Towards Embedded Implementations of Multi-Armed Bandit Algorithms for Optimised Channel Selection

Alex Kangas
Abstract

Towards Embedded Implementations of Multi-Armed Bandit Algorithms for Optimised Channel Selection

Alex Kangas

The proliferation of Internet of Things increases the demand for achieving a high performance for embedded devices on the Internet. The IEEE 802.15.4 wireless network standard provides ultra low complexity, cost, power consumption and data rate for inexpensive devices, although it has problems such as external interference and multi-path fading. The Time Slotted Channel Hopping (TSCH) and MiCMAC are examples of medium access control (MAC) protocols who mitigate these problems by employing channel hopping. However, their limitations are that they predefine channel hopping sequences at compile time and do not perform any optimisations. A suggested solution to these issues are the use of Multi-Armed Bandit (MAB) algorithms, which have shown to give significantly better performances compared to TSCH. However, these algorithms have only been simulated and not implemented for embedded devices.

In this thesis, we present embedded implementations of the MAB algorithms Sliding-Window Upper Confidence Bound (SW-UCB) and Discounted-UCB (D-UCB) in the C programming language with fixed-point arithmetic as an alternative for embedded microprocessors that lack a floating-point unit. We measure the performances of the implementations in terms of packet delivery ratio (PDR) by running them with different parameter settings in simulated stationary and non-stationary environments, where they are compared with corresponding floating-point implementations of these algorithms. The results of the experiments are very promising, no significant differences in PDR between the results of the floating-point and fixed-point implementations are shown, which further suggests that MAB algorithms are viable options for optimised channel selection. The results instead depended more on the proper parameter settings for each algorithm, where the optimal settings differed depending on the environment.
Chapter 1

Introduction

The emergence of the internet of things (IoT) has led to a greater prevalence of embedded devices connected to the Internet [4]. In order to accommodate the proliferation of IoT, network protocols for embedded devices have developed to ensure that they achieve a high performance. A prevalent standard used is the IEEE 802.15.4, which is a radio network standard that provides ultra low complexity, cost, power consumption and data rate for inexpensive devices [1].

However, there are two main problems that can occur in networks of this standard. One is external interference, which means that several network standards operate on the same frequency band and interfere with each other, which can cause packet loss. The other problem is multi-path fading, which occurs when not only a signal is received from the direct line-of-sight path, but multiple copies of that signal is received as a result of it reflecting on other objects [15]. This can cause the signals to destructively interfere with each other and therefore make the receiver node unable to decode the signals. However, the signals may also constructively interfere with each other and therefore make them stronger and easier to decode.

A way to mitigate these problems is the use of channel hopping, a method where the radio channel selected for transmission periodically changes [15]. A commonly used channel hopping medium access control (MAC) protocol is Time-Slotted Channel Hopping (TSCH) [14]. Another promising channel hopping extension to low-power listening MAC protocols is MiCMAC [10]. For instance, when run in a 97-node testbed while running a complete low-power IPv6 stack and with RPL at the routing layer, MiCMAC achieves an end-to-end packet delivery ratio (PDR) of 99% in noiseless channels [10]. However, there are limitations to TSCH’s and MiCMAC’s applications of channel hopping. Both use predefined channel hopping sequences, meaning that there is no guarantee that the selected channels for transmission are unaffected by external interference or multi-path fading.

A suggested solution to this problem is the use of Multi-Armed Bandit (MAB) algorithms [7]. These algorithms solve the kinds of problems where we want to maximise the accumulation of rewards from a number of options,
were the reward distributions between these options are unknown \cite{5,11,13}. Dakdouk et al. suggest that the MAB algorithms can be used to automatise channel hopping by selecting the most optimal channel before each packet transmission. Furthermore, they implement 9 MAB algorithms and compare them with TSCH by measuring their performance in terms of PDR. For example, in an environment of experimentally-decided channel PDR values based on the FiT IoT-LAB platform, the best performing algorithms with 95\% confidence intervals were Thompson Sampling (TS), Switching TS (STS), Switching TS with Bayesian Aggregation (STSBA), Upper Confidence Bound (UCB) and Sliding-Window UCB (SW-UCB) \cite{7}.

1.1 Problem Statement

The main limitation with Dakdouk et al. is that they only provide a Matlab simulation of the MAB algorithms and not actual embedded implementations of them. Making embedded implementations of them causes some issues to arise. Since embedded microprocessors generally lack a floating-point unit (FPU), these algorithms have to be implemented with an alternative to the use of floating-point. One solution is the use of fixed-point numbers, which are stored as integers in memory but where an implicitly specified number of bits represent the fractional part of the number \cite{9}.

Consequently, this solution causes new issues to erupt. Since fixed-point arithmetic uses a specific number of bits to represent fractions, the range of the fractions that can be represented are more limited in comparison with floating-point numbers. This could potentially make fixed-point numbers more inaccurate than floating-point values, and therefore possibly become too impractical to use for implementing MAB algorithms.

