called Speeder and a custom algorithm called UCBCustom, which is tailored for use in the given environment and its effectiveness is evaluated on the game Speeder. The algorithm is able to dynamically re-evaluate benefits of the parameters during runtime. The thesis also describes how we can use prices of features in a freemium game instead of game parameters. Further in the thesis, we show through testing that our designed algorithm UCBCustom is able to identify one of the more enjoyable variants of the game even after a limited number of plays.
Keywords: Multi-armed bandit, exploration versus exploitation, freemium

Contents

1 Introduction

The exploration versus exploitation dilemma is a vital part of reinforcement learning and describes situations where one searches for balance betweeen exploring the environment for profitable actions and trying to take the best action found so far as often as possible. The multi-armed bandit problem is the simplest instance of this particular situation and has been thoroughly studied, since it is associated with many practical use cases. Multi-armed bandit (MAB) problem describes a situation of a gambler in front of a row of slot-machines, that are called arms, where each machine returns a reward from a probability distribution unknown to the gambler whenever the machine is played. The practical use cases include clinical trials or resource allocation for research projects. The MAB problem can also be used in economics, where the aim is to maximize profit by correctly pricing products when the product's market demand is not known beforehand.

We are using the multi-armed bandit framework for its effectiveness in software testing and its potential use in freemium games. Freemium is a type of bussiness model, where the product is provided free of charge to everyone and revenue is generated by selling additional premium features to a small percentage of users - premium users. The intention of the company that developed the product is to convert as many of the users to premium users. Freemium games have lately become very popular, however their generated profit relies heavily on proper pricing of the premium features.

In our thesis, we have decided to design and implement a multi-armed bandit algorithm which determines the best combination of variable game parameters of a developed game by sequentially testing the game. In case of freemium games, each game parameter would correspond to either the price or a different attribute of a premium feature. However, we cannot evaluate our algorithm on the proposed case of freemium games as we would be risking a company's profit in the process. Instead, we have chosen to create a simple online game with variable game parameters and aim to find the most enjoyable variant of the game and its corresponding combination of parameters using our designed algorithm.

1.1 Goals of the thesis

1. **Study the problem of optimal exploration & exploitation in dynamic environments**
   The problem of optimal exploration & exploration in dynamic environments is explained and its importance in practical use cases discussed in the first part of chapter 3.

2. **Study the multi-armed bandit problem and its relation to the optimal exploration & exploitation**
   We study the multi-armed bandit problem and its relation to the op-

timal exploration and exploitation dilemma in chapters 2 and 3. First we create a summary of related work and briefly describe the history of the multi-armed bandit problem in chapter 2. In chapter 3 we formally define the multi-armed bandit problem and introduce some common algorithms that solve the multi-armed bandit problem.

3. **Formalize the problem of optimal exploration and exploitation for a simple online game**
   In chapter 3 we define the specifications of a simple online game for which the multi-armed bandit problem and its associated exploration and exploitation dilemma is then formalized in chapter 4.

4. **Design and implement simple online game with variable game parameters** Our simple online game Speeder including variable game parameters is designed in the last part of chapter 3
   and its implementation is presented in chapter 6.

5. **Design an algorithm for exploration and exploitation in the space of game parameters of the game in 4**
   We create an algorithm for the multi-armed bandit problem, which is inspired by some of the algorithms we described in chapter 3. The algorithm uses a type of upper confidence bound policy called UCB1 and is also able to work in cases where the amount of arms exceeds the total number of tests. The design of our algorithm is fully described in chapter 5.

6. **Implement the algorithm designed in 5**
   The implementation of our designed algorithm along with the corresponding framework, under which the game is run, can be found in chapter 6. Playing the game under this framework then allows our algorithm to eventually determine the best combination of game parameters.

7. **Evaluate the algorithm on the game in 4**
   We evaluate our algorithm by analyzing data collected from testing of the framework and the algorithm. In the testing phase we try out three different variants of our algorithm. The variants differ only in how much they are prone to exploring new arms. The testing phase and the evaluation is presented in chapter 7.

1.2 Structure of the thesis

In chapter 2, we present an overview of published work related to exploration and exploitation dilemma and the multi-armed bandit problem. The chapter consists of two sections. The first section of the chapter summarizes the history of multi-armed bandits and the evolution of the strategies which solve

MAB. The second part of the chapter focuses on practical use cases of the multi-armed bandit problem by summarizing several works, that describe the use of MAB in real world examples, such as clinical trials and so on.

In chapter 3, we first explain the exploration vs. exploitation trade-off and formally describe the multi-armed bandit problem. The next section of the chapter then outlines some of the most known and often used strategies that solve the MAB problem. Simple semi-uniform strategies and the popular upper confidence bound policies are the main focus of the second section and the problem of the infinitely-many armed bandit and its relation to our case is also introduced here. The third section of the chapter describes the use of multi-armed bandit as a testing method. It explains how the MAB framework can be mapped to the testing environment and compares the MAB approach to the popular A/B testing method. The last section of the chapter presents our simple game Speeder and the process of how we designed it.

Chapter 4 focuses on the formalization of the multi-armed bandit problem for the specific testing environment of our game. It is divided into three sections, where the first section describes the chosen game parameters that are to be tested by our MAB framework. The second section contains the domains of the game parameters and the reason why they have to be properly discretized. The last section deals with the problem of normalizing the rewards for our algorithm as it works only with bounded rewards and our time-based rewards are not limited.

In chapter 5, we design our own algorithm UCBCustom that is based on the popular UCB1 algorithm. Our algorithm also has an addition that comes as an inspiration from the UCB-AIR algorithm, which solves the infinitely-many armed bandit problem. That allows us to use our algorithm in our limited testing case, where the total number of tests is extremely small compared to the amount of arms.

Chapter 6 focuses on the practical application of both our game Speeder and the multi-armed bandit framework that contains our designed algorithm from chapter 5. The chapter is divided into two main sections, with the first containing details of implementation of the game Speeder and the second describing the aspects of the MAB framework implementation.

Chapter 7 contains essential information about the evaluation process of our designed algorithm. The first section of the chapter describes the environment in which we have tested our algorithm and points out its numerous differences from the theoretical environment of the multi-armed bandit and how it might affect our resulting data. The second section provides a summary of collected data and graphs illustrating the effectiveness of our algorithm. The last section of the chapter contains a discussion of the results of our evaluation process and compares them to our initial predictions.

2 State of the art

2.1 History of Multi-armed Bandits

The multi-armed bandit framework and its exploration vs. exploitation dilemma was originally described by Robbins in [1]. His work included several problems in sequential design, which is a type of statistical analysis developed for more efficient industrial quality control during World War II. Sequential design differs from classical analysis by not having a fixed sample size and evaluating data as it is collected. Optimal solutions for these problems were not known at the time and later were found to be hard to achieve. Robbins proposed some reasonably good ways to solve these problems, but mainly raised the general question of this problem - that is how to properly learn from past experiences in instances of these problems. Since then, many asymptotically optimal solutions were found and described in literature.

Gittins [2] provided an algorithm for computing the optimal strategy in certain bandit problems. The method however assumes the rewards are geometrically time-discounted, which is usually not the case in practical instances of the multi-armed bandit framework. The algorithm lies in providing a way to compute a Gittins index for each arm seperately without relying on information about the other arms and choosing an arm with the highest Gittins index. Even in the special case of geometrically discounted rewards, the Gittins index method is unfortunately accompanied by computational and other difficulties, since the standard Gittins indices are hard to compute exactly. There is a way to approximate the indices using a normal approximation, but this method can exhibit so-called incomplete learning, which means there is a probability that a suboptimal arm will be played forever. The notion of assigning indices to the arms and choosing the arm with the highest/lowest index was later on used in many other algorithms - all of them denoted as index policies.

The main measure of a strategy's success is called regret and is based on the fact that the optimal arm is not selected every time. Lai and Robbins [3] were one of the first to prove that regret has to grow at least logaritmically for the multi-armed bandit problem. They also developed a whole category of policies that asymptotically achieve this regret. The strategies are all based on computing an upper confidence bound (UCB) for each machine. The computation of the index as proposed originally by Lai and Robbins can be quite difficult, but many later studies, i.e. [4] by Auer et. al, enhanced the algorithms to be more effective in this sense. These modern variations of UCB algorithms are quite popular, since they are asymptotically optimal and simple to implement.

A different, Bayesian perspective on the MAB framework is shown in the work of Steven L. Scott [5]. It describes a method which plays arms randomly based on the Bayesian posterior probability of each arm that it is optimal. This heuristic algorithm is called randomized probability matching and in simpler conditions can outperform certain optimal methods. The basics of this strategy lie in playing the arms in proportion to their corresponding probabilites of being optimal, there is however no guarantee of this method being asymptotically optimal.

2.2 Practical use cases of MAB

Multi-armed bandit framework has been rising in popularity in the past decades and many studies concentrating on applying the theory to real-world instances have been published.

In [6], Hardwick and Stout analyzed the use of multi-armed bandit framework in its most associated instance - clinical trials. The analyzed problem consists of a clinical trial, in which we need to optimally allocate the patients to two treatments in order to decide which treatment is better. However, there is an underlying ethical dilemma that results in the need of minimal patient losses. The article compares several allocation strategies and discovers that the bandit policies perform really well even with the ethical condition of keeping the failures to minimum.

The effectiveness of MAB framework in economics and finance is assessed and summarized in [7]. In economics, bandit strategies are for example used to optimally sequentially allocate prices to products based on an unknown market demand, which is learnt and unveiled through this process. It is shown that with higher intensity experimentation market demand is tracked almost perfectly as opposed to lower intensity experimentation.

Galichet et al. investigated in [8] the use of multi-armed bandit on application domains, where the exploration phase contains hazards and risks - i.e. energy management or robotics. The paper presents the Multi-Armed Risk-Aware Bandit (MARAB) algorithm, which takes the arm's probability of risk into account. It then analyzes the MIN algorithm, to which MARAB tends under certain conditions, and compares both the algorithms with the popular UCB on several experiments.

3 Technical background

3.1 Multi-armed bandit framework

In this chapter, we will write about the multi-armed bandit framework (MAB), particularly what it is and how it works. But to fully understand the nature of the multi-armed bandit problem, one must first be aware of the exploration vs exploitation tradeoff.

There are many theoretical and practical problems, in which decisions are made to maximize a certain numerical reward. In some of these problems, the possible actions that can be taken include not only actions which directly lead to maximizing the expected reward, but also actions that gather more information about the situation and may ultimately lead to an ever higher reward. The main difficulty when solving these problems lies in the balance of those two sets of actions. If one chooses to only gain more knowledge exploring in an environment where the best strategy is already known, they lose a potential amount of reward they might have gained instead. If however, one chooses to only exploit the best strategy based on their knowledge without exploring new options, they might be missing out on an opportunity to gain an even higher reward. This dilemma is called the exploration vs. exploitation trade-off and is a big part of reinforcement learning.

