AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

INSTYTUT TELEKOMUNIKACJI

Praca dyplomowa magisterska

Implementation and Analysis of a Monte Carlo Simulator Extension for ns-3

Implementacja i analiza rozszerzenia symulatora ns-3 o moduł umożliwiający przeprowadzenie symulacji typu Monte Carlo

Autor:	Kamil Chełminiak
Kierunek studiów:	Teleinformatyka
Opiekun pracy:	dr hab. inż. Szymon Szott, prof AGH

Kraków, 2023

Contents

1 Introduction

With the increase of computation power in the last couple of years, the potential of conducting Monte Carlo simulations has risen as well. This powerful tool is extremely helpful in the analysis of a large amount of data in repetitive scenarios. On the other hand there are powerful network simulators, such as *ns-3* which developers seem to either ignore or not be aware of the potential of these methods. With the wide range of possible use cases, there is a strong need to not neglect a particular one, which influences the everyday life of everyone – networking. Due to the increase in the number of users in mobile networks the need to introduce e.g. new, adaptive control mechanisms rises with every day. These factors were the biggest reasons for the implementation of the solution, presented in this thesis.

The main goal of this thesis is to implement and validate a library capable of conducting Monte Carlo simulations in the *ns-3* simulator. The implemented MonteCarloSimulator library is intended to provide a user with built–in options for both performance testing and decision–making based on the per–flow throughput measured separately for each flow active in the network. Additionally, as the throughput is not the only possible measurement option, users should be able to implement their own statistics collectors and models. The important part of this implementation is to keep the most crucial part of the simulations – easy access to the meaningful output data which can be analysed from both the console and the output file, aggregating results from every round.

This thesis is structured as follows. Chapter 2 describes the main principles of Monte Carlo simulation methods and a general concept of every simulation which could be identified as a Monte Carlo simulation. At the end of this chapter is included a brief summary of the most recent papers focused on the networking area of usage of the analysed methods. Chapter 3 describes the basic concepts of the implemented library. It also describes the most important implemented methods which are separate parts of the implemented solution. Chapter 4 introduces three algorithms which are used in the verification of the implemented library. Although they are not directly connected to the main topic of this thesis, the algorithm–wise analysis of obtained results is used to verify correctness of the implemented simulator. Finally Chapter 5 concludes the work, including drawing the conclusions from the obtained results and pointing out possible scenarios for the future work.

2 Background

The history of Monte Carlo methods began in XVIII century when George Louis LeClerc performed an experiment which many believe was the first Monte Carlo simulation – "Buffon's Needle" during which he estimated the value of π by throwing baguettes over his shoulder onto a tiled floor. However comical, this simple experiment was the beginning of a new field of study, named after a casino in Monaco [1], which modernised version was used in the creation of the first atomic bomb during World War II. The World War II examinations were also a breaking point after which a modern approach for these simulations was popularised [2].

2.1 Monte Carlo simulations

Monte Carlo simulations are based on the randomness of the analysed data and statistical analysis, which is a key factor in the decision–making process. Due to these assumptions it is important to take care of the quality of the input data, as Monte Carlo methods are highly susceptible to generate unrealistic outputs based on the biased input data. The simplest way to avoid this situation is to analyse the most interesting scenarios case by case, but this approach might not be efficient as the number of the simulations needed to be performed in order to cover all scenarios grows exponentially with the number of input parameters [3].

The most popular probability destributions used to generate data for the purposes of the Monte Carlo simulations are [4]:

- normal distribution – a distribution described by two parameters m and δ, which characterise the central point and the width of the density function which representation is a bell–shaped curve [5];
- uniform distribution – a distribution closely related to generation of random numbers in simulation processes as each value has the same probability of being generated [6];
- triangular distribution – a three-point distribution based on the lower and upper data limits and mode where the highest value of the probability density function is reached in the mode point [7].

As shown in Figure 1, Monte Carlo methods can be generally divided into two categories [1]:

- Sequential Monte Carlo – the examples are sequentially sampled and their importance is reweighted during the subsequent rounds during which the data is stored for the purposes of future analysis;
- Markov chain Monte Carlo – the exploration of the state space is performed with the use of a Markov chain and the stationary probability of the occurrence of the event converges to a given target probability.

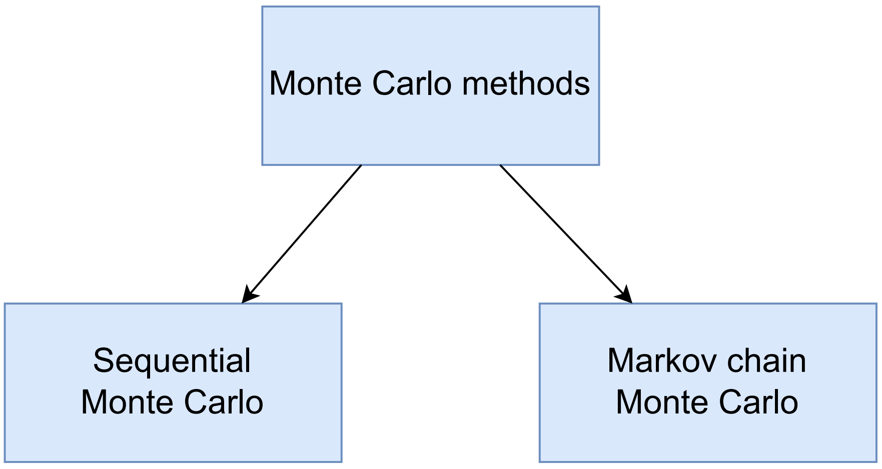

Figure 1: General division of Monte Carlo methods

There are also three paradigms from which a architect of the Monte Carlo solution needs to choose in order to implement the desired simulation. This choice should be based on the expected results and should prioritise either model accuracy or computational complexity. This division of paradigms is shown in Figure 2. In general the following paradigms can be distinguished [1]:

- Approximate model with exact computing in which the algorithm used in the simulations is simplified and other coding practices like dynamic coding help in finding the exact solution to the approximated problem;
- Exact model with local computing in which the initial states are guided by a heuristic and the local solution algorithm is simplified (e.g. gradient descent is used);
- Exact model with asymptotic global computing which uses the large amount of samples which after simulations converge to an optimal solution with high probability.

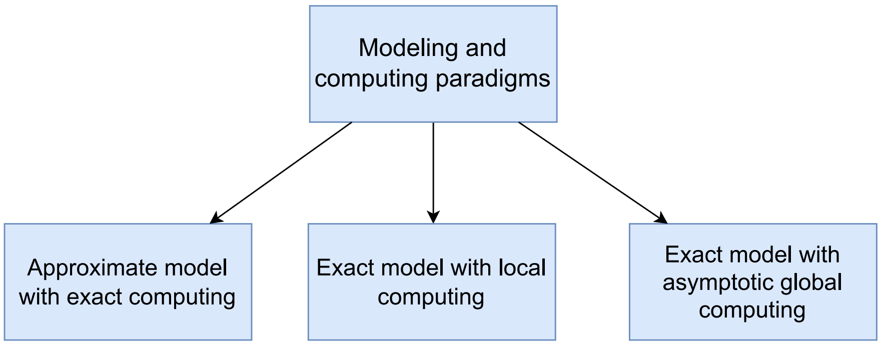
Modeling and computing paradigms
Approximate model with exact computing
Exact model with local computing
Exact model with asymptotic global computing

Figure 2: General division of paradigms used during the creation of Monte Carlo solution

Implemented MonteCarloSimulator library contributes to Exact Sequential Monte Carlo model with asymptotic global computing category as in network scenarios a high amount of simulations is performed and the solution should converge to an optimal solution.

A Monte Carlo simulation consists of three components [4]. The first one are input variables. Due to the vulnerability to biased data, choosing the right data generation method might be the most important step in the whole process. Even with the best design of the examined algorithm the results of the simulation might be incorrect, because of the poor choice of input data generation method. Due to the decision–making process throughout the whole simulation, the input parameters need access to the historic data [3].

The second component of the simulation is the examined model. As R. L. Harrison wrote in [2]: "Key points to consider in defining a model are:

- what are our desired outputs?
- what will these outputs be used for?
- how accurate/precise must the outputs be?
- how exactly can/must we model?
- how exactly can/must we define the inputs?
- how do we model the underlying processes?"

These question are also helpful when deciding about the architecture of implementation of a new Monte Carlo library. But the most crucial thing is that when answering these questions, one can identify an exact model which will be helpful in an examined condition, as there are no universal ones and each solution should be tailored to match the expectations.

The last component is output data. This data is specific in such way that the user is not expected to prepare them in order to run the simulation. Instead, the user is expected to analyse them and draw conclusions about the quality of the input data and examined algorithm. During the data analysis it is expected that the user analyses trends in the results and based on them eliminates biased data or errors in the logic of the examined model.

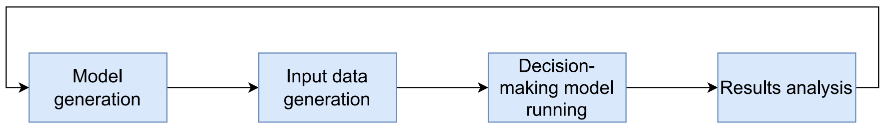

Figure 3: General course of Monte Carlo simulation

The remaining question after the three main components are identified is how is a generalised Monte Carlo simulation performed? The Monte Carlo simulation cycle, presented in Figure 3 consists of four major steps. The first one is a definition of decision–making model. This step should be performed as the first one, because the second one (data generation) is highly–reliant on the required data type and format. After the model is declared, the user can choose the right probability distribution and generate the input data. Third step combines the outputs of the previous two, as it is a moment when input data is transformed with the defined model into output data. This result data is then analysed in the last step, as the most important advantage of this type of simulations is simplicity in the output data analysis. Once the output data has been analysed, the whole process should be repeated until the required quality of the model is reached [3]. The same three components of the simulation and similar process of data processing is also present in Machine Learning. As ML and is currently a trending topic it is important to notice the difference between

Monte Carlo methods and ML. Monte Carlo simulation uses the input data to produce output data with the use of the examined model. On the other hand Machine Learning analyses input and output data and based on them tries to find coefficients which can transform input data in the output data [4]. This difference is depicted in Figure 4.

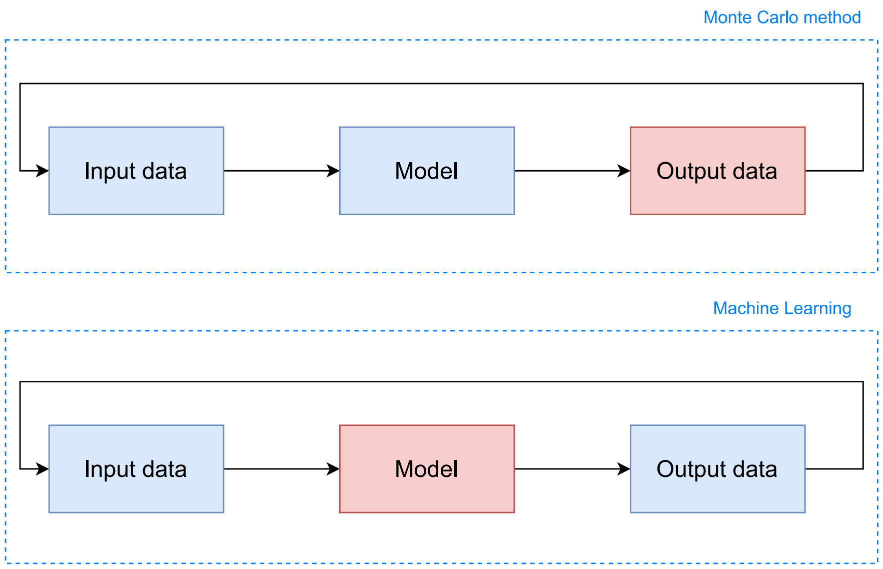

Figure 4: The difference between Monte Carlo method and Machine Learning [4]

There is a number of advantages of Monte Carlo simulations. Among them the most important ones are [8]:

- the improved decision–making as the Monte Carlo simulations are able to provide a high number of possible outputs which positively influence the ability to make decisions;
- simplify solving of the complex problems as with the high number of possible outputs it is clear what could happen in analysed scenario and how likely this outcome is;
- easy visualisation of the possible outcomes and their probability as the output data in most cases is a measurable and obvious outcome, which does not need further processing.