Additionally, implementing TS is particularly complicated without floating-points. This is because it uses a function for drawing samples from a beta distribution \cite{12}. While there exists floating-point implementations for this function, for instance provided in the NumPy-C library, there does not seem to exist a corresponding fixed-point implementation of it. This is an issue since implementing it appears to be considerably convoluted.

Moreover, SW-UCB is also problematic since it makes use of a sliding window, which means that a potentially large amount of samples need to be stored in memory \cite{8}. A more memory efficient alternative to this is the use of the algorithm Discounted-UCB (D-UCB), which has performed slightly worse than SW-UCB but uses less memory by storing its samples into a single aggregate sum \cite{8}.

1.2 Contributions

In this thesis, we provide:
• Implementations of SW-UCB and D-UCB in the C programming language with fixed-point arithmetic as an alternative to floating-point arithmetic.

• Implementations of SW-UCB and D-UCB in the C programming language with floating-point arithmetic in order to compare them with the fixed-point implementations.

• Simulations of the implementations in different environments and parameter settings.

We implement the SW-UCB algorithm since it was one of the best performing ones presented by Dakdouk et al. [7]. TS and its non-stationary variants STS and STSBA were not chosen due to the limitation of TS for fixed-point implementations mentioned earlier. Moreover, UCB was not chosen either since UCB is adapted for the stationary case while SW-UCB is an adaptation of UCB for non-stationary environments which more accurately depict channels whose PDRs changes over time. D-UCB was also chosen due to the memory limitation of SW-UCB mentioned earlier.

In order to measure the performance of the implementations, we implement a floating-point implementation of SW-UCB and D-UCB respectively in order to compare the impact of the fixed-point operations, with PDR as the metric for measuring their performances. We tested in simulations with 16 channels where the PDRs of each channel were randomised with one channel having a PDR of 1.0 and the average PDR of all channels being 0.5.

1.3 Outline

The outline of the thesis consists of a background in Chapter 2, which contains the information for understanding the MAB problem and SW-UCB and D-UCB. Then, in Chapter 3, we describe our design of the fixed-point implementations of SW-UCB and D-UCB. After that, we describe the implementations of SW-UCB and D-UCB in Chapter 4 followed by the descriptions of the experiments, results and discussion in Chapter 5. Finally, we present the conclusion in Chapter 6 including suggested improvements for possible future works.
Chapter 2

Background

In this Chapter, we outline the background information of the MAB algorithms necessary for understanding and implementing the algorithms.

2.1 Multi-Armed Bandit

The Multi-Armed Bandit (MAB) problem refers to problems, where we have several options known as arms. Each arm yields different values known as rewards [5, 11, 13]. The goal of the MAB problem is to maximise the cumulative reward by choosing the arm at each time step that yields the highest reward. However, the distribution of these rewards are unknown at the beginning.

Suppose that we have $K$ arms, where each arm has a reward distribution. Since the reward distributions are unknown, the only way to learn more about the rewards is to select an arm and estimate the true reward based on the results of previous selections, which is known as exploration. Another way to select an arm is to select the optimal based on the current information of the rewards that have been gathered from exploration, which is called exploitation. Therefore, the objective of an MAB algorithm is to find a balance between exploration and exploitation so that the cumulative reward over some period of time steps is maximised.

As has been suggested, wireless channel selection can be expressed as a MAB problem where each arm corresponds to a channel where the reward represents a successful packet transmission over a channel [7]. This means that each arm only yields two values: one to represent a successful packet transmission and the other representing failure. A MAB problem which only yields two values from an arm like this is known as a Bernoulli bandit.

Using mathematical notation, the Bernoulli bandit problem consists of an arm $a \in \{1, 2, \ldots, K\}$ which gives a reward $X_t(a) \in \{0, 1\}$ at time step $t$ where $X_t(a) = 1$ yields with probability $\theta_{t,a}$ [12]. Otherwise, it yields $X_t(a) = 0$ with probability $1 - \theta_{t,a}$. At each time step $t$, we select an arm to perform according to the algorithm that we have chosen.
2.1.1 Stationary environment

There exist different types of MAB environments. An environment where each reward distribution remains constant during a whole time horizon is called a stationary environment [8]. Some MAB algorithms that are adapted for this kind of environment are TS and UCB [8, 12].

SW-UCB and D-UCB were the algorithms that we chose to study for this thesis. However, since SW-UCB is based and considerably similar to the UCB heuristic UCB1, the latter algorithm will also be explained in order to better understand the former.

2.1.2 Upper Confidence Bound

Upper Confidence Bound (UCB) refers to a family of MAB algorithms that balances the exploration vs exploitation dilemma by estimating confidence intervals for each arm and selecting the one with the highest upper bound. Let $X_t(a)$ be the true reward of an arm $a$ at time step $t$. Then, for every arm $a$, we estimate an upper confidence bound so that with high likelihood:

$$X_t(a) < \hat{X}_t(a) + c_t(a)$$

where $\hat{X}_t(a)$ is an estimator of the average reward of $a$ at time step $t$ and $c_t(a)$ is known as a padding function of $a$ at time step $t$.