One of the typical exploration vs. exploitation problem instances are clinical trials [9]. It is a situation, where one wants to determine the best treatment by experimenting and trying new treatments, but at the same time needs to minimize patient losses. Similar example can be found in the case of allocating resources to competing projects, where an organization has a fixed budget and aims to fund only the most successful project. However at the beginning, little is known about all of the projects' eventual rewards. The organization learns the payoffs of each project over time and repeatedly makes decisions which project to move their attention and resources to at any given time.

The multi-armed bandit problem, also called the K-armed bandit problem [4], is a brilliant example of this kind of a problem. It describes a situation of a gambler in front of a row of K slot-machines, which are called one-armed bandits, that all provide some reward based on a probability distribution that is tied to that certain slot-machine. The gambler has no prior knowledge of the rewards given by the machines, and it is up to the gambler to try and maximize their sum of rewards in a carefully selected sequence of lever pulls.

Formally, as per defintion by Auer et al. [4], a K-armed bandit problem is defined by a set of random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$. Each i denotes an index of an arm (machine) of a bandit and n denotes the n-th play of the machine. Therefore, $X_{i,1}, X_{i,2}, X_{i,3}, \ldots$ represent rewards won from subsequent plays of an arm with index i. These are identically distributed based on an unknown distribution with an unknown expected value μ_i. Let $T_i(n)$ be the number of times an algorithm chose to play machine i during a total of n plays. Cumulative regret is then defined as

$$\mu^* n - \sum_{j=1}^{K} \mu_j [T_j(n)] \tag{1}$$

where $\mu^* = \max\limits_{1 \le i \le K} \mu_i$.
Since $T_j(n)$ might be stochastic, it is often more useful to work with a different version of regret - expected regret [10]. In a stochastic environment it is more logical to assess an algorithm against the optimal play from the perspective of expectation rather than against the optimal strategy on the already received rewards. Expected regret (also called pseudo-regret) is defined as

$$\mu^* n - \sum_{j=1}^{K} \mu_j \mathbb{E}[T_j(n)] \tag{2}$$

where $\mu^* = \max\limits_{1 \le i \le K} \mu_i$ and $\mathbb{E}[T_j(n)]$ denotes the expected value of $T_j(n)$.

Algorithms which solve the K-armed bandit problem aim to minimize regret. Regret can be illustrated as the expected loss, since no algorithm will always play the best machine. A zero-regret strategy is an algorithm, for which the average regret per play tends to zero whilst the number of plays tends to infinity.

There is a version of the multi-armed bandit problem, where the gambler knows all the possible rewards of the machines, which has a simple solution that lacks the problem of balancing exploration and exploitation. This version will not get much attention in this thesis, as the first one is significantly more useful in real-life situations. We will try using the theoretical algorithms that solve the multi-armed bandit on our chosen practical problem, which is to derive the most profitable and enjoyable settings of a computer game during its testing phase by providing each tester a game with different settings.

3.2 Bandit algorithms

In this part we will describe the algorithms that are widely used to solve the multi-armed bandit problem.

3.2.1 ϵ-greedy and other semi-uniform strategies

ϵ-greedy strategy [11] is one of the most commonly used methods to solve bandit problems. Although being a very simple and straightforward strategy, it is interesting to point out that it's often very hard to beat. The main principle of the strategy consists of choosing a random machine to play with ϵ-frequency and otherwise playing a machine with the highest mean. The mean is computed and estimated using the rewards from previous rounds. ϵ is a parameter of the algorithm and is left up to the developer. A higher value of ϵ leads to a more explorative strategy, while lower values will result in less exploration-based policies.

There are many variants of the algorithm that vary on the use of the ϵ parameter. The ϵ-first strategy is based on doing all the exploration at the beginning of the algorithm. If we decide to play a total number of N plays, the first ϵN rounds will consist of pure exploration, where the strategy will pick a machine to play at random. The algorithm will then only exploit the gained information and pick the machine with the highest estimated mean in the later $(1-\epsilon)N$ rounds.

The main setback of the basic ϵ-greedy strategy is the use of a constant ϵ. Even after the algorithm might have already found the optimal machine to play and converged, a constant $(1-\epsilon)$ portion of rounds to be played will still be about exploration. The ϵ-decreasing strategy tries to solve this issue by introducing a decreasing ϵ over the course of played rounds in order to asymptotically get closer to the optimal strategy. If the ϵ function is well chosen, this strategy can be considered a zero-regret strategy.

3.2.2 Upper Confidence Bound (UCB) algorithms

The use of upper confidence values for solving the multi armed bandit problem was first introduced in the work of Lai and Robbins [3]. However, their algorithm had the downside of needing the entire history of the previous choices and rewards, which consumes a lot of memory. An often used UCB1 algorithm was introduced by Auer et al. in [4]. It uses the ideas of Lai and Robbins and conserves a lot of memory.

The UCB1 algorithm starts by playing each machine once. Afterwards, during each round, the policy plays a machine i which maximizes

$$\bar{X}_i + \sqrt{\frac{2b \ln T}{T_i}} \tag{3}$$

where $\bar{X}_i$ is the average reward of machine i, T_i is the number of times machine i was played and T denotes the total number of rounds played. This form of the confidence bound assumes the rewards come from a bounded support in $[0, b]$.

The regret for the multi-armed bandit problem is proven to grow at least logaritmically in [3] and this algorithm achieves this logaritmic regret asymptotically. Practically speaking, it means the optimal machine is played exponentially more often than a different, sub-optimal machine. As can be seen in [4], the expected regret of the algorithm UCB1 used on a K-armed bandit problem is after n plays at most

$$[8 \sum_{i:\mu_i<\mu^*} \frac{\ln n}{\mu^* - \mu_i}] + (1 + \frac{\pi^2}{3})(\sum_{j=1}^{K}(\mu^* - \mu_i)) \tag{4}$$

The UCB family of algorithms also contains an important variant of the algorithm called UCB-V. This algorithm was introduced in [12] by Audibert et al. and uses a confidence bound, which differs from UCB1's bound by also weighing in an empirical estimate of the variance of the arm's rewards into the computations. The method is superior to UCB1 especially in cases, where the rewards of the suboptimal arms have low variance. The upper confidence bound index for the UCB-V algorithm is defined for each arm i in the n-th total play as follows:

$$\bar{X}_i + \sqrt{\frac{2V_i\varepsilon_{T_i,n}}{T_i} + c\frac{3\varepsilon_{T_i,n}}{T_i}} \tag{5}$$

where $\bar{X}_i$ is the average reward of machine i and T_i is the number of times machine i. V_i is the empirical estimate of variance for arm i and ε is called an exploration function or exploration sequence, and its typically chosen value is $\varepsilon_{s,t} = \varsigma ln(t)$, so despite being a function of (s, t), it usually only depends on t. $\varsigma, c > 0$ then become additional parameters that can be used to specifically tune the algorithm's behavior and the balance between exploration and exploitation. As in UCB1, this form of the upper confidence bound works with rewards that come from the interval of $[0, b]$. The empirical estimate of variance for arm i after it has been chosen t times is computed using

$$V_i = \frac{1}{t}\sum_{j=1}^{t}(X_{i,j} - \bar{X}_i)^2 \tag{6}$$

where $\bar{X}_i$, as in previous equations, is the average reward of the machine.

UCB-V initializes the same way as other UCB algorithms - by choosing each arm once. Then each round consists of first computing the *upper confidence bound* indices for all the arms of the problem and choosing the arm with the highest index afterwards.

3.2.3 Algorithms for MAB with an infinite number of arms

There exists a variant of the multi-armed bandit problem, where the amount of the arms is not finite and given by an integer, but is instead unlimited. Formally, the infinitely-many armed bandit problem is defined by a set of random variables $X_{i,n}$ for $i \geq 1$ and $n \geq 1$. Each i denotes the index of an arm and n denotes the n-th play of that particular arm. The main difference from the definition of the K-armed bandit problem is the absence of the upper bound on i, which allows us to consider an infinite amount of arms.

The problem is similar to a case where the number of arms is too high for the planned amount of plays. In both described situations it is impossible to explore all the arms and policies starting by trying every arm once are therefore unusable. There are several algorithms that solve the MAB problem in these circumstances, however their optimality usually relies on assumptions made on the mean-reward distributions.

Wang et al. [13] introduced two main algorithms that solve the infinitely many armed bandit problem and both of these policies are based on the Upper Confidence Bound algorithms. The policies demand certain knowledge about the mean-reward distribution to be known beforehand and use this knowledge in from of a parameter $\beta > 0$. Formally, as stated in [13], the probability that a new arm i is ϵ -optimal is of order ϵ^b for infinitely small ϵ, that means $P(\mu_i \geq \mu^* - \epsilon) = \Theta(\epsilon^b)$ for $epsilon \to 0$. μ^* denotes the mean-reward of the optimal arm, and a value of $\beta = 1$ corresponds to the usual uniform distribution.

The first of the two policies is called the UCB-F (fixed horizon) and is the go-to strategy for cases, where the number of total plays is known beforehand. Given a total number of plays n and the parameter β, the algorithm first chooses K arms at random. The value of K depends on the parameter $\beta > 0$ and the mean-reward of the optimal arm $\mu^* \in [0, 1]$ - if $\mu^* = 1$ or $\beta \geq 1$, $K = n^{\frac{\beta}{\beta+1}}$, otherwise the value of $n^{\frac{\beta}{2}}$ is assigned to the variable K. Then, as specified in [13], a UCB-V strategy is run on the K randomly chosen arms with the exploration sequence satisfying $2\ln(10\ln t) \leq \varepsilon_t \leq \ln t$.

Algorithm 1 UCB-F Algorithm

```
1: Given: set of all arms $ARMS$, empty set of arms $A$, $\beta > 0$, number of
   total plays $n$
2: if $\beta \geq 1$ then
3:     $K \leftarrow n^{\frac{\beta}{\beta+1}}$
4: else
5:     $K \leftarrow n^{\frac{\beta}{2}}$
6: end if
7: $A \leftarrow$ random $K$ arms from the set $ARMS$
8: run the policy UCB-V on the set of arms $A$
```

The second of the policies is called the UCB-AIR (arm-increasing rule) and is used in the case, where UCB-F is unusable - when the total number of plays is unknown before the strategy is applied. It starts with an empty set of arms A. In the n-th total round, the following rules apply. If $\beta \geq 1$ or $\mu^* = 1$, a new arm is added to the set A only if the number of arms already included is lower than $n^{\frac{\beta}{\beta+1}}$. Otherwise, if $\beta < 1$ and $\mu^* < 1$, a new arm is added to A if the amount of arms already included is lower than $n^{\frac{\beta}{2}}$. If the number of arms in set A is greater than the appropriate bound, no new arm is included in A that turn. On the set of arms A, a common UCB-V algorithm is applied, with the exploration function satisfying $2 \ln(10 \ln t) \leq \varepsilon_t \leq \ln t$, as specified in [13]. Obviously, whenever a new arm is added to the set, it is immediately chosen afterwards.