Naturally, as the perfect solution does not exist, Monte Carlo simulations have some drawbacks, which need to be considered when choosing the right analysis method for the scenario examination. These drawbacks are [8]:

- high processing power to process high volume of input data during many repeated examinations which needs to processed in a reasonable and finite time;
- input bias which was mentioned before, but plays such a significant role, that it cannot be omitted when discussing drawbacks of Monte Carlo approach;
- vulnerability to chosen probability distribution as not only biased but incorrectly generated data might result in production of meaningless results.

2.2 State of the Art

Monte Carlo simulations have a wide range of applications. It is not surprising that they are used in such industries as [8]:

- finances for the long–term forecasting and risk analysis;
- project management for risk analysis and baseline and costs prediction;
- engineering and physics for pattern analysis, traffic flow or energy distribution;
- quality control and testing for reliability and failure rate analysis;
- healthcare and biomedicine for modeling spread of diseases and in radiotherapy.

From the perspective of this thesis the most important field of study is computer science. In this area Monte Carlo simulations are applicable in the following use cases [8]:

- cybersecurity [9], [10], [11], [12], [13];
- machine learning [14], [15], [16], [17], [18];
- network and system design [19], [20], [21], [22], [23].

As this thesis is focused on the implementation of Monte Carlo library in the *ns-3* simulator, the most important aspect of the conducted studies is the “network and system design” from the computer science field of study. The authors of [19] use the Monte Carlo method to model and analyse flows in a network from both system and customer point of view. In this paper the results are drawn based on the resource utilisation and the performance of the network which is measured based on the length of the queue and waiting time for the connection.

The second analysed paper [20] focuses more on the prevention of the selfish behaviour in the wireless system. As the IEEE 802.11 standard does not provide any means of fair sharing of a common channel execution, the authors propose an additional mechanism, called “DISTRESS”. This mechanism is a safety fuse, which is used when one or more users try to take over the channel by setting the highest possible priority of the transmitted frames. The conducted simulations use an approach when the situation in the network is analysed for specified number of times or until all stations choose to not behave in a selfish way.

The authors in [21] focused on three significant wireless technologies – Wi–Fi, LTE and 5G. As it is noticed, with the development of high–speed mobile networks, the frequencies in the unlicensed band have to be shared. The studies focus on comparison of three different approaches in network modelling, but all are based on Monte Carlo simulations, as the authors need to analyse large set of data because of the scale of the analysed network.

Paper [22] is a combination of both access control and cybersecurity, as the Network Access Controller algorithm does not only grant access to a network, but also encrypts the traffic data. As the authors admit Monte Carlo simulations are the perfect tool to analyse the decision–making process of the proposed algorithm. The reason behind this decision is a fact that the output data is easily analysable and does not need require an additional heavy processing.

The last paper [23] from the networking category is focused on CSMA Wireless Networks. The examined network parameter – signal–to–interference–and–noise–ratio (SINR) – is crucial from both the perspective of user and the network. When this parameter is too low user might not be able to connect to the network, if he is too far from the Access Point. On the other hand if the interferences are too high the Access Point might not have a full view of the network and might not be able to provide a sufficient level of service for its users.

Although all the articles regard the different types of network and are based on the different networking problem, they have a few common points. Firstly they all can and are realised with the help of *ns-3* simulator. Secondly they all use the round–based mechanism to gather the output data. Thirdly they all need to process large amount of input data. Because of the above reasons, the demand for the *ns-3* library which could ease the process of Monte Carlo simulation setup was high and this is the main reason behind the creation of this thesis.

3 Implementation

The aim for the implementation of a framework, which can be used for Monte Carlo simulations was to shorten length of a written code. Framework should be as versatile as possible, but still useful. Keeping that in mind, MonteCarloSimulator framework has a few built–in options, which are the most commonly used in examined scenarios. Apart from them, user must have total control of the whole course of simulations and should be able to customize both actions, which are taken by actors (nodes) and measurement of the results. All the listings, presented in this chapter below, are separate parts of the whole solution, which high–level idea is presented in Figure 5.

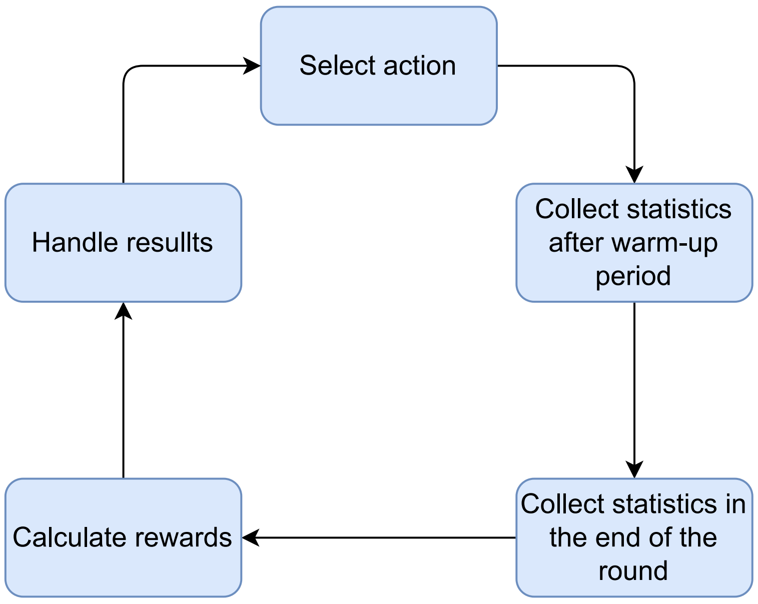
Select action
Collect statistics after warm-up period
Collect statistics in the end of the round
Calculate rewards
Handle resullts

Figure 5: Architecture of the MonteCarloSimulator library

```
MonteCarloSimulator::MonteCarloSimulator(ApplicationContainer*
                                             sinkApplications,
                                         double numberOfRounds,
                                         double roundTime,
                                         double roundWarmup,
                                         std::string outputName,
                                         uint32_t resultsPrinting,
                                         bool useDefaultRewardCalculation,
                                         std::function<void()>
                                             BehaviourFunction)
{
    sinks = sinkApplications;
    rounds = numberOfRounds;
    time = roundTime;
    warmup = roundWarmup;
    outputFileName = outputName + ".csv";
    printing = resultsPrinting;
    useDefaultCalculation = useDefaultRewardCalculation;
    if (useDefaultRewardCalculation)
    {
        Simulator::Schedule(Seconds(warmup),
                            &MonteCarloSimulator::GetWarmupStatistics,
                            this);
    }
    for (int round = 0; round < rounds + 1; ++round)
    {
        if (useDefaultRewardCalculation){
            Simulator::Schedule(Seconds((round + 1) * time),
                                &MonteCarloSimulator::
                                    DefaultRewardCalculation,
                                this);
            Simulator::Schedule(Seconds((round + 1) * time + warmup),
                                &MonteCarloSimulator::GetWarmupStatistics,
                                this);
            Simulator::Schedule(Seconds((round + 1) * time),
                                &MonteCarloSimulator::HandleResults,
                                this);
        }
        Simulator::Schedule(Seconds((round + 1) * time),
                            BehaviourFunction);
    }
    for (int i = 0; i < 200; ++i)
    {
        for (int j = 0; j < 500; ++j)
        {
            throughputArray[i][j] = 0;
            throughputSumArray[i] = 0;
            totalBytes[i] = 0;
            chooseArray[i] = 0;
            rewardArray[i][j] = 0;
        }
    }
}
```

Listing 1: MonteCarloSimulator constructor

The constructor of the MonteCarloSimulator library, shown in Listing 1, has eight mandatory parameters:

- sinkApplications – pointer to a *ApplicationContainer* with *PacketSinks*; by default rewards (average throughput) are calculated, based on the amount of the received bytes by each container;
- numberOfRounds – number of rounds, after which the simulation is stopped;
- roundTime – time of each round; this value should include warm–up period;
- roundWarmup – time of the warm–up period; all the events, that happened during warm–up time do not influence the calculation of the rewards;
- outputName – name for an output file in which the rewards are stored;
- resultsPrinting – number of the round from which the calculated rewards are shown in the output console;
- useDefaultRewardCalculation – *boolean* value indicating whether the default warm–up statistics collector and reward calculation method should be used;
- BehaviourFunction – set of actions/instructions, which the actors are performing during a single round.

As it was mentioned in the description of *sinkApplications* input parameter, the default reward for each stream is the average throughput obtained during a round. The description of this reward calculation method can be found under Listing 3. However, thanks to *useDefaultRewardCalculation*, the user can define his own reward calculation method and use in his research e.g. mean packet delay. Description of how a user can use his own reward calculation method can be found under Listing 5.

Apart from setting appropriate variables and initializing values of all arrays, used during reward calculation, MonteCarloSimulator constructor is an orchestrator for both default reward calculation and behaviour of actors during the concurrent rounds. Important thing to notice in Listing 1 is that a round zero of the simulation was introduced, as all the methods are not scheduled from 0.0 seconds, but rather from *roundTime* seconds (apart from warm–up statistics collector, as the results of this round are handled like results of any other round). This gives a user opportunity to define all the prerequisites outside of the *BehaviourFunction*. If for example there is a prerequisite, that requires certain association in the initial state of the network, this configuration can be done outside of the MonteCarloSimulator object.

An important thing to notice in the definition of the MonteCarlo constructor is its limitation. The library supports up to 200 different flows and up to 500 rounds. This limitation was introduced to allocate enough memory for the purposes of calculation and storage of the rewards.

```
void
MonteCarloSimulator::GetWarmupStatistics()
{
    for (uint32_t applicationIndex = 0; applicationIndex < sinks->GetN();
         ++applicationIndex)
    {
        totalBytes[applicationIndex] =
            DynamicCast<PacketSink>(sinks->
                                        Get(applicationIndex))->GetTotalRx();
    }
}
```

Listing 2: Statistics collection after *warmupTime* in each round

Listing 2 contains the definition of default warm–up statistics collector. Figure 6 depicts a start of an exemplary simulation, performed with the use of MonteCarloSimulator library and the placement in time of a warm–up period and round duration (*roundTime*). The reason for introducing warm–up period appeared during the implementation of algorithms, used in Chapter 4. When a network scenario, in which each station needs to change its association to Access Point in each round, especially if TCP protocol is used, is being implemented, all actors need to instantiate a stable connection to their destination. If a user was to examine the results, they might be understated, because of this period. Meeting this need, MonteCarloSimulator allows the user to specify the length of warm–up period, starting from a round zero. This function is not used if *useDefaultRewardCalculation* in constructor from Listing 1 is set to *false*.