There are several heuristics of UCB that define $c_t(a)$ differently. One heuristic of UCB is the UCB1 algorithm, which defines it by having its value reduced over time [5]. It is defined as follows:

$$c_t(a) = \sqrt{\frac{2 \ln t}{N_t(a)}}$$

where $N_t(a)$ is the number of times arm $a$ has been selected by time step $t$.

Algorithm 1 shows a pseudocode algorithm for UCB1. It starts off by applying each arm once and observing their resulting value. Then, for each time step, it chooses the arm with the greatest upper bound based on the previous results from selecting the arm.

2.2 Non-stationary environment

A non-stationary environment is an environment where each of the unknown reward probabilities regularly change their values. An instance of this is known as piecewise-stationary environment, which means that each $\theta_a$ remain constant over time as in the stationary environment until we reach a breakpoint and the reward distributions abruptly changes [16]. MAB algorithm that are adapted for this type of environment are SW-UCB and D-UCB [8].
Algorithm 1 UCB1

1: Input:
2: $K$ = Number of arms
3: for $t = 1, 2, ..., K$ do
4: Choose arm $a = t$ and observe reward $X_t(a)$
5: end for
6: for $t = K + 1, K + 2, \ldots$ do
7: $a \leftarrow \arg \max_{a \in \{1, 2, \ldots, K\}} \bar{X}_{t-1}(a) + c_{t-1}(a)$
8: Choose $a$ and observe reward $X_t(a)$
9: end for

2.2.1 SW-UCB

SW-UCB is similar to UCB1 except for some major differences in calculating the upper bound. This algorithm only takes into account some time steps back when calculating the upper bound, as opposed to UCB1 which records all the previous samples [8]. The maximum number of selected time steps that are considered is known as the sliding window, which has a size of $\tau$. Here, the estimated reward probability of an arm $a$ is the sample mean of the accumulated rewards of $a$ from the last $\tau$ time steps:

$$\bar{X}_t(\tau, a) = \frac{1}{N_t(\tau, a)} \sum_{s=t-\tau+1}^{t} X_s(a) \mathbb{1}_{\{A_s = a\}}$$

where $A_s \in \{1, 2, \ldots, K\}$ is the arm chosen at time step $s$ and:

$$\mathbb{1}_{\{A_s = a\}} = \begin{cases} 1, & \text{if } A_s = a \\ 0, & \text{if } A_s \neq a \end{cases}$$

and $N_t(\tau, a)$ is the number of samples of $a$ that are currently present in the sliding window:

$$N_t(\tau, a) = \sum_{s=t-\tau+1}^{t} \mathbb{1}_{\{A_s = a\}}$$

The padding function $c_{t}(\tau, a)$ is defined as:

$$c_t(\tau, a) = B \sqrt{\frac{\xi \ln \min(\tau, t)}{N_t(\tau, a)}}$$

where $B$ is the upper bound of the possible rewards such that $X_t(a) \in [0, B]$ and $\xi$ is an appropriate constant such that $\xi > 0$. 
However, due to the limited size of the sliding window, we might end up in a situation where there are no remaining samples of an arm \( a \) inside it, which means that \( N_t(\tau, a) = 0 \). This would cause division by zero in both \( \bar{X}_t(\tau, a) \) and \( c_t(\tau, a) \). We present a solution to this issue in Chapter 4.

Algorithm 2 shows a pseudocode implementation of SW-UCB.

### Algorithm 2 SW-UCB

1: **Input:**
2: \( K \) = Number of arms
3: \( \tau \) = Sliding window size
4: \( \xi > 0 \)
5: \( B \) = Upper bound of the possible rewards
6: **for** \( t = 1, 2, \ldots, K \) **do**
7: Choose arm \( a = t \) and observe reward \( X_t(a) \)
8: **end for**
9: **for** \( t = K + 1, K + 2, \ldots \) **do**
10: \( a \leftarrow \arg\max_{a \in \{1, 2, \ldots, K\}} \bar{X}_{t-1}(\tau, a) + c_{t-1}(\tau, a) \)
11: Choose \( a \) and observe reward \( X_t(a) \)
12: **end for**

### 2.2.2 D-UCB

Discounted Upper Confidence Bound (D-UCB) is an alternative to SW-UCB, which is also adapted for non-stationary environments. However, instead of having a sliding window, it uses a discount factor \( \gamma \in (0, 1) \) which puts more weight on recent samples than older ones [8]. Here, the sample mean of the rewards is defined as:

\[
\bar{X}_t(\gamma, a) = \frac{1}{N_t(\gamma, a)} \sum_{s=1}^{t} \gamma^{t-s} X_s(a) \mathbb{1}_{\{A_s = a\}}
\]

where:

\[
N_t(\gamma, a) = \sum_{s=1}^{t} \gamma^{t-s} \mathbb{1}_{\{A_s = a\}}
\]

Moreover, the padding function for D-UCB is:

\[
c_t(\gamma, a) = 2B \sqrt{\frac{\xi \ln n_t(\gamma)}{N_t(\gamma, a)}}
\]

where \( \xi > 0 \) is an appropriate constant, \( B \) is the upper bound on the rewards such that \( X_t(a) \in [0, B] \) and:
\[ n_t(\gamma) = \sum_{a=1}^{K} N_t(\gamma, a) \]

Algorithm 3 shows a pseudocode implementation of D-UCB.