Algorithm 2 UCB-AIR Algorithm

```
1: Given: set of all arms $ARMS$, empty set of arms $A$, $\beta > 0$, $N = 0$
2: loop
3:     if $\beta \geq 1$ then
4:         $K \leftarrow (N+1)^{\frac{\beta}{\beta+1}}$
5:     else
6:         $K \leftarrow (N+1)^{\frac{\beta}{2}}$
7:     end if
8:     if size(A) $< K$ and $A \neq ARMS$ then
9:         add a random arm $i \notin A$ from $ARMS$ into the set $A$
10:    end if
11:    apply the policy UCB-V on the next round with the set of arms $A$
12:    N++
13: end loop
```

3.3 Testing using Multi-Armed Bandit

Whenever a product is being developed, it often has more variants. Then it is in the developer's best interests to thoroughly test which variant would be more successful, popular and profitable. There are many frameworks that facilitate this for the developer and most commonly they use a technique called A/B Testing. The main principle lies in showing the different variants to different people and measuring the usability and usefulness of the variant. In practice it can be used when creating web sites, i.e. showing two variants of the same page, the only difference being in the size of a button. One variant has a bigger button and the other has a smaller one. The framework keeps tracking the number of people that clicked on the button when they were shown variant A and the same thing for the second variant. Then it computes whether the results are relevanth using statistical significance. After the test has ended, the framework chooses the better variant for the developer. A/B Testing is usually limited by having only two variants, one being the main control variant, and the second - treatment - being the amended one. There are several modifications of A/B Testing that involve more variants, but those are not used very often.

Multi-armed bandit approach can be similarly used as a testing method instead of A/B Testing. It is often debated to be the better method and for example Google uses MAB algorithms in their testing service Google Website Optimizer. Essentially showing a variant i of the software or website to the user corresponds to playing a machine i in the multi-armed bandit problem. The reward then depends on the actions of the user, i.e. whether he completed the provided task, how long it took him to complete the task and so on. Unlike A/B Testing, MAB testing is not limited by the number of the variants and one can get the exact size of the button in pixels simply because MAB solves the exploration vs exploitaton problem.

The main goal of this thesis was to find a proper algorithm for MAB testing on freemium games not unlike Candy Crush, Farmville and so on. Freemium is a business model combining the Free and Premium business models. The game itself is free to play, but it has microtransactions in the game and the player may opt to buy additional content or pass and enjoy the game for free. MAB testing would be perfect for determining the right prices for the microtransactions, since every bandit algorithm aims to minimize regret. And in this instance regret corresponds to loss of profit against the best possible profit of the game as if the prices were optimal from the beginning.

This is however very difficult to test, since the programmer is basically risking the game's profit. So in this thesis we will test the bandit algorithms on a different aspect of developing the game. Our situation is very similar to determining the right prices and attributes of the microtransactions in the fremium model. The prices and attributes of the microtransactions can be viewed as parameters of the game, which are to be determined by our MAB testing. We will however choose several different parameters of our game to test using our bandit algorithms. For example we will try to determine how many lanes our game should have, how fast the car should go at the beginning and so on. The rewards from the testing won't have the form of profit, but will result in higher satisfaction of the players, which we will measure by tracking how long has the user played the game. The principle of the testing is the same, but it avoids the danger of potentially losing a potential company's money and profit from their game.

3.4 Developing the game

To test the theory in practice, we had to develop the game - the platform on which to test the algorithms. The game had to be simple, catchy, easily accessible and developed for use under our own framework, which would implement the best algorithm for multi-armed bandit testing and deliver the parameters to the game. Firstly, I will describe the idea of the game and then the technologies and libraries used to develop the game itself.

3.4.1 Game of Speeder

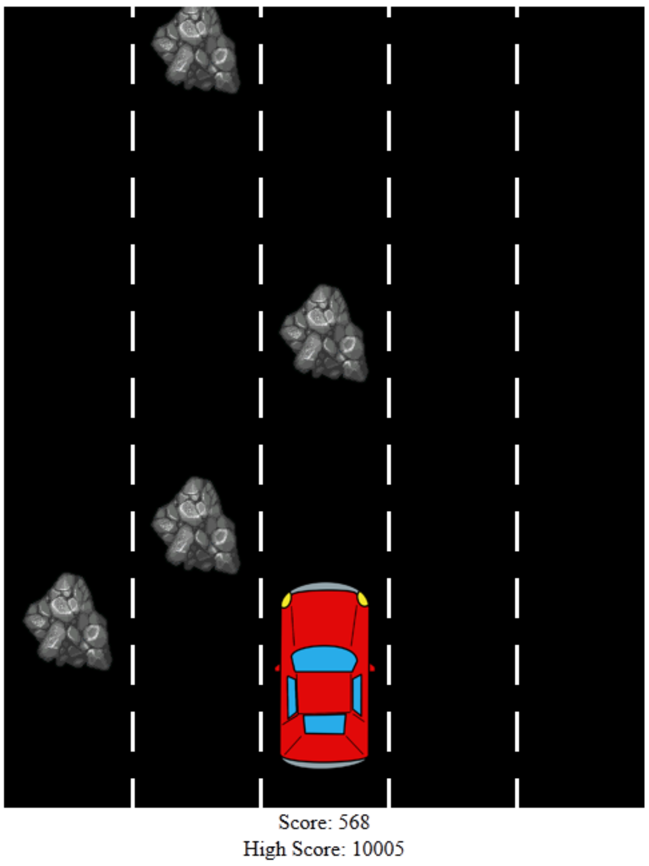


Figure 1: Game of Speeder

In recent years, the catchiest games are based usually on a very simple idea. One brilliant example can be Flappy Bird - even though it is very easy to implement, the game and idea itself caught on and millions of people played it in their free time. So one of the requirements of the game was for it to not be too difficult to implement, since many successful games took many years and teams full of developers to create, but these are resources we didn't have at our disposal. We tried to find inspiration in past games, especially from the early history of the gaming era with games like Pong and so on.

The game revolves around a speeding car, which tries to avoid as many obstacles as possible. The car has no option to slow down and the only way to avoid obstacles is to switch lanes. The player switches lanes using the keys A and D on the keyboard. The game screen is fixated on the car itself and the car speeds up every time he misses an obstacle. The goal of the game is to survive as long as possible and the difficulty lies in the need of quicker and quicker reactions from the player. When the car runs into an obstacle, the game restarts to the beginning speed and the score changes back to zero.

4 Problem formalisation

Formally, the problem of choosing the best combination of game parameters during its testing phase can be seen as a K-armed bandit problem [4], where each arm corresponds to a certain variant of the game. A variant of the game is defined by its combination of game parameters. Showing a variant to a testing player can be seen as playing the corresponding arm in MAB, and the reward is based on the satisfaction of the user - in our case this is represented solely by the amount of time spent playing. The MAB framework can then solve this problem while minimizing the regret, which is essentially the dissatisfaction of users. In this chapter we will talk about the chosen game parameters, their discretization and other formalisation problems.

4.1 Chosen game parameters

For the MAB framework to have the biggest impact, the observed and tested game parameters needed to have a large influence on the player's satisfaction. We have chosen the game parameters as follows:

Number of lanes
One of the most significant parameters of our game is the total number of lanes. A change of this parameter is immediately perceived by the player, since it has such a visual impact on the game.

Initial speed
The optimization of the initial speed of the car is very important for the game playability. If it is too low, the game might become too dull and slow-paced, since the player starts over every time they crash into an obstacle and their speed reverts back to the initial speed. However, if it's higher than it should be, the game becomes too difficult or even impossible to play.

Speed-up coefficient
The car speeds up every time it successfully avoids an obstacle. The game is designed in a way so that the speed grows exponentially - $s_i = qs_{i-1}$, where s denotes the speed. We describe the value q as the speed-up coefficient. For example, if the speed was to be increased by 5% every time the car avoids an obstacle, the value of q would be set to 1.05.

Speed cap
Since the speed of the car is designed to grow exponentially, there has to be a value of speed, at which the exponential growth stops. If the growth didn't stop, the game would most likely become unplayable.

However, we leave this to be determined by the MAB framework. We allow a value 0 for this parameter, which means the growth won't stop.

Frequency of obstacles
This parameter denotes the time between generating new obstacles in seconds, that is if a suitable lane is found for the block.

Minimal space
We didn't want the game to be impossible, so the generating algorithm needed to guarantee a possible path for the car to take. That is ensured by the parameter of minimal space, which describes the minimal distance between two obstacles that are located in adjacent lanes.

4.2 Discretization of parameters

We have described all the game parameters, now we need to specify their domains. There are upper and lower limits associated with each of the parameters, which are described in the table below. The limits alone however aren't enough to make the values viable for the bandit algorithms. The MAB framework works with a set of arms, which in our case corresponds to the cartesian product of the domains of each parameter. If we chose n possible values for each game parameter, the amount of all arms would be equal to n^6, since we have 6 parameters. Therefore, the domains for each game parameter have to be carefully discretized and picked, especially for our limited testing.

Parameter	**Lower bound**	**Upper bound**
Number of lanes	≥ 2	≤ 5
Initial speed	> 0	$\leq$ speed cap
Speed-up coefficient	> 1	≤ 2
Speed cap	$\geq$ inital speed	≤ 50
Frequency of obstacles	≥ 0	≤ 2
Minimal space	$\geq$ car length	$\leq$ game screen height

Number of lanes needs to have an integer value of at least 2. Having just one lane defeats the purpose of our game, since it is mainly about avoiding obstacles by switching lanes quickly. It could be set to any integer values higher than 2, but based on our observation, having more than 5 lanes doesn't seem to have a significant effect on the gameplay.

Initial speed has to be higher than 0 for the car to even start moving and it shouldn't exceed the value of the speed cap, since that is the maximum speed the car can travel at.

Speed-up coefficient should be a real number higher than 1 for the car to keep speeding up and not slowing down. However, a value higher than 2 seemed redundant based on our experience during manual experiments -

the value corresponds to a 100% increase of speed every time the car avoids an obstacle, which usually leads to reaching the speed cap after only 2 or 3 obstacles.

Speed cap has to be greater or equal to the initial speed and is technically limited only by the screen height of the game, but a higher value than 50 is way too fast to react to.