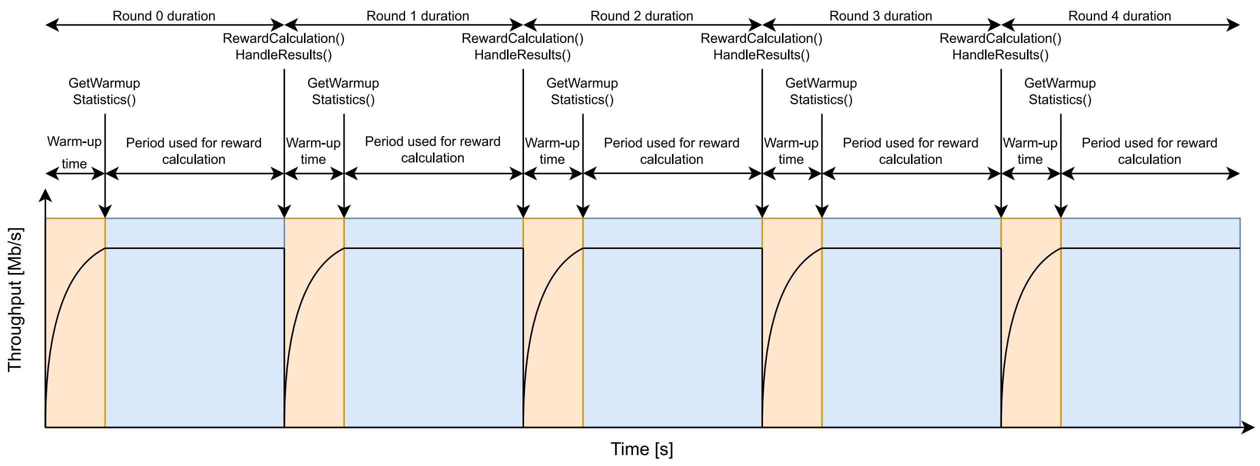

Figure 6: Warm–up time and round time in subsequent rounds

The warm–up statistics collector stores the amount of received bytes for each of the data stream in *totalBytes* vector. The position of the statistic in the vector is the same as the number of the examined *PacketSink*. This stored data is later used during the per–flow throughput calculation, where it is subtracted from amount of bytes received during the whole round, giving the user information about the throughput offered in time period from *roundWarmup* seconds, to *roundTime* seconds.

```
void
MonteCarloSimulator::DefaultRewardCalculation()
{
    uint64_t totalBytesThroughput[sinks->GetN()];
    for (uint32_t applicationIndex = 0; applicationIndex < sinks->GetN();
         ++applicationIndex)
    {
        totalBytesThroughput[applicationIndex] =
            DynamicCast<PacketSink>(sinks
                                            ->Get(applicationIndex))
                ->GetTotalRx();
        throughputArray[applicationIndex][currentRound] =
            (totalBytesThroughput[applicationIndex] -
             totalBytes[applicationIndex]) * 8 /
            ((time - warmup) * 1000000.0);
        totalBytes[applicationIndex] =
            totalBytesThroughput[applicationIndex];
    }
    for (uint32_t applicationIndex = 0; applicationIndex < sinks->GetN();
         ++applicationIndex)
    {
        if (throughputArray[applicationIndex][currentRound] > 0)
        {
            chooseArray[applicationIndex] += 1;
            throughputSumArray[applicationIndex] +=
                throughputArray[applicationIndex][currentRound];
            rewardArray[applicationIndex][currentRound] =
                throughputSumArray[applicationIndex] /
                chooseArray[applicationIndex];
        }
    }
}
```

Listing 3: Default reward calculation

Similarly to warm–up statistics collector from Listing 2, DefaultRewardCalculation() function from Listing 3 is used only when *useDefaultRewardCalculation* is set to *true* during the MonteCarloSimulator object definition. It is assumed that the default metric, which is measured, is throughput obtained for each data stream, represented by the amount of received bytes in *ns-3* as *PacketSink*. However, since throughput throughout consecutive rounds can have a high variation, the value of throughput might not be a good information about the state of the network. Keeping that in mind, the default reward for each data stream is the average throughput, obtained during the rounds in which the stream was active. For example, if the simulator is currently in round 15 and the examined stream had a 20 Mb/s throughput during 10 rounds and 0 Mb/s during the others, its reward will be set to 20.

```cpp
void
MonteCarloSimulator::HandleResults()
{
    if (currentRound >= printing)
    {
        std::cout << "Results for round " << currentRound << ": "
                  << std::endl;
        for (uint32_t applicationIndex = 0;
             applicationIndex < sinks->GetN();
             ++applicationIndex)
        {
            std::cout << "Reward for application number "
                      << applicationIndex << ": "
                      << rewardArray[applicationIndex][currentRound]
                      << std::endl;
        }
    }
    std::ofstream outputFile;

    std::string outputCsv = outputFileName;
    if (FileExists(outputCsv))
    {
        // If the file exists, append to it
        outputFile.open(outputCsv, std::ios::app);
    }
    else
    {
        // If the file does not exist, create it and set the header line
        outputFile.open(outputCsv, std::ios::app);
        outputFile << "StageNumber";
        for (uint32_t applicationIndex = 0; applicationIndex <
                                             sinks->GetN();
             ++applicationIndex)
        {
            outputFile << ",Reward" << applicationIndex;
        }
        outputFile << std::endl;
    }

    outputFile << currentRound;
    for (uint32_t applicationIndex = 0;
         applicationIndex < sinks->GetN();
         ++applicationIndex)
    {
        outputFile << "," << rewardArray[applicationIndex][currentRound];
    }
    outputFile << std::endl;

    outputFile.close();
    currentRound += 1;
}
```

Listing 4: Results handler

The HandleResults() method, presented in Listing 4, is the only built–in function, which will always be executed during the usage of MonteCarloSimulator. The library by default produces an output comma–separated file, named *outputName.csv*, in which all rewards are stored. It is worth noticing, that the round zero rewards are also included in this output file. The MonteCarloSimulator will create a new file with an appropriate name or add new lines to the bottom of an existing one.

The Results handler introduces an optional functionality, which is printing the results of a round in console. This option is controlled by the *printing* variable, which is set by the constructor from Listing 1. By default the library will print results in the console, starting from round zero. If a user specifies another value, the first round which results will be printed is the round numbered *printing*.

```
void
MonteCarloSimulator::SetRewardCalculationFunction
    (std::function<void()> RewardCalculationFunction)
{
    if (!useDefaultCalculation)
    {
        for (int round = 0; round < rounds + 1; ++round)
        {
            Simulator::Schedule(Seconds((round + 1) * time),
                                RewardCalculationFunction);
            Simulator::Schedule(Seconds((round + 1) * time),
                                &MonteCarloSimulator::HandleResults,
                                this);
        }
    }
}

void
MonteCarloSimulator::SetWarmupStatisticsCollectionFunction(
    std::function<void()> WarmupStatisticsFunction)
{
    if (!useDefaultCalculation)
    {
        for (int round = 0; round < rounds + 2; ++round)
        {
            Simulator::Schedule(Seconds((round)*time + warmup),
                                WarmupStatisticsFunction);
        }
    }
}

void
MonteCarloSimulator::SetEndConditionFunction
    (std::function<bool()> EndConditionFunction)
{
    for (int round = 0; round < rounds + 1; ++round)
    {
        Simulator::Schedule(Seconds((round + 1) * time),
                            [EndConditionFunction]()
                            {if (EndConditionFunction()){
                Simulator::Stop();
            }});
    }
}
```

Listing 5: Set functions

Listing 5 contains three different customization options. With use of these functions, a user can customize the statistics collection process. When using these customisations, one should call the SetRewardCalculationFunction() as the penultimate (the last one if the SetEndConditionFuntion() is not used), because in its definition, the *Scheduler* function is invoked, which schedules results handling, introduced in Listing 3. Calling the mentioned function as a penultimate, ensures correct handling of the results. Important thing to notice here is that calling these two methods will not have any effect, if *useDefaultRewardCalculation* parameter was set to *true* during the MonteCarloSimulator object definition.

SetEndConditionFunction() is probably the most interesting one from the Listing 5. It features a simple lambda function, which is invoked in a scheduled event frame. This lambda function checks if *bool EndConditionFunction()*, passed as a reference by the user returned a *true* value, and if it did, it ends the simulation.

There are also several functions from a *Get()* family. They were not featured in this chapter, as their only function is to return a reference to arrays, used in reward calculation process or the reference to number of the current stage. These *Get()* functions are introduced to protect the code from accidental changes in core functionalities, while providing the user with the possibility to check them and use them in simulated algorithms.

To use the Monte Carlo Simulator framework, the user creates an instance of MonteCarloSimulator by calling the constructor function from Listing 1 and providing all required input parameters. Additionally, user can provide an external function for reward calculation with the use of the first function from Listing 5. If such function is provided, user should also use the second method to set the collection of appropriate statistics after the *warmupTime*. On the other hand, if such function is not provided, the framework by default will calculate it as the mean throughput from previous non-zero throughput rounds by using the function from Listing 3. By default the framework will also produce a *.csv* output file with the number of the round, throughput and calculated reward (for each player) as shown in Listing 4. Next, an optional parameter is the end condition. If such a function is not provided with the use of the third function from Listing 5, the framework will perform a specified number of rounds. By default the framework will present in the console the results of each of the rounds. If the user wants to obtain in-console results starting from a specified round (e.g., from the 4th one to the last one), the user must provide a positive integer as the *printing* parameter while calling the constructor method. An example of using the optional features is presented in Listing 6.

```
MonteCarloSimulator monteCarloSimulator = MonteCarloSimulator(
    &sinkApplications, numRounds, roundTime,
    roundWarmup, "outputName", 4, true, &BehaviourFunction);
monteCarloSimulator.SetRewardCalculationFunction(RewardCalculationFunction);
monteCarloSimulator.SetEndConditionFunction(EndConditionFunction);
monteCarloSimulator.SetWarmupStatisticsCollectionFunction(WarmupFunction);
```

Listing 6: Exemplary usage of optional features

4 Simulation Results and Analysis

To test the validity of the MonteCarloSimulator library, a test scenario with three algorithms was created. Each scenario follows the general simulation idea. At the start of each round station chooses an Access Point to which it associates, based on the examined algorithm. Next, a short warm–up period occurs after which statistics are collected and the simulation continues. At the end of the round rewards are calculated and results are handled in a way described in Chapter 3. The first examied algorithm is very simple: in each round, each station associates itself to to a random AP. This benchmark scenario serves as reference for the other studies. The other two algorithms were described by Marc Carrascosa and Boris Bellalta in "Multi-Armed Bandits for Decentralized AP selection in Enterprise WLANs" [24]. Although the authors presented extended study, performed in multi–station and multi–AP environment, they also introduced a "toy scenario". This simple scenario with two stations transmitting each to either of two Access-Points is implemented and results from simulations, performed on this topology are used in validation of the implemented solution. The two other algorithms introduced in [24] are ε-greedy and ε-sticky.

This chapter contains a number of tables with **reward** values calculated for each round and figures with **throughput** obtained in each round and **chosen Access Point**. Although these values are dependant, they should not be mistaken and cannot be treated as interchangeable. Tables with reward values present a simulation from algorithm point of view as most of the algorithms make decisions based on them. On the other hand, figures present the state of the network from a user point of view. In other words figures provide an answer for the question: *How does the state of the network look like?* and tables provide answer for: *Why does the state of the network look like this?*.

4.1 Scenario Configuration

The topology of the used network is the same as in the "toy scenario" in article [24]. Two stations have a possibility to associate to two Access Points. The goal of each of the stations is to establish such a connection, which provides it with the highest throughput. This simple network topology is depicted in Figure 7.