**Algorithm 3 D-UCB**

1: **Input:**
2: \( K \) = Number of arms
3: \( \gamma \) = Discount factor
4: \( \xi > 0 \)
5: \( B \) = Upper bound of the possible rewards
6: **for** \( t = 1, 2, \ldots, K \) **do**
7: \( a = t \) and observe reward \( X_t(a) \)
8: **end for**
9: **for** \( t = K + 1, K + 2, \ldots \) **do**
10: \( a \leftarrow \arg \max_{a \in \{1, 2, \ldots, K\}} X_{t-1}(\gamma, a) + c_{t-1}(\gamma, a) \)
11: \( a \) and observe reward \( X_t(a) \)
12: **end for**
Chapter 3

Design

In this chapter, we present the design of our implementations of SW-UCB and D-UCB. In particular, we discuss the implementation of fixed-point arithmetic into the algorithms and the necessary mathematical functions.

3.1 Fixed-Point Arithmetic

One problem with implementing SW-UCB and D-UCB is that embedded microprocessors generally lack an FPU. This means that an embedded implementation has to avoid using floating-points in order for it to work correctly for embedded microprocessors. One way to do this is to use what is known as fixed-point arithmetic.

The idea with fixed-points is that each number is internally represented as integers with an implicitly specified radix point, where the digits on the left-hand side of the point define the integer part of the number while the right-hand digits define the fractional part. For instance, the five-bit fixed-point number “101.01” has 101 as its integer part, “01” is the fractional part and the dot is the radix point separating them. However, the number presents in memory as just “10101”. Another way of describing the fixed-point number represented in memory is by the product $a \cdot 2^x$, where $a$ is our original fractional number and $2^x$ is known as a scaling factor with $x$ number of bits in the fractional part.

Implementing addition and subtraction for fixed-points is trivial. Given that we have two terms with $n$ bits that are in the same fixed-point format, the results of those operations will also be in the same format with $n + 1$ bits, where the extra bit is used in case of overflow. However, if the terms are small enough where no overflow is probable to occur, it suffices to use only $n$ bits to store the result.

Multiplication and division are slightly more difficult to implement. When multiplying two fixed-points numbers, each term’s scaling factor will also be multiplied. For instance, if the two terms have a scaling factor of $2^x$, then the product will have a scaling factor of $2^{2x}$. In order to represent the product in
the same format as the terms, we need to first store the result with at least \( n + x \) bits and then divide it with the scaling factor. Similarly, the resulting scaling factor after a division becomes 1 since the scaling factors of the dividend and the divisor cancel out each other. This can be solved by extending the number of bits in the dividend to \( n + x \) bits and then multiplying it by the scaling factor before the division.

However, these operations are not enough to implement SW-UCB and D-UCB. By looking at their algorithms, it becomes apparent that we need to implement the square root and natural logarithm with fixed-point arithmetic. There is a multitude of methods to implement them. Conveniently, there already exists a C library that provides fixed-point implementations for these algorithms, which is called libfixmath \[6\]. Therefore, we will describe the algorithms of these functions as they are realised in the library.

### 3.1.1 Square root

The idea of the following algorithm for calculating the square root is by calculating the root number digit-by-digit, going from the most significant digit to the least significant one. If \( N \) is a fixed-point number with \( n \) number of bits that we want to calculate the square root of, then we want to find the solution \( a \) such that \( a^2 \leq N \). We can write the solution as \( a = a_1 + a_2 + \ldots + a_n \), where \( a_m = c_m \cdot 2^{n-m} \) for each bit \( m \in [1, n] \) and \( c_m \in \{0, 1\} \) is the value of bit \( m \) in \( a \). We can then find an approximation \( r_n \) of \( a \), where \( r_m = r_{m-1} + a_m \) and \( r_0 = 0 \), by using the following inequality:

\[
\begin{align*}
r_m^2 &\leq N \iff (r_{m-1} + a_m)^2 \leq N \\
&\iff r_{m-1}^2 + 2r_{m-1} \cdot a_m + a_m^2 \leq N \\
&\iff 2r_{m-1} \cdot a_m + a_m^2 \leq N - r_{m-1}^2 \\
&\iff r_{m-1} \cdot 2a_m + a_m^2 \leq X_m
\end{align*}
\]

For this algorithm, we store each of the three terms \( r_{m-1} \cdot 2a_m, a_m^2 \) and \( X_m \) in the inequality into three variables. Then, we iterate over all the \( n \) bits from \( m = 1 \) to \( m = n \) by setting the greatest values of each bit so that the inequality holds, and update the variables at each iteration. Thus, we always have \( a_m = 2^{n-m} \) since we begin each iteration by assuming \( c_m = 1 \).