Frequency of obstacles needs to be higher or equal to 0, otherwise the obstacles wouldn't spawn at all. We have also limited frequency to be lower or equal to 2, since more than 2 seconds between potential obstacles seemed too long of a time.

Minimal space should have a value higher than the length of our car, for the obstacle generating algorithm to guarantee a path for the player.

4.3 Normalisation of rewards

As we have earlier described, time spent playing our game is considered as the reward in our multi-armed bandit framework. However, some of the algorithms depend on the reward being between the values 0 and 1, so the value that is returned to the framework has to be normalised before it is stored in memory.

Every time a new reward is recorded, it is normalised as a portion of the highest reward. If the new reward is higher than the maximum value, it is normalised to a value of 1 and all the rewards stored in memory have to be normalised as a portion of the new maximum. If this normalisation procedure is called every time a new time is received, every reward recorded will be greater or equal to 0 and less or equal to 1. Then the MAB framework can decide which arm to play based on these normalised rewards.

Bandit strategies don't necessarily store all the recorded rewards and many of them use the rewards' mean values instead. In case of recorded mean values the procedure remains the same and it can be shown the result is equal to normalising each reward by itself and then computing their mean value anew.

Algorithm 3 Reward normalisation

```
1: Given: $maxReward$
2: procedure NORMALISE($a$)                      ▷ Normalisation of reward $a$
3:     if $a > maxReward$ then
4:         for all recorded rewards $r$ do
5:             $r \leftarrow r * \frac{maxReward}{a}$
6:         end for
7:         $maxReward \leftarrow a$
8:     end if
9:     return $\frac{a}{maxReward}$               ▷ Returns the new normalised reward
10: end procedure
```

5 Algorithm

In this chapter we describe the algorithm we have designed and implemented. Our algorithm is based on the Upper Confidence Bound algorithm UCB1 introduced by Lai and Robbins in [3] with some inspiration from the algorithm UCB-AIR, that is used to solve infinitely many-armed bandit problems. The main part of our algorithm is a further simplified and efficient variant of the original UCB1 algorithm. The policy is based on the following idea. Instead of choosing a machine that was the best in the previous rounds (highest mean of rewards) and since we cannot know which machine is truly optimal for sure, the algorithm plays the machine which is most likely to be the best one. A machine is likely to be optimal in two cases:

- The machine's mean reward is higher than the others and it has been measured very well, so it can be said with a high degree of statistical significance.
- The machine's mean reward is not the highest, but the confidence interval is large - it has not been played very often, so the mean reward is likely to be inaccurate and possibly higher.

The algorithm is based on the Chernoff-Hoeffding bound and inequality, which requires the random variables we work with - in our case the rewards from playing the machine - to be bound by an interval [a,b], where a, b are real numbers. Our index policy assumes the rewards are bound by the interval [0,1], so we need to normalise our time-based rewards before working with them, since they originally have no upper bound. The normalisation process is described in section 4.3 and is referenced in the code in this section.

As described in the work of Auer et Al. [4] and Audibert et Al. [14], the algorithm UCB1 initializes by playing each machine once and in following rounds always plays the machine maximizing value

$$\bar{X}_i + \sqrt{\frac{\rho \ln T}{T_i}} \tag{7}$$

where $\bar{X}_i$ is the average reward of machine i, T_i is the number of times machine i was played, T denotes the total number of rounds played and $\rho > 1$ is the exploration coefficient - the algorithm is much more prone to exploration the higher the coefficient is. As in [4], we have chosen the parameter ρ to be equal to 2, which is the most commonly used value for this parameter and doesn't result in a too exploration-heavy policy, which could result in a potentially higher regret.

We cannot however initialize our algorithm by playing each machine once, since each arm corresponds to a combination of our game parameters. Even when proper bounds are created for the parameters and the bounded intervals are discretized, the total amount of arms rises exponentially in comparison to the number of possible values for each arm. Similar situation can be seen in the infinitely many-armed bandit problem described in [13] by Wang et al., where one also cannot explore all the arms, since there is an unlimited amount of them. However, there are some requirements for their introduced algorithms UCB-F and UCB-AIR that we cannot fulfill. Those requirements include setting a value for their parameter β, that characterizes the mean-reward distribution. In our case we have no knowledge about the distribution of our mean-rewards and therefore cannot know the proper value for the parameter β with certainty.

Our addition to the simple UCB1 algorithm comes as inspiration from the UCB-AIR (arm-increasing rule) policy. Our algorithm uses the UCB1 algorithm on a set of arms A, that is initialized as an empty set and expands in the n-th round only if the condition $|A| <$BOUNDFUNCTION(n) is fulfilled. Furthermore, the UCB-AIR algorithm uses the UCB-V policy, while our custom UCB algorithm utilizes the simpler UCB1 policy.

The BOUNDFUNCTION(n) function allows for further experimentation with the explorative properties of the algorithm. In our testing phase, we want to test three different variants of the algorithm. All of the variants are associated with a certain value of β and for all of our tested variants the BOUNDFUNCTION(n) returns a value of $\sqrt{\frac{\beta}{\beta+1}}$. The default variant UCBCustom1's β value is set to 1, which corresponds to a uniform reward-mean distribution and is the commonly used value. The two other variants UCBCustom2 and UCBCustom3 correspond to $\beta = 2$, $\beta = 3$ respectively. Both of the additional variants are more explorative variations of our algorithm, in the sense that the set of arms A expands more quickly than the versions with lower β values.

Algorithm 4 UCB-CUSTOM Algorithm

```
 1: Given: set of all arms $K$, set of included arms $A = \emptyset$, N=0
 2: loop
 3:     if size(A) < BOUNDFUNCTION(N+1) textbfand $A \neq K$ then
 4:         add a random arm $i \notin A$ from $K$ into the set $A$
 5:         best ← $i$
 6:         skip to line 12
 7:     end if
 8:     for each machine i in $A$ do
 9:         UCBIndex[i] ← mean[i] + $\sqrt{\frac{2*\ln(N)}{numOfPlays[i]}}$
10:     end for
11:     best ← argmax(UCBIndex)
12:     reward ← NORMALISE(playMachine(best))
13:     mean[best] ← $\frac{mean[best]*numOfPlays[best]+reward}{numOfPlays[best]+1}$        ▷ Update mean
14:     numOfPlays[best]++                         ▷ Update number of plays
15:     N++                                  ▷ Update total rounds played
16: end loop
```

6 Implementation

In this chapter we describe how we have implemented both the simple online game of Speeder and our Multi-Armed Bandit framework, which is able to determine the best possible combination of parameters for our game thanks to our developed algorithm. We also consider how our framework could be re-used for other games, websites or similar online projects, where the best variant is needed to be determined through testing.

6.1 Implementation of Speeder

The game of Speeder, designed specially for testing and tweaking our algorithm, is a simple game implemented to be playable in a web browser. After the idea for the game was created, we started to look for a suitable game engine to implement it with. One of the options was Unity, a well-known cross-platform game engine used mainly by both beginning and seasoned indie developers. It's main drawback is the need of either downloading the game itself in an executable file or installing a special web browser plugin called Unity Web Player. Although being quite easy to use, Unity seemed far too advanced for developing a very simple game like ours.

In the end, our final choice went to a sprite game engine called Pjs 2D Game Engine which is written in an object-oriented programming language called Processing.js. Processing.js is a JavaScript port of the original Processing language and is used to display animations or interactive content on web sites without the need of either a Java Applet or a web browser plugin like aforementioned Unity Web Player or Flash Player. The Processing language is based on Java, but simplifies its syntax and makes it very easy to work with graphical content. It's often used for introducing programming mechanics to creative people with no programming experience. Processing.js converts code written in this language into pure JavaScript, which then uses the canvas element on the web site to render the interactive content. Since the final product is rendered by pure JavaScript, it's very easy to facilitate communication between the site and Processing.js content. This is very useful in our case, since it helps us deliver the game parameters from our algorithm into the game. That is done by adding a script element to the source code of the web site containing precisely these parameters. Then you can just work with them in the Processing language as if they were defined variables.

The Pjs 2D Game Engine was created by a Processing.js developer called Pomax and aims to make it even easier for people to use Processing.js for developing games. It uses sprites for the graphic visualization of the player, characters and other interactive objects in the game. It implements many features and game mechanics including not only ways to control the sprites, but also methods facilitating keyboard and mouse input, trigger areas, camera control and so on. However, several limitations of the rather simple engine became apparent during our development and needed to be worked around during our implementation.

The most significant limitation of the game engine, considering our game design, lies in the fact that it works with game levels that have to be strictly bounded. However, our game is built around a car that is speeding with no intention of stopping, hence the level would have to be considered as infinite, which is sadly impossible in the Pjs Game Engine. The workaround we have realized lies in making the car stationary at the bottom of the screen (level) and making the obstacles and lane marks move towards the car from the top, making it seem like the car is speeding with the camera following its movement forwards.

Both the Processing and Processing.js languages are built on the Java language, but have a simplified syntax. They have several additional functionalities in the area of graphics programming, since it is mainly used for electronic arts and other visual designs. With Processing.js being an object-oriented programming language, we have tried to fully utilize that fact and build our objects upon the underlying code of the Pjs Game Engine by extending their classes.

6.1.1 Level and car

Since the Pjs Game Engine is built around the use of levels, we needed to implement at least one level with one level layer for our purposes. The `Level` class is mainly a container for `LevelLayer` instances, but also defines the level's size and controls its view box (camera). In our case both the level's height and the view box's height are set to 600 and their widths are computed based on one of our game parameters - the number of lanes. Our `Level` class contains only one `LevelLayer` instance - an instance of our `MyLeverLayer` class, which extends `LeverLayer`. Lever layers are the main source of the game's content in Pjs Game Engine. They control adding characters, interactable objects and for us are an ideal place to store the obstacle generating logic in.

The method `draw()` is overriden in `MyLevelLayer` and calls the obstacle generating method `generateNewBlock()` every now and then. Another one of our game parameters, the Frequency of obstacles parameter, defines how often `generateNewBlock()` is called. The `draw()` method is called every time a new frame is requested, and since the game's frames per second setting is set to 30, we are able to uphold the frequency with a simple counter. The generating algorithm first randomly picks a lane for the obstacle to spawn in. An obstacle is created in this lane if it is guaranteed that there will be a path for the car to avoid the obstacles. That is managed through the Minimal space parameter, which is the minimal distance between the last spawned obstacles in this lane and at least one of the adjacent ones. Then the car is guaranteed to be able to avoid the obstacles by steering into one of the adjacent lanes.