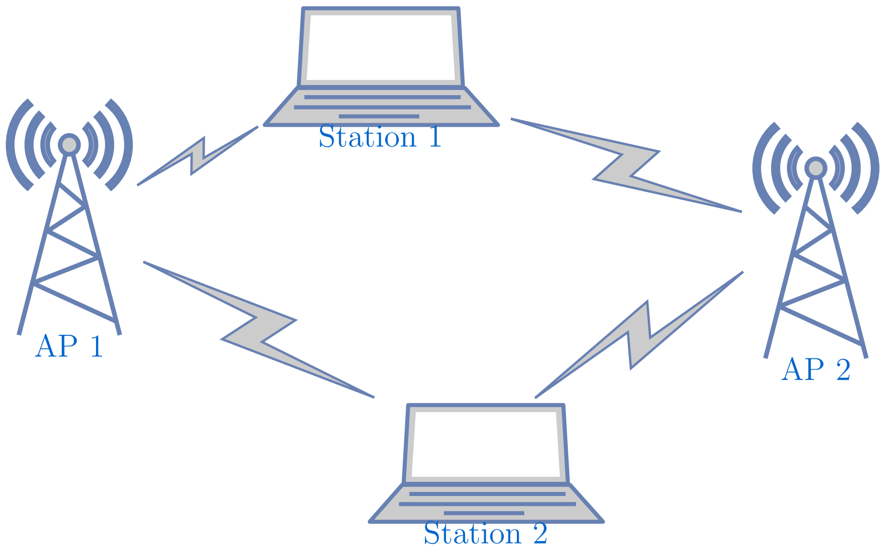

Figure 7: Network topology used in the simulations

Each station is associated at a time to one of two available Access Points. These Station–Access Point flows are configured in such way that one of them should provide theoretically worse throughput and the second one should be more reliable. This situation is achieved by setting an appropriate *Modulation Coding Scheme* and transmission power level, which were chosen for each flow in such a way, that when both stations are associated to the same Access Point, their transmissions interfere. This results in lower throughput, calculated for each station. Additionally, only AP 1 can grant Station 1 a maximum required throughput (when Station 2 is associated to AP 2). Possible associations and throughputs obtained by each station for the combination of each association is gathered in Table 1. The aim of the analysed algorithms is to learn to choose the optimal scenario 2. The whole configuration of the parameters, which were used during the simulation is summarised in Table 2.

Table 1: Possible associations and throughput obtained by stations in each of them

Scenario number	Associations	Throughput	Percentage of required throughput
1	Station 1 – Access Point 1	8.6 Mb/s	71.67%
	Station 2 – Access Point 1	7.6 Mb/s	50.67%
2	Station 1 – Access Point 1	12 Mb/s	100%
	Station 2 – Access Point 2	15 Mb/s	100%
3	Station 1 – Access Point 2	10.7 Mb/s	89.17%
	Station 2 – Access Point 1	15 Mb/s	100%
4	Station 1 – Access Point 2	6 Mb/s	50%
	Station 2 – Access Point 2	11 Mb/s	73.33%

Table 2: Network and algorithm–specific parameters, used during the simulation

Name	Value
Wi-Fi standard	IEEE 802.11ax
Frequency	2.4 GHz
Channel width	20 MHz
Channel number	36, 44
MCS	3, 6, 4, 5
Guard interval	800 ns
Transmission power	-79.5 dBm, -76.45 dBm, -71.1 dBm, -72.3 dBm
Transport layer protocol	UDP
Transmission port number	9, 10, 11, 12
Packet size	1472 B
Application data rate	12 Mb/s, 15 Mb/s
Number of rounds	200
Duration of a single round	12 s
Warmup time	2 s
ε	0.3, 0.5
Sticky Counter	2, 1

There are a few multiple–value entries in the Table 2. The first value in *Channel number* row corresponds to all the transmissions to the Access Point 1 and second value to Access Point 2. Data rate is summarised per–station – first value refers to Station 1 and the second one to Station 2. In case of 4–value entries the scheme the values refer to the following transmissions: Station 1 to Access Point 1, Station 1 to Access Point 2, Station 2 to Access Point 1, Station 2 to Access Point 2. Due to the requirement of maintaining the same transmission power level throughout the whole simulation time, it is assumed that a lossless channel is used. The setting of the transmission power level, represents the attenuation of the wireless channel, as the authors of [24] assumed the constant transmission attenuation. Instead of setting separately the transmission power level and attenuation of the channel, an approach when these two parameters are reflected in the transmission power level is taken.

In Table 2 are also gathered the algorithm–specific parameters: ε and *Sticky Counter*. Their meaning is explained in Chapter 4.4 and Chapter 4.5.

4.2 Code Efficiency

The main idea behind developing MonteCarloSimulator library was to shorten the code, needed to be written to perform Monte Carlo simulations. To provide a reliable metric, two separate simulation scenarios were written: one without the use of the library and the second one, attached as Appendix B, which uses MonteCarloSimulator. The comparison between these two programs is presented in Table 3.

Table 3: Comparison of code length with and without the use of MonteCarloSimulator library

Number of lines in initial code	**Number of line in the code, when MonteCarloSimulator was used**	**Improvement in number of lines**
500	380	24%

It is worth noticing, that the number of the lines of the code from Appendix B is higher than the one in Table 3. The reason behind is that code from Appendix B was adapted to fit into length of the line of this thesis. Values from Table 3 were gathered from the code, which meets the requirement of the clean code development.

4.3 Random Association

First examined algorithm is random Access Point selection. Although not complicated, it is a solid base, on top of which the next elements were subsequently added, forming new algorithms. The results of simulations, performed with a use of this algorithm, are also a reference point for future examinations.

4.3.1 Algorithm Description

In each round, each station separately chooses a random number from a uniform distribution $U[0,1)$. The chosen value is then compared with the next values of the $\{\frac{1}{n}, \frac{2}{n}, ..., \frac{n}{n}\}$ sequence, where n is the number of Access Points in the network. For the purposes of this thesis, 2 Access Points were used, thus the comparison comes to a single condition check – whether the random value is lower or greater than 0.5.

This algorithm requires no reward calculation, as the association condition does not use its value. Although, such cases are not common and most of the simulation are reward–based, thus the decision was made, to not make the reward calculation process optional.

4.3.2 Result Analysis

Unlike Chapters 4.4.2 and 4.5.2, result analysis for the random association algorithm does not contain a table with the rewards, calculated during the first several rounds. The reason is that this algorithm does not use the rewards in decision making thus its analysis does not provide any valuable conclusions. Analysis of the results, depicted in Figure 8 should provide enough information for the purposes of performance evaluation of the MonteCarloSimulator library.

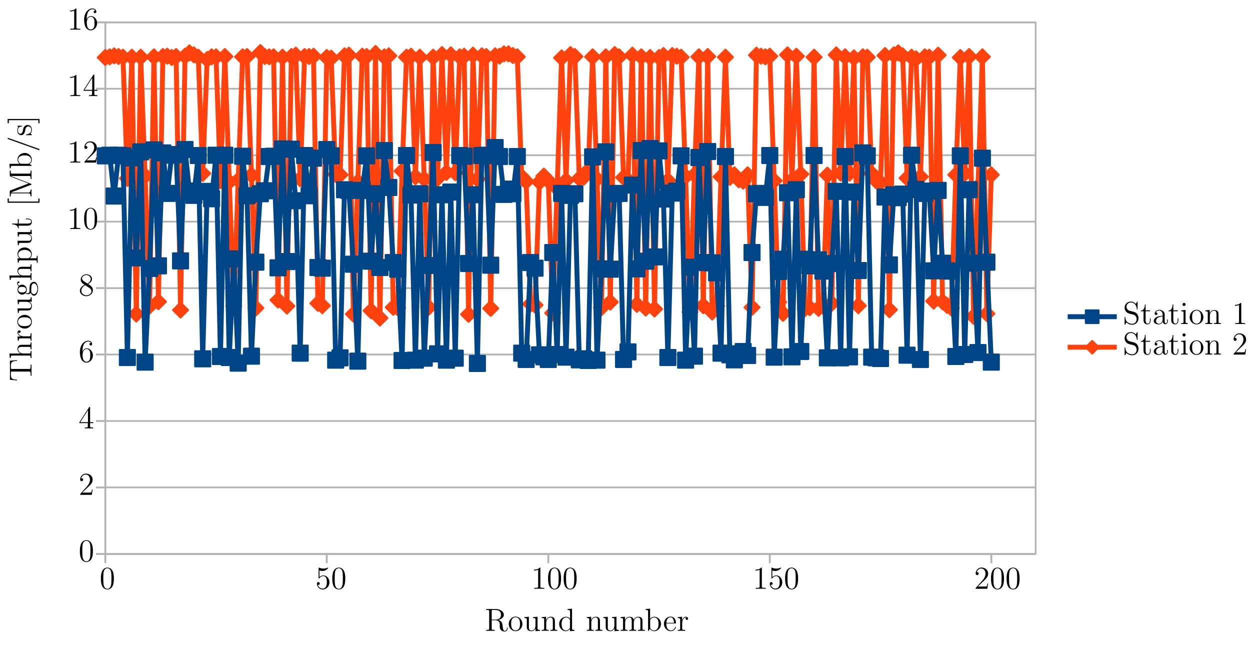

Figure 8: Throughput obtained by stations by randomly associating to Access Points over 200 rounds

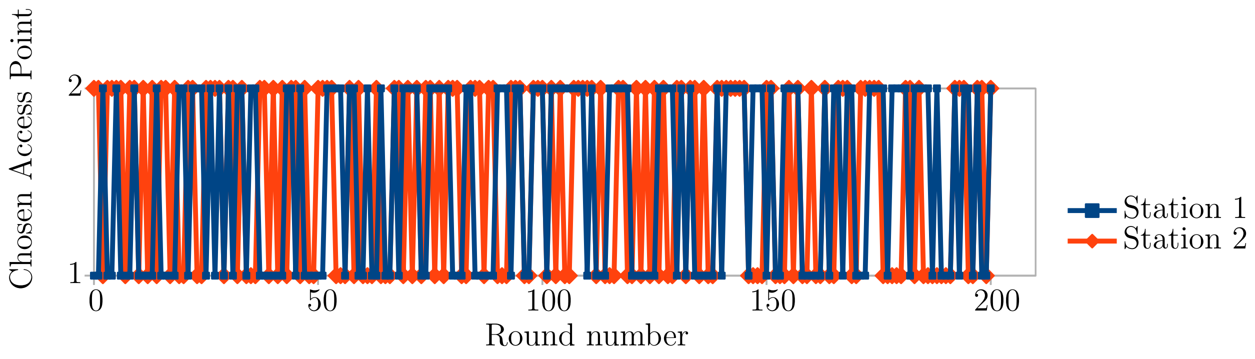

Figure 9: Changes in associations to Access Points over 200 rounds when random association was used

The results of the random association algorithm, depicted in Figures 8 and 9 illustrate the most important characteristic of this approach – a high rate of changes. There are only a few periods, during which the single station maintained a connection with the same Access Point. However, this does not guarantee, that an optimal connection was established. Even after 200 rounds, the algorithm could not provide a desired guidance in connection establishment. Although, it cannot be used as a standalone algorithm, its high rate of changes should create an opportunity for the other algorithms to get out of a sub–optimal scenario – a case when only one station achieves the required throughput.