For each iteration, we see if the inequality holds. If it does, then we update \( X_m \) to \( X_{m+1} \) by doing:

\[
X_{m+1} = X_m - (r_{m-1} \cdot 2a_m + a_m^2)
\]

Then, we add the bit \( c_m = 1 \) to our approximation by updating \( r_{m-1} \) to \( r_m \), which is done by adding the term \( r_{m-1} \cdot 2a_m \) with \( 2a_m^2 \):

\[
r_{m-1} \cdot 2a_m + 2a_m^2 = r_{m-1} \cdot 2a_m + a_m \cdot 2a_m = (r_{m-1} + a_m) \cdot 2a_m = r_m \cdot 2a_m
\]
However, if the inequality does not hold, then we add the bit $c_m = 0$ to our approximation. In this case, we would make $a_m = 0$, which means that $X_{m+1} = X_m$ and $r_m = r_{m-1}$. Therefore, we need no additional operations to update the terms.

In order to iterate to the next bit, we update the term $r_m \cdot 2a_m$ to $r_m \cdot 2a_{m+1}$, which is done by dividing it by 2:

$$\frac{r_m \cdot 2a_m}{2} = \frac{r_m \cdot 2 \cdot c_m \cdot 2^{n-m}}{2} = r_m \cdot 2 \cdot c_{m+1} \cdot 2^{n-(m+1)} = r_m \cdot 2a_{m+1}$$

where $c_{m+1} = c_m = 1$ since we always start each iteration by having $c_m = 1$. Similarly, we update the term $a_{2m}$ to $a_{2m+1}$ by dividing it by 4.

During the last iteration of the algorithm, we will have the term $r_{n-1} \cdot 2a_n$. This means that $a_n = 1$ and therefore, $r_{n-1} \cdot 2a_{n-1} = r_n \cdot 2$. When the last bit has been added to the approximation, we have the term $r_n \cdot 2$. Finally, we retrieve the final approximation $r_n$ by dividing that term by 2.

### 3.1.2 Natural logarithm

Additionally, both SW-UCB and D-UCB make use of the natural logarithm. One way of implementing this is the use of Newton’s method. The problem can be expressed as finding the value $u$ such that:

$$u = \ln(v) \iff e^u = v \iff e^u - v = 0$$

Now, using Newton’s method, we find $u$ by setting $f(u) = e^u - v$. We then use the formula of Newton’s method based on some good initial guess $u_0$:

$$u_{k+1} = u_k - \frac{f(u_k)}{f'(u_k)}$$

where $f'(u) = e^u$.

Evidently, this implementation makes the use of the exponential function, which means that we additionally need a fixed-point implementation of it. A function that libfixmath also provides.

### 3.1.3 Exponential function

An algorithm for this is the use of the power series of the exponential function:

$$e^b = \sum_{i=1}^{\infty} \frac{b^i}{i!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + ...$$

The idea of this algorithm is that we store the aggregate sum of all the terms above by adding one term at a time in the series to the sum at each iteration of
the algorithm. Then, we can easily calculate the next term to add to the sum by multiplying the previous term added to the sum with $b_i$. Due to the limitation of memory, the next term will eventually be so small that it will be represented as 0 in memory. At that point, we have achieved a sufficiently good approximation of $e^b$. 

Chapter 4

Implementations

The implementation of SW-UCB and D-UCB operates in an environment with 16 channels, as specified in IEEE 802.15.4 [1].

In order to evaluate the impact that embedded implementations of SW-UCB and D-UCB have on their performances, two implementations of each of the algorithms have been done: one with floating-points and one with fixed-points. Both are implemented in the C programming language.

The floating-point implementation uses the doubles for representing the floating-point numbers in order to achieve as high of an accuracy as possible. The natural logarithm and square root function used are from the C standard library.

The fixed-point implementation uses 32-bit integers for representing the fixed-point numbers. We use the libfixmath library for getting the implementations of the square root, natural logarithm and exponential function operations as presented in the previous chapter [6]. Since the library implements these operators for 32-bit fixed-point numbers with 16 bits in the fractional part, our implementation uses 16 bits for the fractional part as well.

Even though SW-UCB also makes use of the natural logarithm, that function has only been implemented for D-UCB. Rather, since the logarithm in $c_t(\tau, a)$ takes an integer no greater than the window size as input, a lookup table is used instead where the input value directly maps to its corresponding logarithm output value in the fixed-point format. This makes it easier to implement as well as faster to run the algorithm.