The `MyLeverLayer` class also creates and adds our main character of the game - the car. It is implemented in the class `Car` by extending the `Player` class and its main purpose is to handle the keyboard input and move the car to adjacent alnes. Input handling is already mostly implemented in the Pjs Game Engine and we mainly only use those functions to register the key strokes and reposition the car accordingly, while checking for level boundaries. Pressing the key A moves the car to the left adjacent lane and pressing the key D results in the car being moved to the right adjacent lane.

6.1.2 Obstacles and other objects

The obstacles are to be considered the main enemies of our game. The class implementing an obstacle is called a `Block` and it extends the `Interactor` class. The `Interactor` objects are the interactable in-game objects or enemies and the main difference from other `Actor` objects is the ability to detect and handle interaction with other actors. The logic behind the interaction with other in-game objects or characters is stored in the `overlapOccuredWith()` method. Since our obstacles can only overlap with our car and not with each other, the interaction logic is quite clear. Whenever the obstacle overlaps with something, the game is over and started over.

Besides the obstacles, we have also implemented a trigger area based on the class `Trigger`. Triggers are areas of the level that run a certain procedure when an actor of the game passes through them. Our `SpeedupTrigger` object's area is spread over the entire width of the level and is just below the visible viewbox and the car. The main purpose of the trigger area is to speed up the game continually. Every time the car avoids an obstacle, the obstacle moves into the area of the trigger and the trigger removes the obstacle and speeds up the game by multiplying the current speed by the Speed-up coefficient parameter, if it is still below the Speed cap parameter. If the new speed is higher than the cap, it is limited to the value of the Speed cap and no longer increased every time an obstacle hits the trigger area.

The last objects we have implemented into our game are instances of the `LaneLine` class. These objects have a purely visual impact on the game, since they simulate the road beneath the car moving downwards. They are the visual separators of adjacent lanes and move downwards at the same speed as the obstacles. When they hit the speed-up trigger area, they are moved back to the top of the level. This enables us to create a better illusion of the car moving, even though it is implemented as a vertically stacionary actor.

6.1.3 Integration of the game into a web page

Applications written in Processing.js are meant to be rendered in the web element `<canvas>`, which has references to the application code. The Processing.js JavaScript file has to be present aswell to do both the code interpretation and rendering itself. We have designed the game of Speeder to work with several game parameters that are needed for the game's functionality, but are specified just before the game is instantiated. Those parameters have to be present in the web page in a JavaScript `<script>` element. That allows our MAB framework to generate the page dynamically based on the MAB algorithm we have developed.

6.2 Implementation of the MAB framework

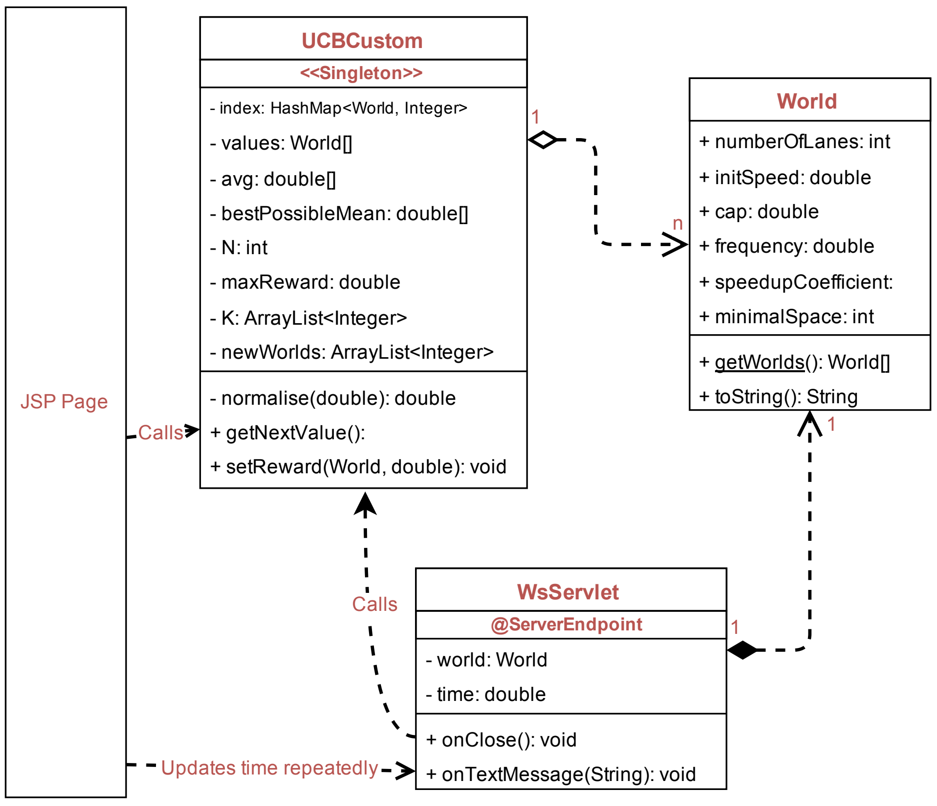


Figure 2: Class diagram of our MAB framework implementation

The main product of this thesis is the framework, providing the bandit algorithm and delivering the chosen parameters into the game itself. Our Multi-Armed Bandit framework was conceived as a server-side application, which would be running on a web server and would contain the logic of our developed MAB algorithm. The website with our game would then be able to request the new set of parameters from the web server and all the computing would be done on the server.

We have chosen to implement the framework using the Java language, specifically using the Java Enterprise Edition (Java EE) platform. Java EE is a runtime environment for developing and running network and web applications and services. It contains many API specifications that ease the developer's work with technologies broadly used in network applications, such as e-mail, XML, web services and so on. It also features several specifications unique to Java EE, for example Enterprise JavaBeans, servlets, JavaServer Pages and other.

Since the platform only describes functionalities through those specifications, the exact implementation of those functionalities depends on the application server. Due to this fact, Java EE applications with these features are required to run on a server and will not work as a stand-alone application. We are using a Java application server called Apache Tomcat. It is not considered a full Java EE server, since it lacks several features like Enterprise JavaBeans or Java Messaging Service. However, most of the missing features are of no use to us and Tomcat contains the implementaton of our most needed specifications - Java Servlets, JavaServer Pages and WebSockets. Since Java is an object-oriented programming language, we describe each component of our framework as an object.

6.2.1 Implementation of arms/machines

As we have already specified, each arm of the K-armed bandit problem [4] corresponds to a certain variant of our game. Each variant is then fully defined by its properties, the game parameters. We have implemented the variants as objects of the class `World`. The variables of each world are as follows:

Game parameter	**Property**	**Data type**
Number of lanes	`numberOfLanes`	`int`
Initial speed	`initSpeed`	`double`
Speed-up coefficient	`speedupCoefficient`	`double`
Speed cap	`cap`	`double`
Frequency of obstacles	`frequency`	`double`
Minimal space	`minimalSpace`	`int`

The instances of the `World` class have a purely data-holding purpose, hence the methods implemented in the class include only the parameters' getters and setters, and the methods `equals()` and `hashCode`. According to the principle of encapsulation, each member variable of a class should not be directly accessible from the outside of the object, but only through so-called accessors - setters and getters. A setter is a public method that alters the value of the member variable and sets it to a new, given value, while a getter is a public method that returns the value of the private property of the object. The methods `equals()` and `hashCode()` are functions that help us find the stored variants by their game parameters' values, for example when we link the arms to their reward means using a `HashMap`.

6.2.2 Implementation of the algorithm

Our developed algorithm needs to store several values, such as reward means, from the first playthrough of the game until the end of the testing phase. The instance of the object also needs to be globally accessible, since we do not intend to create a new instance every time the game is played. Due to these facts, we have decided to store the algorithm logic in a class that is implementing the Singleton design pattern. A singleton class is a type of class that cannot be instantiated more than once. When an instance of the class is requested for the first time, it is instantiated and stored into a private static variable of the class. The instance then becomes globally accessible through a static getter and remains that way until the application is stopped, restarted or undeployed. This allows us to store relevant data into the object's member variables and we are guaranteed the data will not be deleted by Java's garbage collector.

The entire design was implemented in the class `UCBCustom`. The object is provided with an array of all the possible variants during its first instantiation and then proceeds to create all the required data structures necessary for the algorithm to function properly. Those include an array to hold all the corresponding reward means (`avg`), an array containing the number of plays of each arm (`numOfPlays`) and an array used for computing the upper confidence bound indices - `bestPossibleMean`. All of the arrays have a length equal to the total number of all variants. A mapping $World \rightarrow arrayindex$ is created and stored in a `HashMap` to help us later on find out which array index belongs to a certain set of game parameters. We also work with two `ArrayLists` - `K` and `newWorlds` - in the UCB-AIR inspired part of the policy, that lets us work with a high amount of arms even in the case of small or limited testing. All of these data structures are vital for the algorithm and are all stored as member variables of the object. The algorithm also stores the maximum reward recorded sofar in the variable `maxReward`, which allows us to normalise the rewards to the bounded interval [0,1]. The last data structure included in the object is a simple integer counter, that counts the total number of plays. There are no getters and setters implemented, since all of these variables are only used within the class itself. The whole process of the UCB1 algorithm is then divided into the two main public methods - `getNextValue()` and `setReward()`. Other methods do not contain the logic of the policy and are present to simplify logging.

The method `getNextValue()` is called when the next arm needs to be chosen - in our case the method's return value is requested when the website of the game is loading for a user. The method returns the arm it has chosen to play - a `World` object containing a certain combination of parameters that are to be used by the game. The method works according to our designed UCB algorithm and uses two array lists `K` and `newWorlds` to properly expand our set of explored worlds continually. `K` represents the worlds already included in the previous plays, while `newWorlds` contains all the possible unexplored worlds. If one of the arms in the set `K` wasn't played yet - their corresponding `numOfPlays` value equals to 0 - the arm is chosen. If none of the `numOfPlays` values for the arms in `K` are equal to 0, the upper confidence bound indices are computed as in 4 for each explored machine and stored in the `bestPossibleMean` array. The world that corresponds to the maximum value of this array is then returned by the method.

The method `setReward()` is the second part of our algorithm, used when the reward is to be recorded into our framework. It has to be separated from the first part, since we need to wait for the user to finish playing the game before we collect the final reward. Before the received reward can be recorded and our stored reward means accordingly amended, the reward neeeds to be first normalised as specified in Section 4.3. The method contains the implementation of the normalisation procedure and uses it before storing the new data.

6.2.3 Dynamic web page

The web page containing the game of Speeder is implemented using a Java EE specification called JavaServer Pages. This specification is used to create dynamically generated web pages, which allows us to initialize the game with different parameters each time the page is requested, based on the chosen variant by our MAB algorithm. JavaServer Pages are written in HTML, with the addition of JSP scriptlets that are written in the Java language. The JSP is compiled by the application server the first time it is accessed. The scriptlets inside the JSP are run when a user requests the page and using those fragments of Java code we are able to create the page dynamically.