4.4 ε-Greedy

The first algorithm from the ε–family is a greedy approach. In this case each station behaves in a selfish way – it tries to establish a connection with a highest reward Access Point in each round. However, this approach might not entirely exclude finding an optimum, as the stations interfere with each other. This should be visible in the changes of the rewards, which should potentially enforce stations to choose the optimal connection.

4.4.1 Algorithm Description

The ε-greedy algorithm uses up to two random values in each round. The first one is compared with the provided ε value. It was assumed, that the ε value should represent the probability of the "exploration" of the station in each round – a behaviour when the station does not choose the highest reward, thus both values should be in range [0,1). If the value is lower than ε, the station explores the network – chooses the second random value to associate to an Access Point in the same way as in the random association approach. If the value is higher, the station chooses the Access Point with the highest reward calculated in the previous round. This algorithm is depicted in Figure 10.

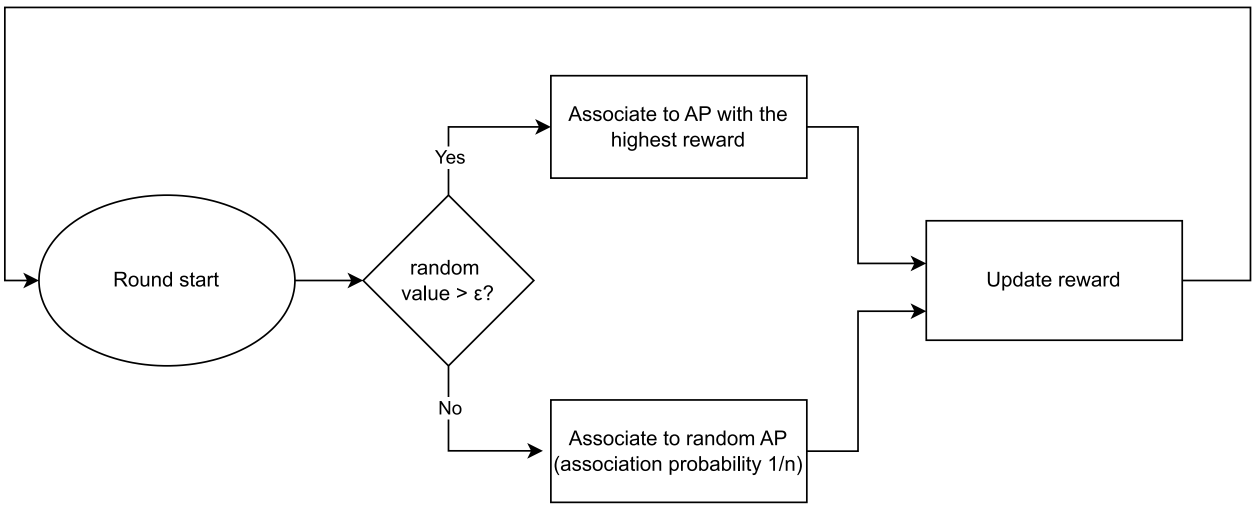
Round start
random value > ε?
Yes
Associate to AP with the highest reward
No
Associate to random AP (association probability 1/n)
Update reward

Figure 10: ε-greedy block diagram

4.4.2 Result Analysis

The first examined ε value is 0.3. With this value, two independent simulations were conducted. These two cases have a major difference – the starting associations. The throughput obtained by each station in each round is depicted in Figures 11 and 13.

The first case is the perfect example of the benefits, introduced with this algorithm. Both stations started from the worst network configuration: Station 1 obtained roughly 70% of the requested throughput, while Station 2 was less fortunate and was satisfied only in about 50%. This situation quickly changes, as the stations begin to explore and luckily for them, they don't interfere with each other during the several first rounds. The stations quickly found an association, which could satisfy both of them in almost 100%.

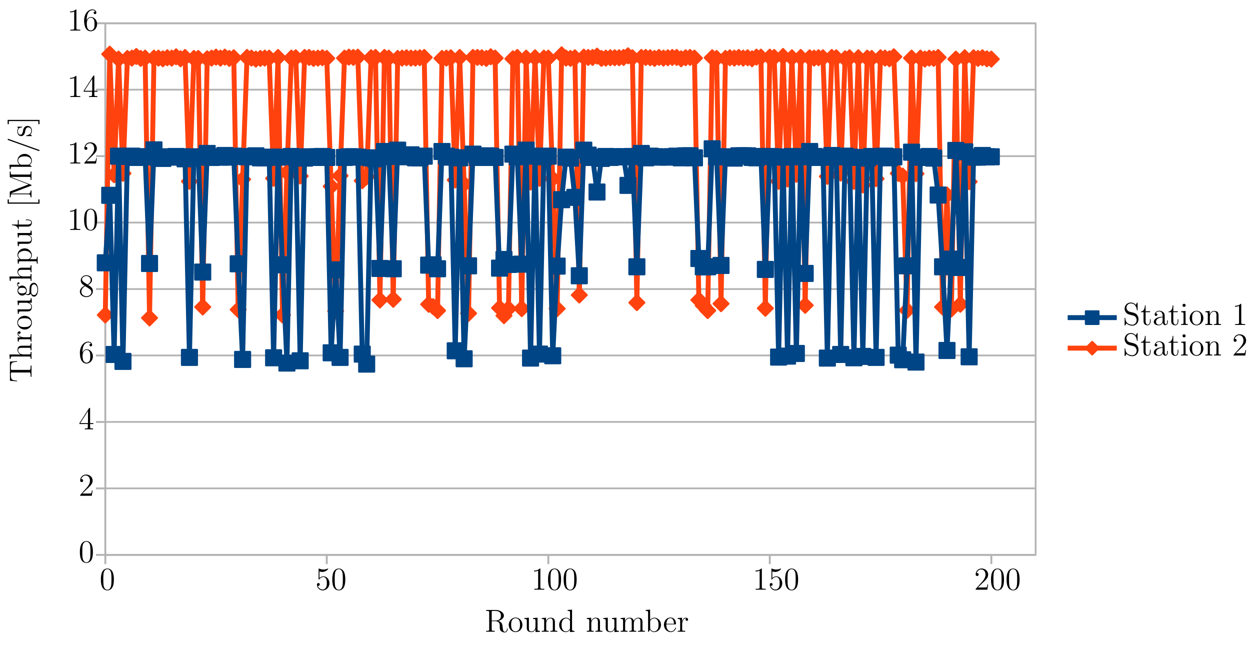

Figure 11: Throughput obtained by stations by associating to Access Points, using ε-greedy algorithm over 200 rounds – case 1

The deviations in the output throughput are not algorithm–specific and more tool-specific. As *ns-3* is an event–based simulator it is normal, that for edge–cases, like the examined one in which the station is on the verge of obtaining the maximum requested throughput, the output result will not be constant. The authors of the simulator emphasize, that it was made to be as realistic as possible [25] and a scenario when the station has a constant maximum throughput is not common in real–life scenarios.

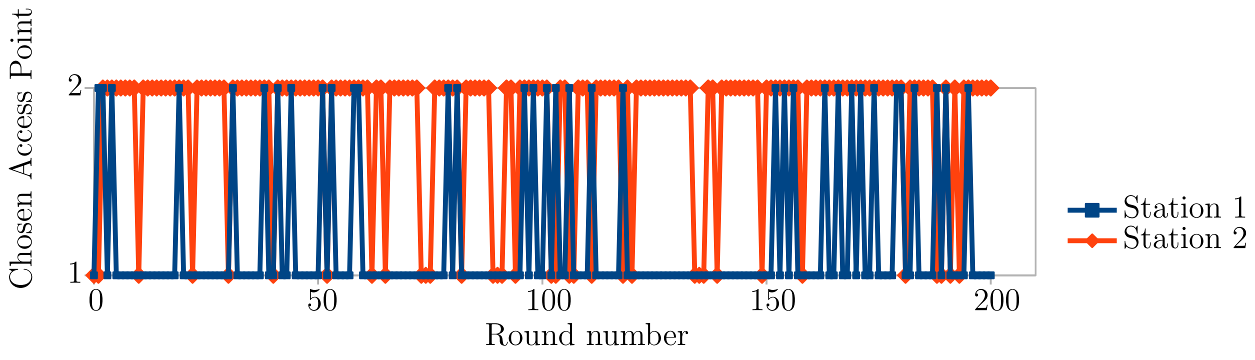

Figure 12: Changes in associations to Access Points over 200 rounds when ε-greedy was used – case 1

Changes in associations during the simulation are presented in Figure 12. If the station started exploration, it did not maintain it for longer than one round. This is the effect of the greedy approach – once station has discovered the association with the highest reward, it maintains it until the better one appears. However, this was not a case during the first simulation, as stations quickly discovered the optimal associations and the rewards could not be decreased to a value, for which the second association would be better. The results are also correct from the statistical point of view, as each station explored the possible actions only in about 25% of rounds.

Case 2 is more interesting, as Station 1 gets stuck in a sub–optimal association with Access Point 1. As shown in Figure 13, an optimal network situation – when both stations are granted their required throughput – happens only twice: in round 140 and 196. As the exploration period is a random process it is not impossible for such scenario to happen. Simply in almost every exploration round Station 1 associated with Access Point 1 at the same time as Station 2. The probability of this happening so many times in a row, makes this example a perfect corner case to be examined. This also raises a question whether the assumed ε was not too low. For the purposes of these two simulations, it was assumed that 70% of the time, station should keep the association with the highest reward Access Point. This point is analysed later in this chapter.

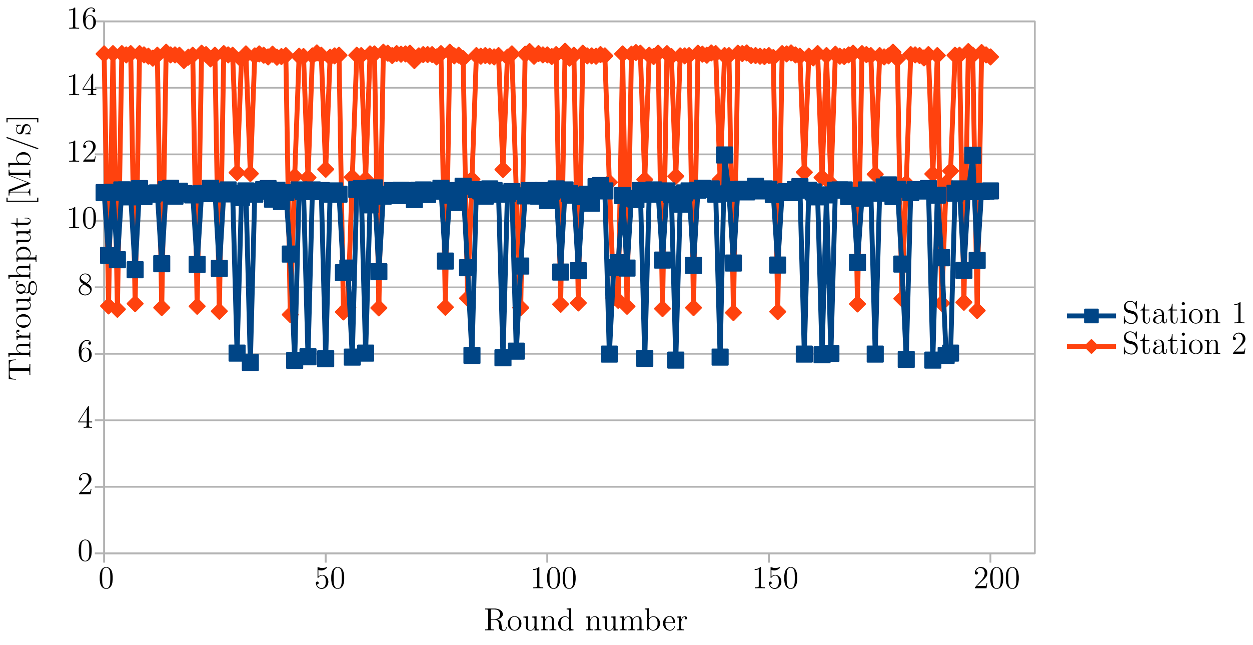

Figure 13: Throughput obtained by stations by associating to Access Points, using ε-greedy algorithm over 200 rounds – case 2

Although the stations have not found the optimal associations in which both of them could be satisfied in 100%, Figure 14 confirms the correctness of the implementation of ε–greedy algorithm. Station 2 dominated the connection to Access Point 1 and almost each time Station 1 tried to explore this connection, it obtained a relatively small reward. There are also two important things to notice on this diagram. The first one is that Station 1 had a few subsequent rounds when it was connected to the Access Point with smaller reward. The second thing is that Station 1 tried to explore the possible actions 61 times

(30.5% of rounds). These two things should not be associated with a fact that a network was not in optimal state, but are a result of random numbers generated during this particular simulation. The algorithm itself does not provide any opportunity to detect a sub–optimal state of the network and because of this such conclusions would be false.