Since SW-UCB uses a sliding window, it will eventually forget samples from channels that have been stored there for a long time. Eventually, if the PDRs of each vary greatly, we might end up with no samples of a channel stored in the window. This is an issue since if no samples of a channel are present in the sliding window, then $N_t(\tau, a) = 0$ for channel $a$, which means that we get division by zero when calculating $\bar{X}_t(\tau, a)$ and $c_t(\tau, a)$. In order to mitigate this problem, if we have a channel with no samples in the sliding window, we will automatically select it for transmission regardless of the calculated upper bounds of the other channels.
Moreover, because SW-UCB makes use of a sliding window that might toss samples over time, we need to store every sample in memory, which can be considerably memory consuming if the window is large enough. However, since D-UCB does not toss samples but rather make them ever less significant by multiplying them with the discount factor $\tau$, all samples can be stored in a single aggregate sum which gets added with a new sample and multiplied with $\tau$ after each packet transmission. Consequently, this makes D-UCB much more memory efficient than SW-UCB.
Chapter 5

Experiments

In this chapter, we present a description of the experiments that were conducted to evaluate the fixed-point implementations of SW-UCB and D-UCB. Then, we present the results of the experiments together with a discussion of them.

5.1 Experiments

In order to evaluate the effect of the fixed-point implementations of SW-UCB and D-UCB, we conducted a number of experiments. These were constructed by simulating 1000 packet transmissions for each of the floating-point and fixed-point implementation of SW-UCB and D-UCB with different values for the parameters $\tau$, $\gamma$ and $\xi$. Specifically, the values chosen for each parameter were $\tau \in \{100, 200, ..., 1000\}$, $\gamma \in \{0.950, 0.955, 0.960, ..., 1.0\}$ and $\xi \in \{1.0, 0.1, 0.01, 0.001, 0.0001\}$. They were conducted by simulating an environment randomizing the PDRs of each channel where one channel always had a max PDR of 1.0 while the average PDR of all channels was 0.5. They were divided into a stationary part, where the channel PDRs remained constant over the experiment, and a non-stationary part, where in one case there was one breakpoint at time step $t = 500$ and another case with 4 break points evenly spaced at every 200th packet transmission. We repeated all cases a hundred times each, with different PDR values generated between each one of them, where we present the results as the mean PDR of all the repetitions.

Additionally, since our fixed-point numbers uses 16 bits for the fractional part, they cannot represent fractions smaller than $2^{-16} \approx 0.00001526$. However, there might exist smaller values of $\xi$ that give higher PDRs for our algorithms. Therefore, we used the floating-point implementations of SW-UCB and D-UCB in order to tune and possibly find optimal values of $\xi$ that cannot be represented with our fixed-point numbers.
5.2 Results

In this section, we present the results of the experiments. The results for SW-UCB can be seen in Figures 5.1-5.3 while the results of D-UCB can be viewed in Figures 5.4-5.6. Then, the parameter settings yielding the optimal PDRs of each implementation of SW-UCB and D-UCB is presented in Table 5.1. Lastly, we present the best floating-point parameters for D-UCB in Table 5.2.

5.2.1 SW-UCB

For SW-UCB, we see that in every graph of figures 5.1-5.3, the performances of the fixed-point and the floating-point implementations are largely identical.

For the stationary environment in Figure 5.1, larger values for of the sliding window seem to give better performances. This is expected, since having a larger window means that less time is needed to explore channels whose samples in the window have been tossed. Interestingly, it seems that for smaller $\xi$, the performances for every setting of window sizes start to increase. This might be the case due to the algorithm quickly being able to find the optimal channel and therefore a smaller amount of time is required to explore bad channels when $\xi$ is lower.

In Figure 5.2, we see the performances of SW-UCB in an environment with 1 breakpoint. Here, for higher values of $\xi$, the algorithm with lower values of $\tau$ perform worse than for higher values. However, when $\xi$ is lower, the lower values of $\tau$ start to perform much better with $\tau = 500$ having the best performance overall. This is probably due to lower values of $\tau$ have to spend time relearning channels once the sliding window has reached its limit. However, when lowering $\xi$, the impact of the exploration part $c_t(\tau, a)$ is also lowered, which means that SW-UCB will be less inclined to explore potentially bad channels for too long. Similar results can be seen in Figure 5.3, where we ran SW-UCB in an environment with 4 breakpoints. Again, for $\xi = 1.0$, lower values of $\tau$ perform worse while for low values of $\xi$, $\tau = 200$ has the best performance.
Figure 5.1: Performances of SW-UCB in the stationary environment with different settings of $\xi$ in each graph and $\tau$ values in the x-axis.
Figure 5.2: Performances of SW-UCB in the non-stationary environment with 1 breakpoint with different settings of $\xi$ in each graph and $\tau$ values in the x-axis.
5.2.2 D-UCB

For D-UCB, we also see, like with SW-UCB, that the fixed-point and floating-point implementations perform mostly identical.

In the stationary case in Figure 5.4, we see that for lower values of $\xi$, the algorithm seems to perform much better overall for every value of the discount factor $\gamma$. This is again probably due to D-UCB becoming less inclined to explore bad channels when $\xi$ is low, thus boosting the performance.