When our JavaServer Page is requested by a user, we first get ahold of the instance of our UCB algorithm by calling the `getInstance()` method on our singleton class containing the algorithm. We proceed by querying the algorithm for the next chosen variant and its corresponding game parameters. Then we put the values of those parameters inside a `<script>` HTML element in the form of JavaScript. The parameters then become visible and usable for the game.

6.2.4 Collecting the reward

Sadly we are unable to provide the reward to the UCB algorithm using a JSP scriptlet, since they are run only when the page is requested. Quitting the game is possible only through closing the browser or leaving the page, which we are also unable to intercept, because each browser treats those events differently. The workaround we have implemented works with one other Java EE specification - WebSockets. This technology is used for communication between the client (the browser) and the server.

Our JavaServer Page estabilishes the connection to our server-side implementation, which is stored in a class called `WsServlet`. The class is marked by the `@ServerEndpoint` annotation and implements methods `onClose()` and `onTextMessage()`. The page includes a JavaScript timing event, which continuously waits and executes a function at given time-intervals. The function sends the used variant and its current reward information - game parameters and time so-far played - to the server endpoint of the connection over and over. The servlet is instantiated once per connection, so every time it receives a message containing the new information, it updates its corresponding member variablee with the current reward. When the connection is closed, the `onClose()` method is called and the reward with the corresponding variant is passed on to our multi-armed bandit algorithm. This way we have guaranteed the safe transfer of our reward after the user is done playing the game.

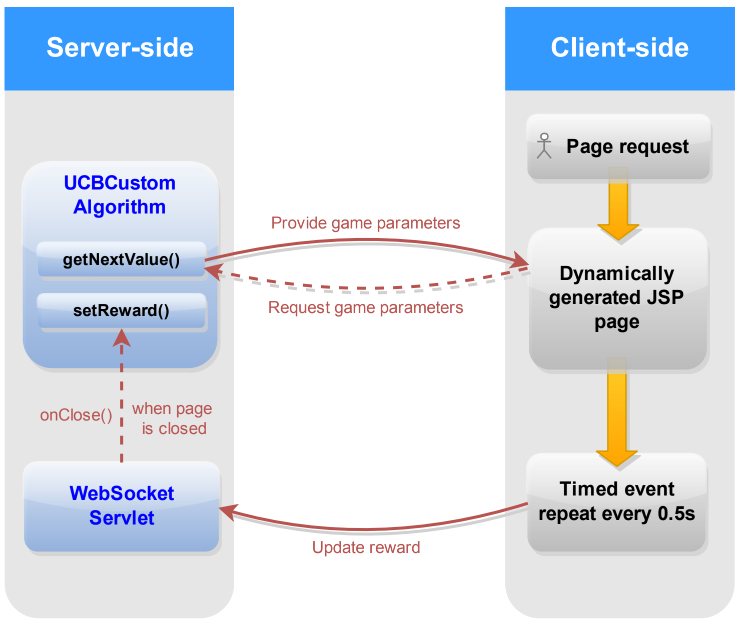


Figure 3: Server-client communication in our framework

7 Evaluation

The final goal of the thesis was to evaluate our designed and implemented algorithm on our created game Speeder. In this chapter, we describe the testing environment we have tested the algorithm in, illustrate the resulting data we have obtained and discuss the outcome of the testing phase.

7.1 Testing environment

Since our algorithm doesn't require the total number of plays to be known beforehand, the simplest solution to testing our framework was to take our implemented Java EE application and upload it to a public web server, making it accessible via a website on the Internet in the process. The deployment of the application on the server marked the start of the testing phase of our algorithm. We have distributed the website to our friends and colleagues with the intention of collecting as much data as possible. We have also designed a logging mechanism beforehand for the purpose of monitoring the collected data by implementing a singleton object in our application containing the desired logs. Corresponding JavaServer Pages were created to allow us to view and collect the resulting data from the logs located in the singleton object.

We have already specified the domains of our game parameters in chapter 4, but we still need to further limit the possible values of the parameters to just a few values for our limited testing environment. We have decided to choose 4 significant possible values for each parameter, which results in a total of 4096 arms in our MAB problem. The possible values for each parameter are usually not picked evenly from their domains, since values near the bounds of the domains often already result in an unplayable game. The following table contains the chosen possible values for each of our game parameters.

Game parameter	**Values**			
Number of lanes	2	3	4	5
Initial speed	2	4	6	8
Speed-up coefficient	1.02	1.05	1.15	1.5
Speed cap	10	12	15	18
Frequency of obstacles	0.1	0.3	0.5	0.8
Minimal space	160	180	200	220

Even though we have limited and discretized our game parameters properly, our MAB algorithm still needs to take into account the high amount of arms compared to the small expected number of plays during our testing. As we have described in chapter 5, our designed MAB strategy starts with an empty set of arms A and with the increasing number of plays keeps expanding the set of arms that an UCB policy is applied on. The speed at which our MAB algorithm expands the set of arms depends on a parameter β - if $\beta > 1$, the set A expands in the n-th round only if the size of A is lower than the value of $n^{\frac{\beta}{\beta+1}}$. Therefore β can be viewed as the algorithm's exploration parameter, that defines the way the algorithm handles and explores the considerably high amount of arms. To test the significance of this parameter, our testing environment contains three instances of our game and it's associated MAB algorithm, that differ in the value of the exploration parameter β. UCBCustom1 is the main variant of the algorithm with the default value of $\beta = 1$, while UCBCustom2 and UCBCustom3 have the parameter set to $\beta = 2$ and $\beta = 3$ respectively.

Multi-armed bandit strategies are proved to work correctly and efficiently in a set of certain conditions, that we unfortunately cannot guarantee in our particular use case. Bandit strategies aim to create a sequence of plays, where the next play is chosen always after the reward from the last round is collected. However, we cannot guarantee the sequentiality of successive plays in our testing environment. If a player begins the game before the last user is finished playing, our multi-armed bandit framework will instantiate the game for the new player with the same combination of parameters as the previous user. In theory it corresponds to playing the arm twice instead of once after it has been chosen. MAB strategies also require the arms to return a reward from a bounded interval, but our original time-based reward does not have an upper bound at all. We have used a normalisation technique described in section 4.3, that is widely considered to work quite well in practical cases. However, there are no theoretical guarantees for this normalisation technique and both of the described issues can hinder the effectiveness of our designed algorithm in our testing environment.

7.2 Examples of UCB indices

In this section we will illustrate the evolution of the upper confidence bound indices as the number of plays grows. The upper confidence bound index of an arm consists of two parts - the average reward of the arm and the bias factor. The index is computed as a sum of those two parts and the significance of the bias factor shifts over to the average reward as the number of rounds increases. Furthermore, the bias factor is smaller for arms that have been played more often and bigger for arms the algorithm has not experimented with very much. The following tables show how the upper confidence bound indices evolve over time and illustrate how they affect the balance between exploration and exploitation.

Arm	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Number of plays	8	9	10	7	5	5	5	1
Average reward	0.281	0.409	0.422	0.208	0.059	0.110	0.078	0.063
Bias factor	0.988	0.932	0.885	1.057	1.251	1.251	1.251	2.797
UCB1 index	1.269	1.341	1.307	1.265	1.310	1.361	1.329	**2.860**

Table 1: UCB indices for 8 selected arms after $n = 50$ rounds

Arm	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Number of plays	13	38	16	12	11	10	10	12
Average reward	0.135	0.487	0.198	0.099	0.028	0.027	0.015	0.101
Bias factor	0.865	0.547	0.81	0.899	0.976	1.006	1.006	0.899
UCB1 index	1.000	**1.034**	1.008	0.998	1.004	1.033	1.021	1.000

Table 2: UCB indices for 8 selected arms after $n = 190$ rounds

7.3 Results and discussion

We have collected the resulting data after a period of time that encompassed a total of 426 plays of our game Speeder. The game instance associated with the main variant UCBCustom1 of our UCBCustom algorithm was played 190 times, the second variant UCBCustom2 was played 153 times and the last UCBCustom3 had 83 rounds recorded. The main variant UCBCustom1 is the most likely to show convergence, since it is the least explorative variant and was played the most from the three instances.
From the total of 4096 arms, UCBCustom1 explored 14 arms over the course of its 190 rounds, UCBCustom2 had a total of 29 explored arms after 153 plays and UCBCustom3 was able to try out 28 arms in only 83 rounds. Just from this short summary we can clearly see the significance of our algorithm's exploration parameter β. UCBCustom3 explored the same amount of arms nearly twice as quickly as its less explorative sibling UCBCustom2. Even though we will be able to explore more arms with the UCBCustom3 version of our algorithm, the high rate of exploration will most likely hinder the strategy's ability to identify and exploit the best arm from the already explored arms.
We have created several graphs illustrating the development of the number of plays of each arm as the total number of plays grows for each of the versions of our algorithm. Those graphs can be found in appendices A,B and C for the versions UCBCustom1, UCBCustom2 and UCBCustom3 respectively. In the appendices you can find two columns of graphs, where one column contains bar graphs showing the number of plays for each world (arm) and the second one illustrates the corresponding average rewards computed from received rewards for these arms.

As we can see from the graphs in appendix A, our previous prediction that the UCBCustom1 variant will show the strongest signs of convergence was confirmed during our evaluation phase. The exploitation portion of our algorithm can be clearly visible if we compare the number of plays of arm number 3 in the graphs A.2 and A.3. In graph A.2, which represents the state of the MAB problem after 120 rounds, the world with the highest average reward was chosen approximately twice as much as the other suboptimal worlds. After another 70 recorded plays, this optimal arm is played nearly three times as much as some of the worse performing arms and the algorithm was still able to explore three new worlds in the meantime. The most optimal arm found by UCBCustom1 corresponds to a game variant with three lanes, where the car speeds up quickly and continues travelling at a very high speed. The game parameters for this world have the minimal space between obstacles set to the lowest setting, which results in a high pace game that requires the player to have quick reactions and fully concentrate when avoiding obstacles. It is not surprising that this version of our game is considered as the most enjoyable by our algorithm.
UCBCustom2, the second version of our algorithm, is a more explorative variant of the algorithm compared to UCBCustom1. However, we can still show some, but much weaker signs of convergence in the graphs provided in appendix B. The arm number 18 has the highest average reward in both graph B.2 and B.3. Even though the graphs are separated by 53 rounds, the best world was chosen only once during those rounds. The reason lies in the explorative properties of our algorithm - instead of exploiting the best arm as much as possible, the strategy sacrificed a big portion of the 53 rounds to explore new options and worlds due to the higher β parameter. However, if we continued the experiment, we would see the best arm played more and more often as the pauses between adding new arms would grow longer, leaving more opportunities for the exploitation part of our algorithm.
The results obtained from UCBCustom3 do not show nearly any attempts of exploitation due to the very low amount of rounds. After 83 plays, the version UCBCustom3 has not yet advanced from the first, exploration-heavy phase of the algorithm and we can see the speed of the algorithm's convergence is heavily reliant on the exploration parameter β. With versions with higher β values we would need to wait a much longer time for the algorithm to fully converge. However, one of the goals was to test which one of the three versions of our algorithm most effectively exploits the found optimal arm while exploring the suboptimal arms at the same time and from that perspective, the UCBCustom1 variant is clearly the best one.