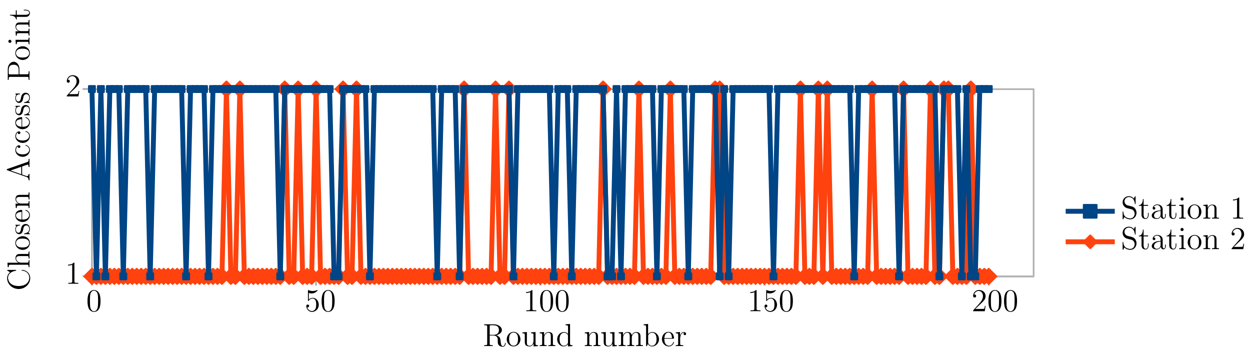

Figure 14: Changes in associations to Access Points over 200 rounds when ε-greedy was used – case 2

When analysing the results of the second case one should think why after the discovering the optimal associations, the algorithm did not continue to use them and instead, it chose to continue transmission to sub–optimally chosen Access Points. The reason behind this is visible, when looking at rewards, obtained by each flow in each round. The rewards calculated in the round 0 and 11 subsequent rounds for both cases are gathered in Table 4.

Table 4: Average reward over time for ε–greedy. Bolded rewards show the chosen Access Point for each association round

Case 1													
		Reward											
Assoc. round		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Station 1	**AP 1**	**0.73**	0.73	0.73	**0.87**	0.87	**0.91**	**0.93**	**0.95**	**0.95**	**0.96**	**0.93**	**0.94**
	AP 2	0	**0.9**	**0.7**	0.7	**0.63**	0.63	0.63	0.63	0.63	0.63	0.63	0.63
Station 2	**AP 1**	**0.48**	**0.74**	0.74	0.74	0.74	0.74	0.74	0.74	0.74	0.74	**0.66**	0.66
	AP 2	0	0	**0.76**	**0.88**	**0.84**	**0.88**	**0.9**	**0.92**	**0.93**	**0.94**	0.94	**0.94**
Case 2													
Station 1	**AP 1**	0	**0.75**	0.75	**0.74**	0.74	0.74	0.74	**0.73**	0.73	0.73	0.73	0.73
	AP 2	**0.9**	0.9	**0.9**	0.9	**0.91**	**0.91**	**0.9**	0.9	**0.91**	**0.9**	**0.9**	**0.9**
Station 2	**AP 1**	**1**	**0.75**	**0.82**	**0.76**	**0.81**	**0.84**	**0.86**	**0.82**	**0.84**	**0.86**	**0.87**	**0.88**
	AP 2	0	0	0	0	0	0	0	0	0	0	0	0

The calculated rewards in case 1 are straightforward. Both stations started the simulation associated do Access Point 1. Then in round 1, Station 1 explored and found an association, which granted the higher reward. In round 2 Station 1 kept the association and Station 2 explored and found association to Access Point 2. Round 3 is the first round when both stations found the optimal association by connecting with the Access Point with a higher reward. From this point until the end of the simulation, both stations further increased the values of these rewards and the explorations of neither of the stations could not change this situation, as there is no other optimal scenario in this network.

Case 2 is more interesting for the rewards analysis. For the first 12 rounds, Station 2 uses either the greedy approach or random approach to associate to Access Point 1. Station 2 in the beginning explores the association to Access Point 1 twice, but because of the association of Station 2 to the same Access Point, the reward, calculated for this flow can never increase and it is even likely to decrease, based on the realism of the *ns-3* implementation. The algorithm has no option to enforce an optimal solution, apart from waiting for the desired scenario to occur. Although as Table 4 contains only the rewards for the first 12 rounds, Table 5 was introduced. This table stores the information about the rewards obtained by each flow between rounds 28 and 39, as it is the first time Station 2 explored the association with Access Point 2.

This change in the association of Station 2 could not be efficient, because Station 1 did not simultaneously switch its association to Access Point 1. In this case, the only thing that really happened in the network was a slight decrease (0.01 and 0.02) in the reward calculated for the flow Station 1–Access Point 2 and calculation of the reward for the flow Station 2–Access Point 2, which was around 16.66% lower than the flow from Station 2 to the other Access Point. The answer for the question why the algorithm did not stick to the optimal scenario from round 140 is that the reward for such connection was simply too low to be considered optimal by the greedy approach.

Table 5: Average reward over time for ε–greedy – first transition of Station 2 to AP 2. Bolded rewards show the chosen Access Point for each association round

		Reward											
Assoc. round		**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
Station 1	**AP 1**	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73
	AP 2	**0.9**	**0.9**	**0.89**	**0.89**	**0.89**	**0.87**	**0.88**	**0.88**	**0.88**	**0.88**	**0.88**	**0.88**
Station 2	**AP 1**	**0.9**	**0.9**	0.9	**0.91**	**0.91**	0.91	**0.91**	**0.91**	**0.92**	**0.92**	**0.92**	**0.92**
	AP 2	0	0	**0.76**	0.76	0.76	**0.76**	0.76	0.76	0.76	0.76	0.76	0.76

This exact situation leads to another question: what if the authors of the article [24] chose $\varepsilon = 0.3$, because of the scale of the network? To answer this question, another simulation, with a different value of the ε parameter was examined. This time, the used parameter was set to 0.5. The obtained throughput throughout the 200 rounds is depicted in Figure 15 and associations of each of the stations in each round are shown in Figure 16.

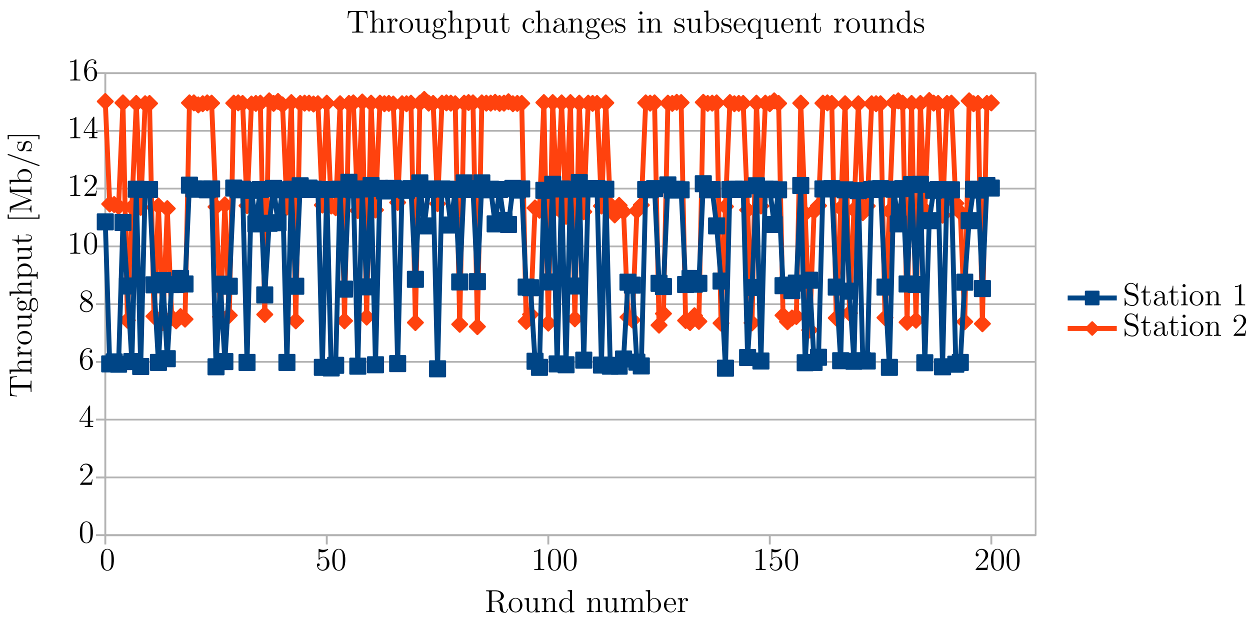

Figure 15: Throughput obtained by stations by associating to Access Points, using ε-greedy algorithm over 200 rounds – case 3

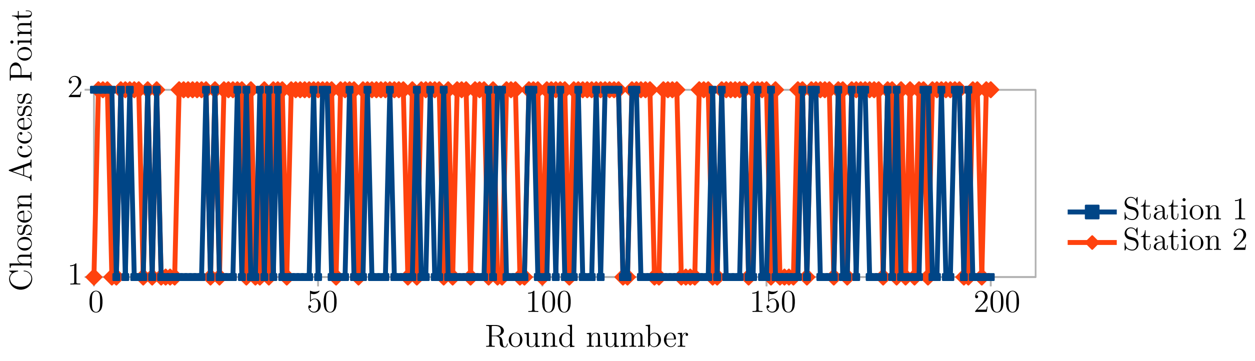

Figure 16: Changes in associations to Access Points over 200 rounds when ε-greedy was used – case 3

Figure 15 resembles Figure 8, but there a few things to notice when examining these results. The first and most important is that the probability of the situation in which station will not choose the highest reward association has increased by 66.67% compared to the previous simulations. This, however, encourages both stations to look for the new association possibilities. The main thing that differs Figure 15 from Figure 8 are short, but visible periods where the network is in an optimal state. What is more important, even when stations explore new association possibilities, they now have a mechanism which helps them establish an optimal connection. This is even better visible in Figure 16. When comparing Figure 16 and Figure 9, the first mentioned looks more regular. The random algorithm resembles a total

When choosing optimal input parameters one should remember to choose the appropriate number of rounds in the simulation. All figures in this thesis depict the situation throughout 200 rounds, but this is made to depict the long–term state of the network. In real–time scenarios, such rapid changes would and will be annoying for the user demanding a high quality of service for such data streams as, e.g., video or sound. The reason behind shortening the number of examined rounds is also visible in the rewards data, an excerpt of which is shown in Table 6.