However, looking at Figure 5.5 where we have 1 breakpoint in the middle, the optimal value of $\gamma$ starts to shift to lower values the lower $\xi$ becomes. This can also be seen in Figure 5.6, where we have an environment with 4 breakpoints. In fact, higher values of $\gamma$ seem to perform worse with lower values of $\xi$. This might be due to that for higher $\gamma$, old samples from channels will take a longer time to diminish, which means that older samples of a previous optimal channel will still have a great impact even after a breakpoint. Coupled with a lower value of $\xi$, D-UCB will also be less inclined to explore other channels that might have a better performance, thus making the overall performance of D-UCB drop. However, with lower values of $\gamma$, old samples of channels will diminish more quickly, thus making it more likely to choose the current optimal channel.
Figure 5.4: Performances of D-UCB in the stationary environment with different settings of $\xi$ in each graph and $\gamma$ values in the x-axis.
Figure 5.5: Performances of D-UCB in the non-stationary environment with 1 breakpoint with different settings of $\xi$ in each graph and $\gamma$ values in the x-axis.
5.2.3 Optimal Parameter Settings

In Tables 5.2a-5.2d, we see for each implementation the highest resulting PDR for each environment and the parameter settings that yielded them. When comparing each algorithm’s floating-point and fixed-point implementations, we see that their PDR results are very similar to each other, as well as the parameter settings yielding them. Furthermore, for SW-UCB, we see that when an environment has more breakpoints, a higher value of $\xi$ and lower values of $\tau$ and $\gamma$ are needed to give the best PDRs. This can also be seen for D-UCB with its discount factor $\tau$. However, the optimal value of $\xi$ remained for every environment for D-UCB.

Additionally, when comparing the PDRs of SW-UCB and D-UCB, we see that they have almost identical results in the stationary environment. This is expected, since both D-UCB and SW-UCB essentially behave like UCB1 when $\gamma = 1.0$, which means that every sample is stored with equal significance in the history, and the sliding window can store all samples without tossing anyone. However, for the other environments, SW-UCB performs slightly better than D-UCB, although the differences are slim. This is also expected since SW-UCB has slightly outperformed D-UCB in other experiments between them [8].

Moreover, in Table 5.2, we see the parameter setting that yielded the greatest
<table>
<thead>
<tr>
<th>Environment</th>
<th>$\xi$</th>
<th>$\tau$</th>
<th>PDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stationary</td>
<td>0.0001</td>
<td>1000</td>
<td>0.9850</td>
</tr>
<tr>
<td>1 Breakpoint</td>
<td>0.001</td>
<td>500</td>
<td>0.9690</td>
</tr>
<tr>
<td>4 Breakpoints</td>
<td>0.01</td>
<td>200</td>
<td>0.9122</td>
</tr>
</tbody>
</table>

(a) SW-UCB with floating-point arithmetic.

<table>
<thead>
<tr>
<th>Environment</th>
<th>$\xi$</th>
<th>$\tau$</th>
<th>PDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stationary</td>
<td>0.0001</td>
<td>1000</td>
<td>0.9851</td>
</tr>
<tr>
<td>1 Breakpoint</td>
<td>0.001</td>
<td>500</td>
<td>0.9679</td>
</tr>
<tr>
<td>4 Breakpoints</td>
<td>0.01</td>
<td>200</td>
<td>0.9125</td>
</tr>
</tbody>
</table>

(b) SW-UCB with fixed-point arithmetic.

<table>
<thead>
<tr>
<th>Environment</th>
<th>$\xi$</th>
<th>$\gamma$</th>
<th>PDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stationary</td>
<td>0.0001</td>
<td>1.0</td>
<td>0.9850</td>
</tr>
<tr>
<td>1 Breakpoint</td>
<td>0.0001</td>
<td>0.990</td>
<td>0.9538</td>
</tr>
<tr>
<td>4 Breakpoints</td>
<td>0.0001</td>
<td>0.970</td>
<td>0.9107</td>
</tr>
</tbody>
</table>

(c) D-UCB with floating-point arithmetic.

<table>
<thead>
<tr>
<th>Environment</th>
<th>$\xi$</th>
<th>$\gamma$</th>
<th>PDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stationary</td>
<td>0.0001</td>
<td>1.0</td>
<td>0.9850</td>
</tr>
<tr>
<td>1 Breakpoint</td>
<td>0.0001</td>
<td>0.990</td>
<td>0.9565</td>
</tr>
<tr>
<td>4 Breakpoints</td>
<td>0.0001</td>
<td>0.970</td>
<td>0.9095</td>
</tr>
</tbody>
</table>

(d) D-UCB with fixed-point arithmetic.

Table 5.1: Parameter settings of each implementation that yielded the highest PDR for each environment.