8 Conclusion

We have studied and researched the multi-armed bandit problem and its associated exploration versus exploitation dilemma and presented an overview of typical strategies solving this particular problem in theoretical environments. We have described several practical use cases, where the multi-armed bandit problem is effectively used, and proposed the potential of using the bandit algorithms in freemium games.
We have formalized the multi-armed bandit for the specific environment of a simple online game with variable game parameters and shown how it can be mapped to the case of pricing the products in freemium games. For this formalized setting, we have then developed our own bandit algorithm UCBCustom, which is mainly based on upper confidence bound algorithms and is also able to solve the multi-armed bandit problem in environments, where the amount of arms exceeds the total number of plays.
Given the risk of losing a company's profit associated with evaluating the algorithm on an existing freemium game, we have rather designed and implemented a simple online game Speeder, where the player controls a speeding car and tries to avoid obstacles by switching lanes. The design of Speeder includes variable game parameters, that have a significant impact on the gameplay. Instead of finding the most profitable attributes and prices of premium features in a freemium game, we have used our algorithm to identify the most enjoyable variant of our game Speeder and its underlying combination of game parameters.
For the evaluation phase, we have created three instances of our algorithm, each with a different exploration parameter β that defines the strategy's behaviour when exploring new arms. Higher β value results in a more explorative strategy, while a version with a lower β value will not expand the set of explored arms very often. The default version, UCBCustom1 with the exploration parameter set to $\beta = 1$, ended up showing the greatest promise in minimizing the regret and playing the best arm significantly more than the other suboptimal arms, while still exploring other options. The other two versions, UCBCustom2 ($\beta = 2$) and especially UCBCustom3 ($\beta = 3$), showed weaker signs of convergence than UCBCustom1. Those versions would require a much higher number of rounds to advance from the early state mainly consisting of exploration into a phase where the exploitation of the best arm would become more apparent.

References

[1] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 1952.

[2] John Gittins. *Multi-armed Bandit Allocation Indices.* John Wiley and Son, 1989.

[3] T. L. Lai and Herbert Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 1985.

[4] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 2002.

[5] Scott and Steven L. A modern bayesian look at the multi-armed bandit. *Appl. Stoch. Model. Bus. Ind.*, 2010.

[6] Janis Hardwick and Quentin F. Stout. Bandit strategies for ethical sequential allocation. *Computing Science and Statistics*, 1991.

[7] Dirk Bergemann and Juuso Valimaki. Bandit problems. Cowles Foundation Discussion Paper No. 1551, 2006.

[8] Nicolas Galichet, Michèle Sebag, and Olivier Teytaud. Exploration vs exploitation vs safety: Risk-aware multi-armed bandits. *Asian Conference on Machine Learning 2013*, 2014.

[9] Michael N. Katehakis and Arthur F. Veinott Jr. The multi-armed bandit problem: Decomposition and computation. *Mathematics of Operation Research*, 1987.

[10] Sébastien Bubeck and Nicolo Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, 2012.

[11] Joannes Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *Machine Learning: ECML 2005.* Springer Berlin Heidelberg, 2005.

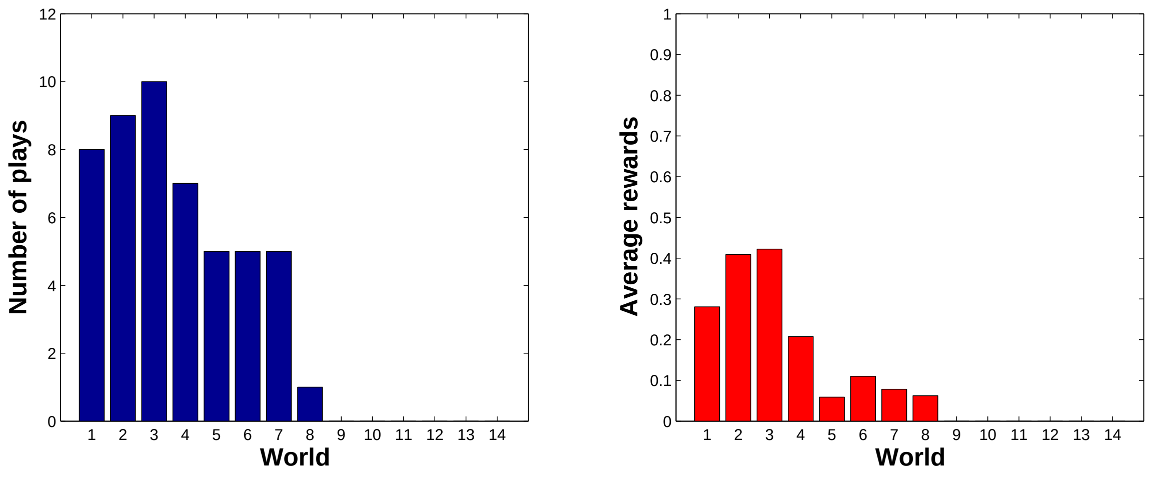
[12] J.-Y. Audibert, R. Munos, and Cs. Szepesvári. Tuning bandit algorithms in stochastic environments. *Algorithmic Learning Theory*, 2007.

[13] Yizao Wang, Jean yves Audibert, and Rémi Munos. Algorithms for infinitely many-armed bandits. In *Advances in Neural Information Processing Systems 21.* Curran Associates, Inc., 2009.

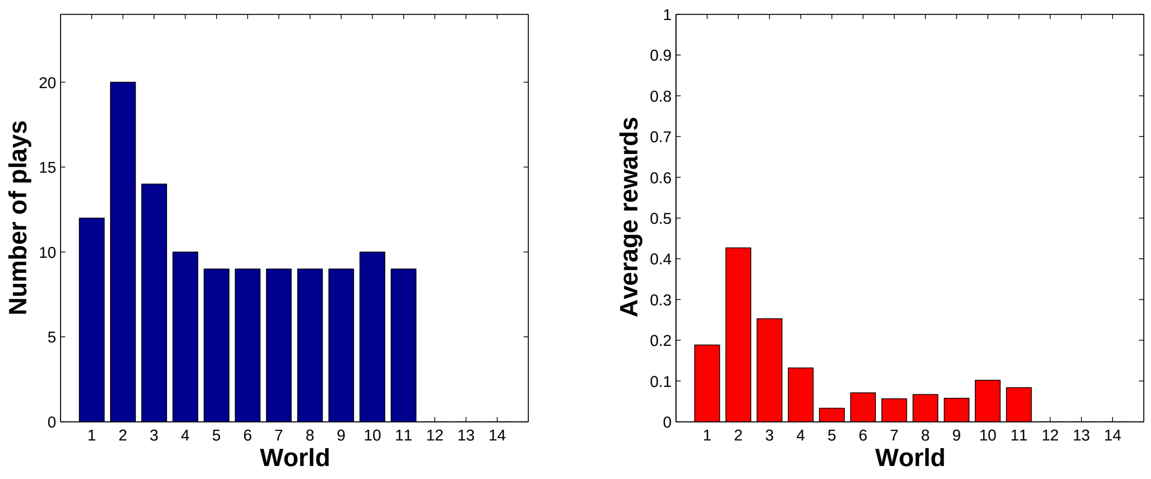
[14] J.-Y. Audibert, R. Munos, and Cs. Szepesvári. Exploration-exploitation trade-off using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 2009.