Table 6: Average reward over time for ε–greedy – case 3. Bolded rewards show the chosen Access Point for each association round

		Reward											
Assoc. round		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Station 1	**AP 1**	0	0	0	0	0	**0.72**	0.72	**0.86**	0.86	**0.9**	**0.93**	**0.89**
	AP 2	**0.9**	**0.7**	**0.63**	**0.6**	**0.66**	0.66	**0.63**	0.63	**0.61**	0.61	0.61	0.61
Station 2	**AP 1**	**1**	1	1	1	**1**	**0.83**	0.83	0.83	0.83	0.83	0.83	**0.75**
	AP 2	0	**0.76**	**0.76**	**0.76**	0.76	0.76	**0.76**	**0.81**	**0.8**	**0.83**	**0.85**	0.85

Similarly to results obtained for $\varepsilon = 0.3$, when stations are exploring the possibilities, their reward varies over rounds. Station 1–Access Point 2 flow reward is constantly decreasing, while flow from Station 2 to the same Access Point has a constant reward over rounds. What is also worth noticing is that Station 2–Access Point 1 had a reward equal 1 for five rounds and still this association was optimal in terms of the whole network. This was also a breaking point for the algorithm, as in each subsequent round in which both stations kept the optimal association, the reward for this flow was increased with every round. The two exploration rounds (8 and 11) played an important role, as from round 11 the two non–optimal flows had a worse reward than the optimal ones. If the simulation was to end after this round and the associations were to be made based on the values of the rewards, the network would end in an optimal state.

4.5 ε-Sticky

The second algorithm based on the ε value is a development of the concept, introduced in Chapter 4.4. While still promoting the highest–reward association, an additional sticky-mechanism is added to ensure that once the optimal state of the network is reached, the association should not change. Although this algorithm manages some of the problems which arouse with the use of ε-greedy algorithm, the stickiness concept is not flaw-free.

4.5.1 Algorithm Description

The new parameter introduced in ε-sticky algorithm is *stickyCounter*. It is an integer value based on which an association retention is realised. The main problem with ε-greedy is that it does not support choosing an optimal state of the whole network in a longer period of time. The best example for such a situation are results depicted in Figue 13, where the optimal state was reached, but because of the results of the previous rounds it was not chosen as the best one. With the sticky counter such a situation will be maintained for a *sticky-Counter* number of rounds.

When using this algorithm, if a station associates with an Access Point, which can grant it 100% of the required throughput, the station will choose this Access Point in the next *stickyCounter* rounds. At the end of the round, the obtained throughputs are compared with throughput required by a station. If both of these values are the same, the Sticky Counter is set to *stickyCounter*. If the flow had a positive Sticky Counter value at the start of the round and ended the round with a throughput smaller than required, the Sticky Counter is decreased by 1. If at the start of the round the Sticky Counter value is a non–positive one, the station will use the ε-greedy algorithm to ensure both choosing the best possible association as well as possibility to explore the other association possibilities. This algorithm is depicted in Figure 17.

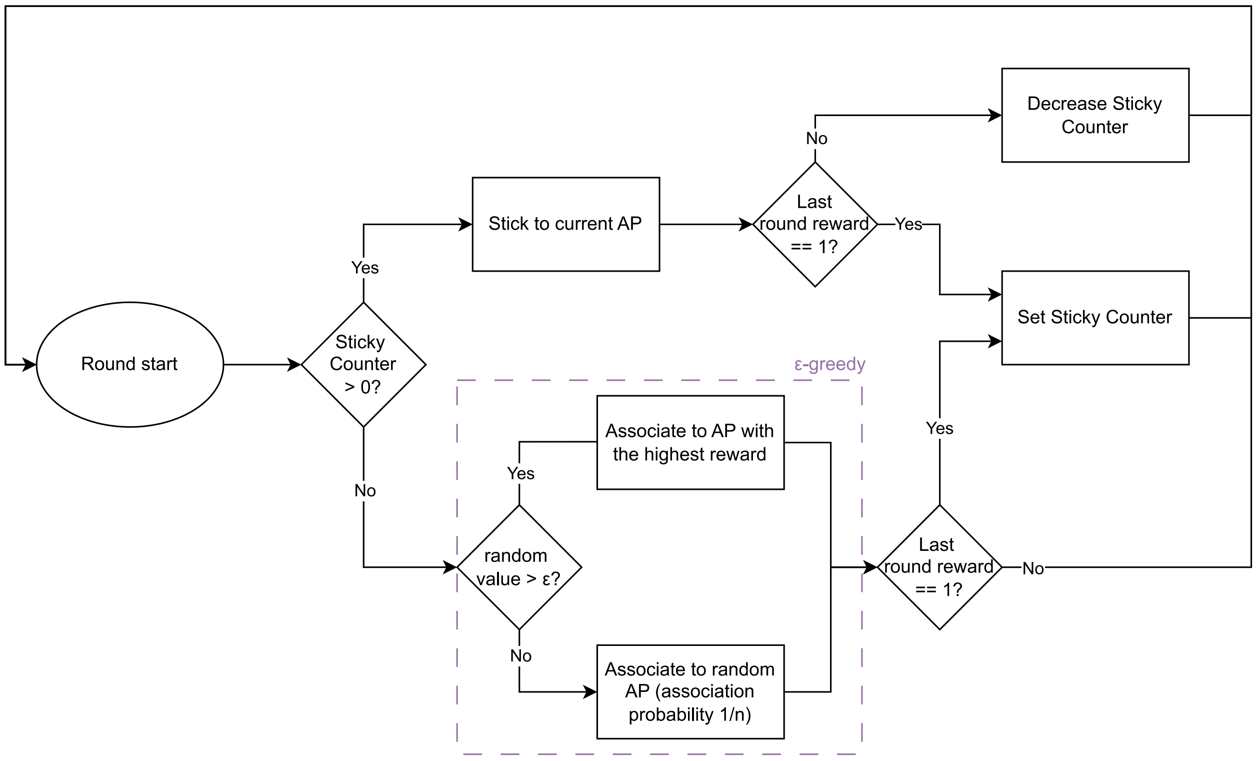

Figure 17: ε-sticky block diagram

Due to the randomness inherent to the *ns-3* simulator, the last condition was changed during the implementation. In the implementation of the algorithm, the station needs to obtain at least 0.996 of the required throughput. This roughly equals a situation when the obtained reward was rounded to two decimal places, as obtaining a perfect 1 as a reward is an unlikely scenario.

4.5.2 Result Analysis

The two initial simulations with ε-greedy were conducted with the following parameters: $\varepsilon = 0.3$, $stickyCounter = 2$. Like in Chapter 4.4.2, these two simulations have different starting associations. The obtained throughput for each station throughout 200 rounds is depicted in Figures 18 and 20.

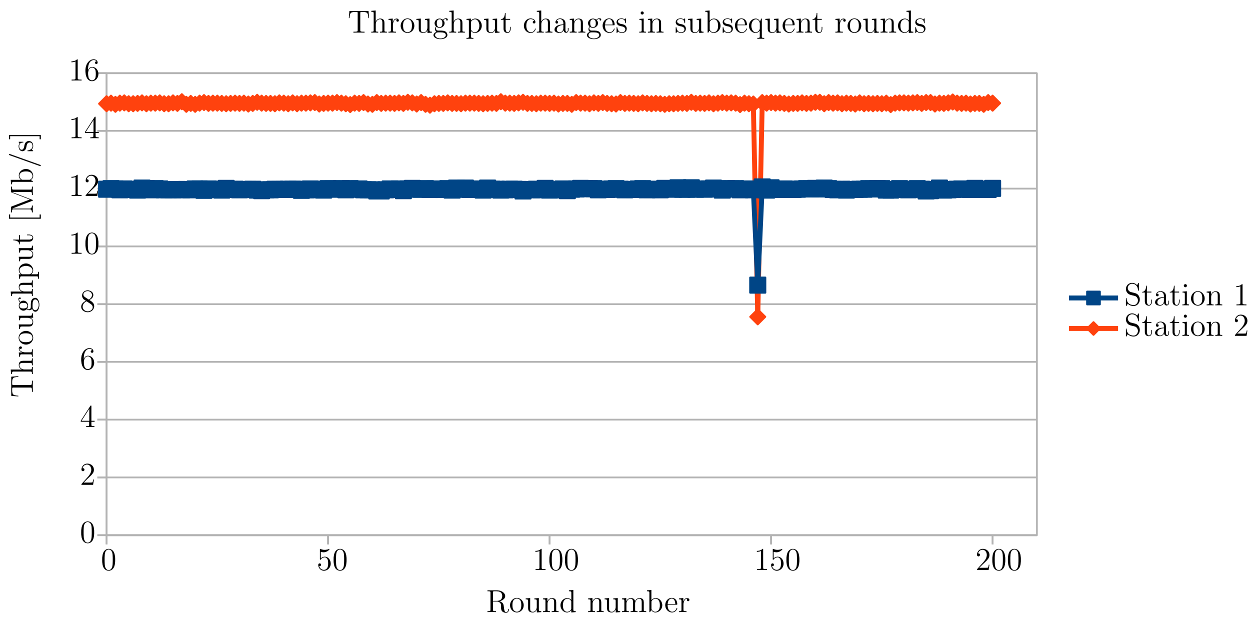

Figure 18: Throughput obtained by stations by associating to Access Points, using ε-sticky algorithm over 200 rounds – case 1

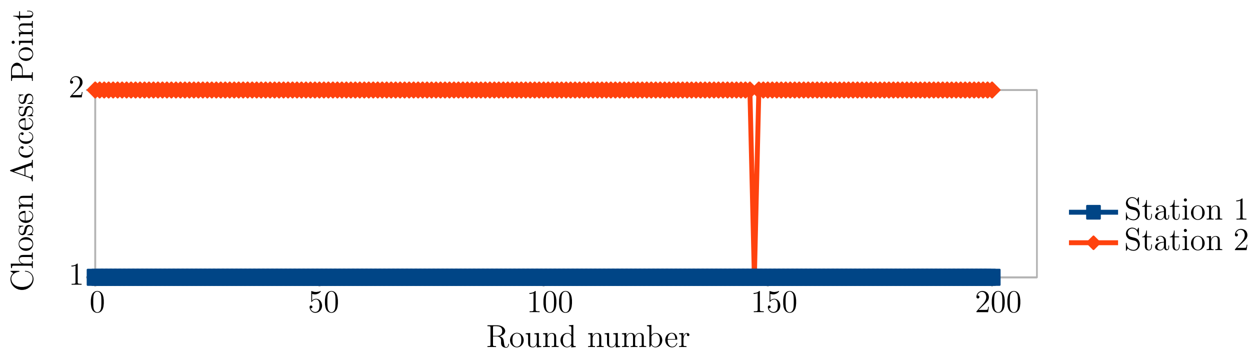

Figure 19: Changes in associations to Access Points over 200 rounds when ε-sticky was used – case 1

The first simulation is a perfect example of the power of the ε-sticky algorithms. It is a perfect scenario during which the network was not in an optimal configuration only during round 147. In every other round the associations were used either by sticking to the previous Access Point or by choosing the association with the highest reward. This situation is theoretically impossible, as each station should have 100% of the required throughput available, but because of the implementation of *ns-3* there are some fluctuations in the output results which lead to non–positive Sticky Counter values. Round 147 is a perfect example of a round where the rewards from the two previous rounds were smaller than 0.996 of the required throughput and both stations chose to explore possible actions. What is also visible thanks to Figure 19 is a fact that only Station 2 explored the other possible association, but immediately after the ε-greedy algorithm changed the association to the one with the highest reward.