PDRs when tuning $\xi$ for the floating-point D-UCB with lower values than our fixed-point numbers allow. The floating-point settings for SW-UCB are not shown in Table 5.2 since the settings in Table 5.1b ended up yielding the best PDRs for it. However, we see in Table 5.2 that there was an increase in PDR for D-UCB with very low values of $\xi$. The most prominent difference were for the 1 breakpoint environment, while there was only a very slight difference for the stationary environment and a small increase for the 4 breakpoint environment when comparing with the results in Table 5.1c. Consequently, these PDRs are now very close to the optimal PDRs of the floating-point implementation of SW-UCB. We can also see that the optimal value of $\xi$ for floating-point D-UCB is higher in the 4 breakpoint environment than the 1 breakpoint environment, which is expected since the algorithm needs to put emphasis on exploration when there are more breakpoints.
Table 5.2: Parameter settings for D-UCB with floating-point arithmetic that yielded the highest PDR for each environment when tuning $\xi$ with lower values than our fixed-point numbers allow.

<table>
<thead>
<tr>
<th>Environment</th>
<th>$\xi$</th>
<th>$\gamma$</th>
<th>PDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stationary</td>
<td>$10^{-8}$</td>
<td>1.0</td>
<td>0.9851</td>
</tr>
<tr>
<td>1 Breakpoint</td>
<td>$10^{-10}$</td>
<td>0.960</td>
<td>0.9685</td>
</tr>
<tr>
<td>4 Breakpoints</td>
<td>$10^{-8}$</td>
<td>0.925</td>
<td>0.9171</td>
</tr>
</tbody>
</table>

5.3 Discussion

Based on the aforementioned results, we see that the differences between the fixed-point and floating-point implementations of both SW-UCB and D-UCB are almost non-existent. This seems to indicate that fixed-point implementations with 32-bit integers with a 16-bit fractional part are a feasible replacement for floating-point arithmetic in terms of accuracy of the algorithms. In fact, the differences in performances seem to be much less do to with the differences in implementations and more to do so with specific settings of the parameters.

Moreover, SW-UCB slightly outperformed D-UCB in the non-stationary environments. However, the differences were considerably minor. This means that D-UCB could be usable as an alternative to SW-UCB since it uses less memory than SW-UCB, which can be important as embedded devices have limited memory. Additionally, when optimally tuning $\xi$ and $\gamma$ for the floating-point implementation of D-UCB, the differences in PDRs between SW-UCB and D-UCB became even lower. However, these values of $\xi$ were tuned to far lower values than our fixed-point numbers can represent, with the lowest optimal value being $10^{-10}$ for the 1 breakpoint environment. In order to represent that fraction as a fixed-point number, we would need to use $\log_2\left(\frac{1}{10^{-10}}\right) \approx 33.2 < 34$ bits in the fractional part, which would mean that we need to use 64-bit fixed-point numbers. This is problematic due to the limited memory of embedded devices. One possible mitigation to this problem could be to scale up all the values that we use by some constant. This would make it possible to represent very small values of $\xi$ with our 32-bit fixed-point numbers. However, one concern with this solution is that other values might scale up to become so large that they cannot be represented using our 32-bit fixed-point numbers.

However, one limitation of these results are that they are a result of simulations of wireless channels, which means that they do not depict very accurately the behaviours of real wireless channels. In order to get more accurate results, these implementations can be tested in experiments based on real data of wireless channels or integrate these implementations into actual hardware. However, based on the results in this thesis, the performances of the floating-point implementations and fixed-point implementations of SW-UCB and D-UCB would probably not differ anything significantly.
Chapter 6

Conclusions and Future Work

In this thesis, we have implemented embedded implementations of SW-UCB and D-UCB in the C programming language with fixed-point arithmetic as an alternative to floating-point arithmetic for embedded microprocessors that lack a floating-point unit (FPU). The performances of the algorithms have been measured in terms of packet delivery ratio (PDR) by running them in simulated stationary and non-stationary environments with different parameter settings, where they were compared with corresponding floating-point implementations of these algorithms. The results of the experiments were very promising, since no significant differences have been between the results of the floating-point and fixed-point implementations. Instead, the results depend more on the proper parameter settings, where the optimal settings differed depending on the environment.

As mentioned previously, a limitation of these results are that they were obtained from simulations. More accurate results could be achieved by using real data to set up experiments or by integrating these implementations into hardware.

These experiments only describe the scenario of one transmitter node using the channels for transmissions. It would be interesting to see the effects of having multiple transmitter nodes operating on the same channels.

One could also try to implement the other multi-armed bandit algorithm, such as TS, STS and STSBA, using fixed-point arithmetic and compare them with SW-UCB and D-UCB.

Since PDR was the only metric used in this thesis for measuring the performances of the implementations, other metrics such as memory consumption and execution time might also be of interest of studying. For instance, since 32-bit integers with 16-bits in the fractional part gave a very accurate result compared with the floating-point implementation and since embedded microprocessors have a low amount of memory, one could also try to make the implementation more memory efficient by changing the size of the fixed-point integers or by changing the size of the fractional part. Moreover, one might also try to find faster or less memory consuming fixed-point mathematical functions than those provided by
the libfixmath library.
Bibliography