Appendix A - UCBCustom1 graphs

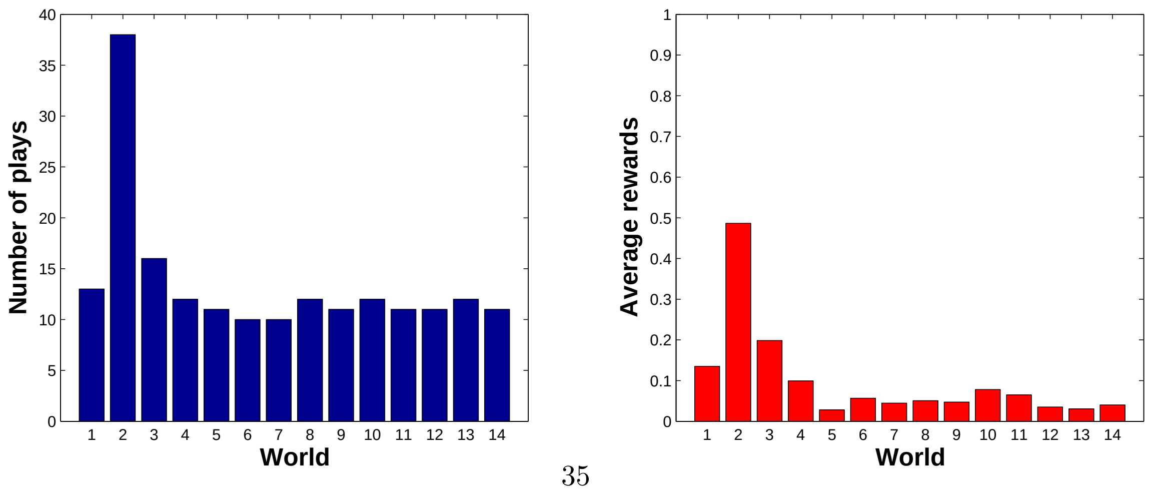
Graph A.1 After $n = 50$ plays



Graph A.2 After $n = 120$ plays

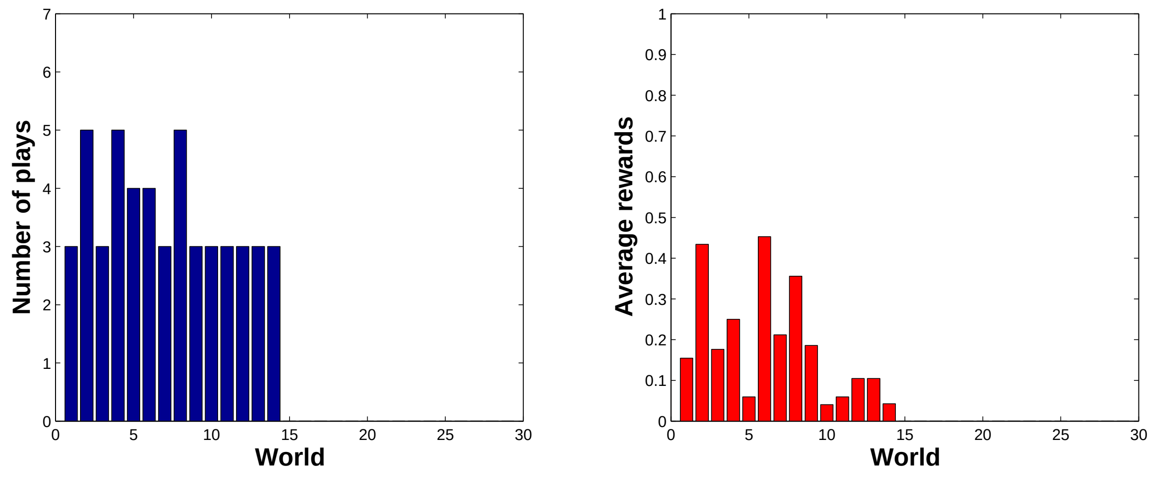


Graph A.3 After $n = 190$ plays

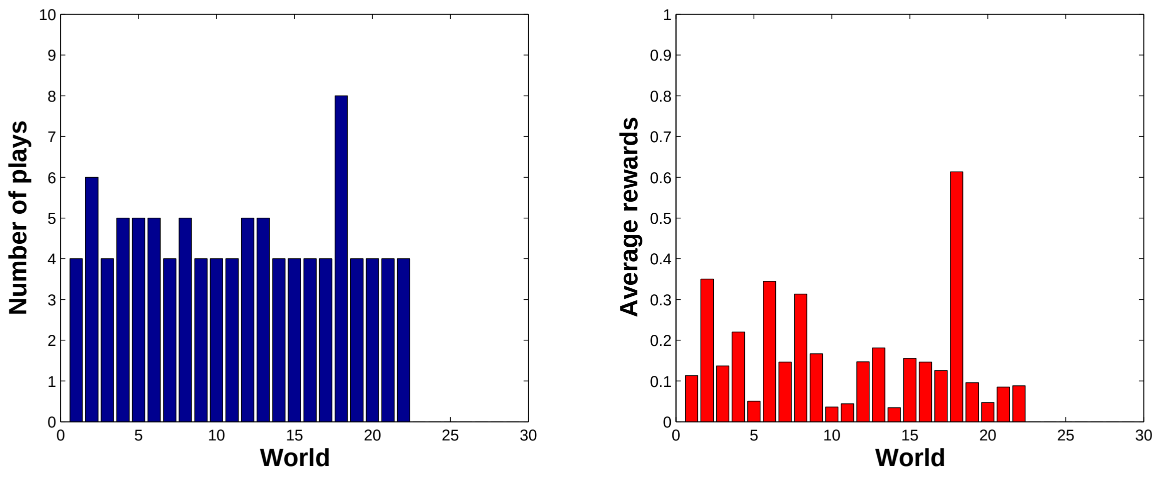


Appendix B - UCBCustom2 graphs

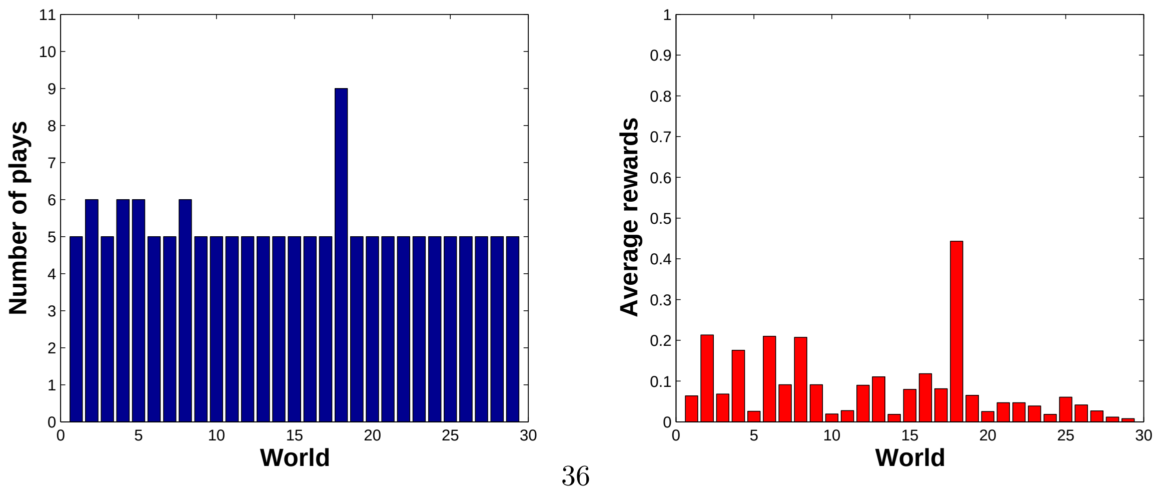
Graph B.1 After $n = 50$ plays



Graph B.2 After $n = 100$ plays

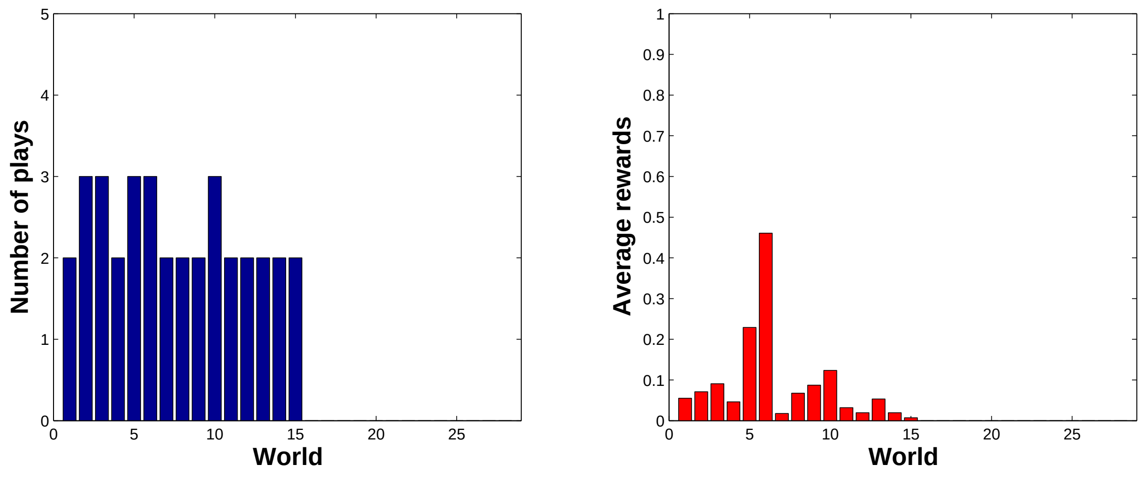


Graph B.3 After $n = 153$ plays

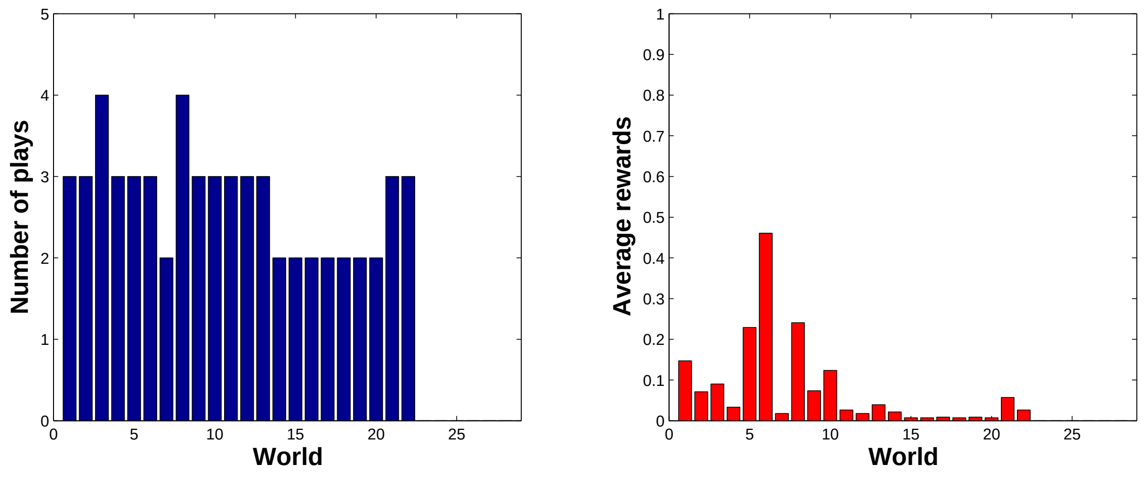


Appendix C - UCBCustom3 graphs

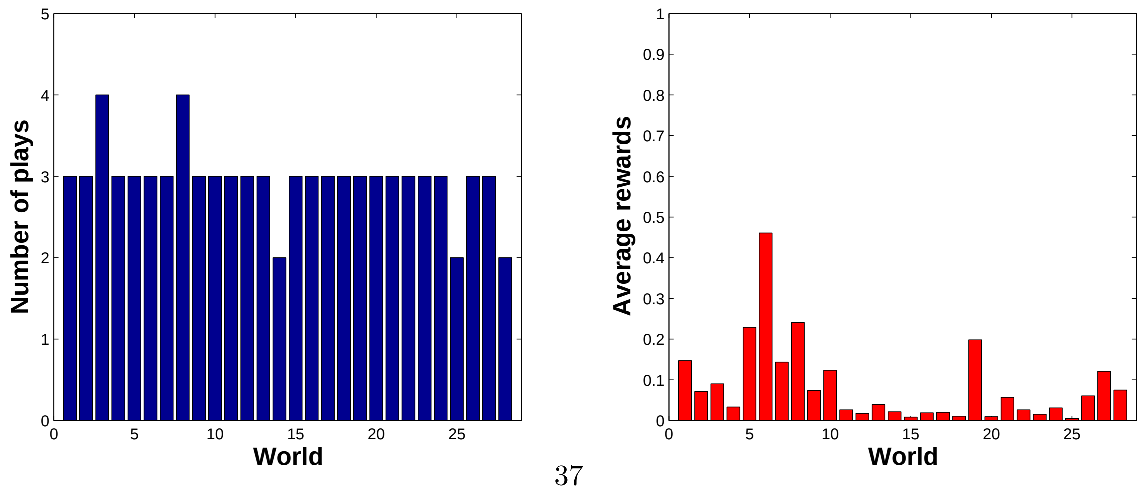
Graph C.1 After $n = 35$ plays



Graph C.2 After $n = 60$ plays



Graph C.3 After $n = 83$ plays


Number of plays
World
Average rewards
World