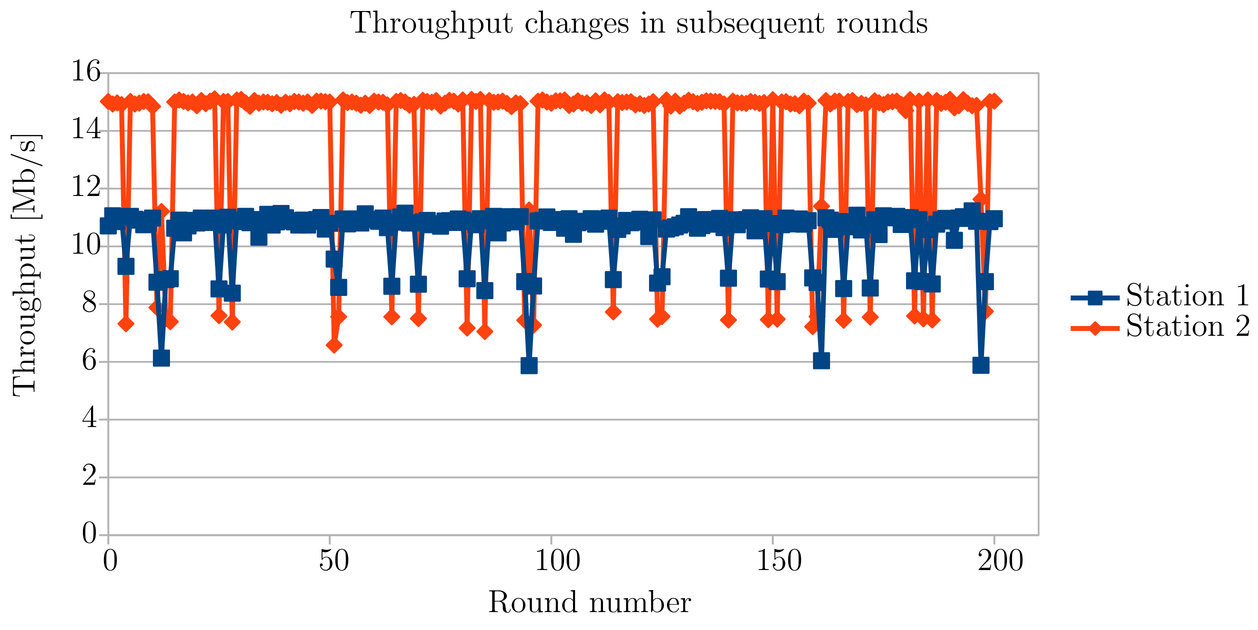

Figure 20: Throughput obtained by stations by associating to Access Points, using ε-sticky algorithm over 200 rounds – case 2

The second scenario is the same as the second scenario from Chapter 4.4.2 – the starting configuration assumes sub–optimal associations where only Station 2 is offered the required throughput. Unfortunately, the introduction of the Sticky Counter does not have an impact on the results for the assumed values of the input parameters. Due to the fact that Station 2 is almost exclusively associated to Access Point 1, because of the values of the rewards and due to random nature of the network exploration process, the throughput offered for Station 1 never reaches the required throughput.

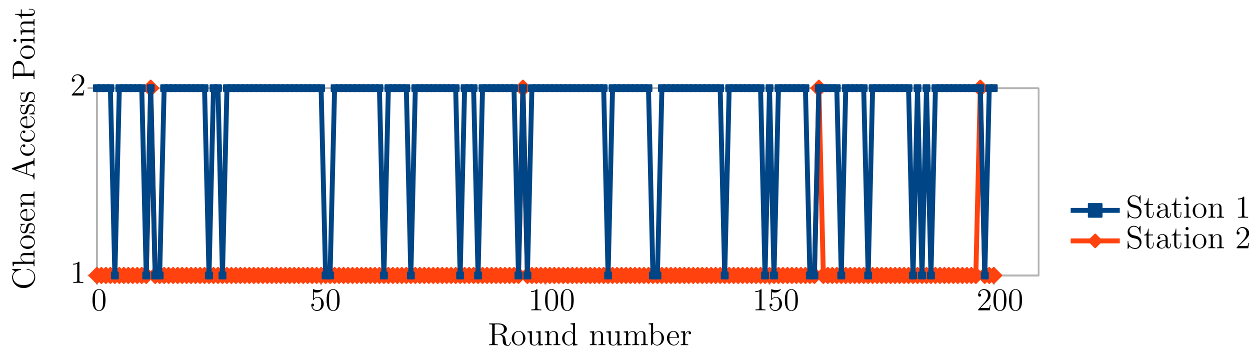

Figure 21: Changes in associations to Access Points over 200 rounds when ε-sticky was used – case 2

Figure 21 is a proof, that examined situation in the network is correct algorithm–wise. There are two distinct behaviours each assumed by a different station. Station 1 did not meet conditions to set a sticky counter in any round. Based on the block diagram of ε–sticky algorithm from Figure 17, Station 1 took a greedy approach so the associations diagram resembles Figure 14. On the other hand Station 2 found an Access Point which could satisfy it in 100% and most of the time it took sticky approach. The only four times when Station 2 explored the association to Access Point 2 are the results of the fluctuations of output results and are connected to the implementation of *n-3* and not to the implementation of neither ε–sticky algorithm nor MonteCarloSimulator. It is also important to analyse the average reward calculated in the first 12 rounds. The rewards are placed in Table 7.

Table 7: Average reward over time for ε–sticky. Bolded rewards show the chosen Access Point for each association round

Case 1													
		Reward											
Assoc. round		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Station 1	**AP 1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
	AP 2	0	0	0	0	0	0	0	0	0	0	0	0
Station 2	**AP 1**	0	0	0	0	0	0	0	0	0	0	0	0
	AP 2	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Case 2													
Station 1	**AP 1**	0	0	0	0	**0.78**	0.78	0.78	0.78	0.78	0.78	0.78	**0.75**
	AP 2	**0.89**	**0.91**	**0.91**	**0.91**	0.91	**0.91**	**0.91**	**0.91**	**0.91**	**0.91**	**0.91**	0.91
Station 2	**AP 1**	**1**	**1**	**1**	**1**	**0.9**	**0.92**	**0.93**	**0.94**	**0.95**	**0.95**	**0.96**	**0.92**
	AP 2	0	0	0	0	0	0	0	0	0	0	0	0

The first case is not interesting for the purposes of the reward analysis. Each station obtained an exact 1 in every round and even the drop in the throughput in round 147 did not change the rounded average reward for Station 1 (Station 1 remained associated to Access Point 1 and Station 2 explored association to this Access Point). What the ε–sticky algorithm did better than ε-greedy is the fact that it retained the optimum associations in 99.5% of the time, while ϵ-greedy periodically forced stations to change their associations.

Case 2 is another example of the weaknesses of the algorithm from the ε family. If the input parameters are not matched with the analysed scenario, the results will never be optimal. In this case if only Station 2 explored in round 11 and found the next association with reward equal to 1, it might have reached its optimum. Although, in this particular round Station 2 was bound to Access Point 1 by the Sticky Counter. It is not visible in the average reward table, but in the previous round Station 2 received the highest possible reward and had to stick to the current association. Although the authors of [24] examined that *stickyCounter* $= 2$ is an optimal value, it might not necessarily be true for smaller networks.

To examine this hypothesis, another simulation was executed, this time with the parameters $\varepsilon = 0.5$ and *stickyCounter* $= 2$. The new value of ε was used previously as an additional simulation scenario in Chapter 4.4.2, but the results at the end of the simulation did not change when compared with those obtained for $\varepsilon = 0.3$. It became obvious that the improved version of that algorithm should also use this value in order to examine if the introduction of the Sticky Counter would help in discovering optimal associations. Throughput obtained by each station in this experiment is depicted in Figure 22 and the selection of Access Point by each station is depicted in Figure 23.

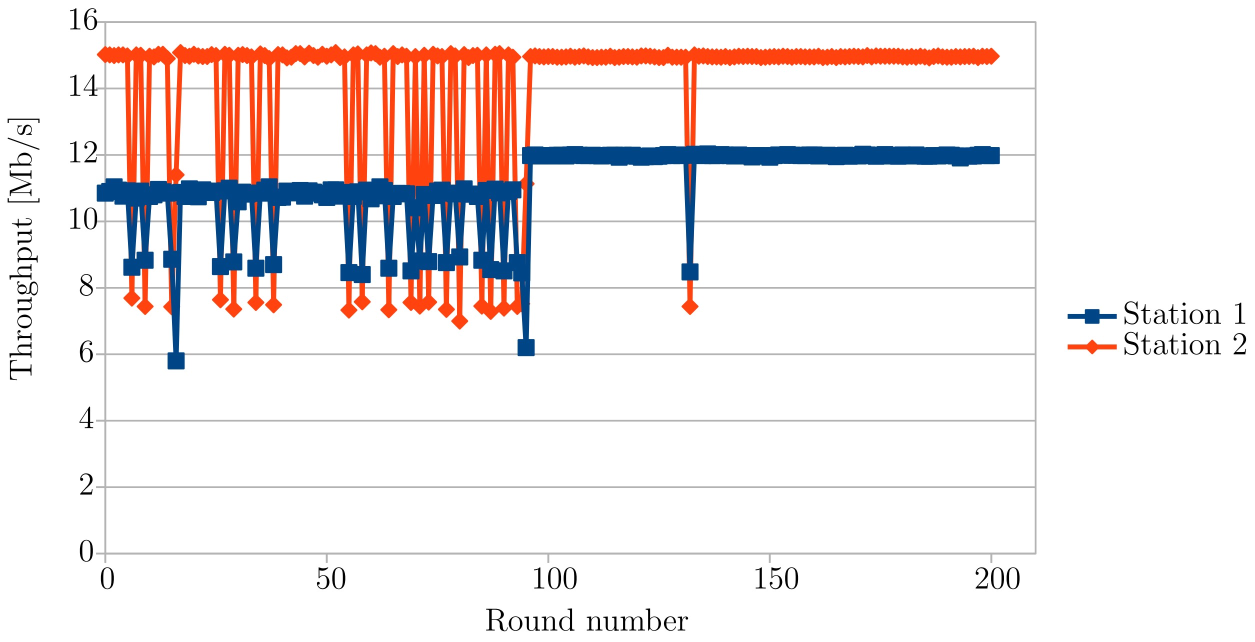

Figure 22: Throughput obtained by stations by associating to Access Points, using ε-sticky algorithm over 200 rounds – case 3

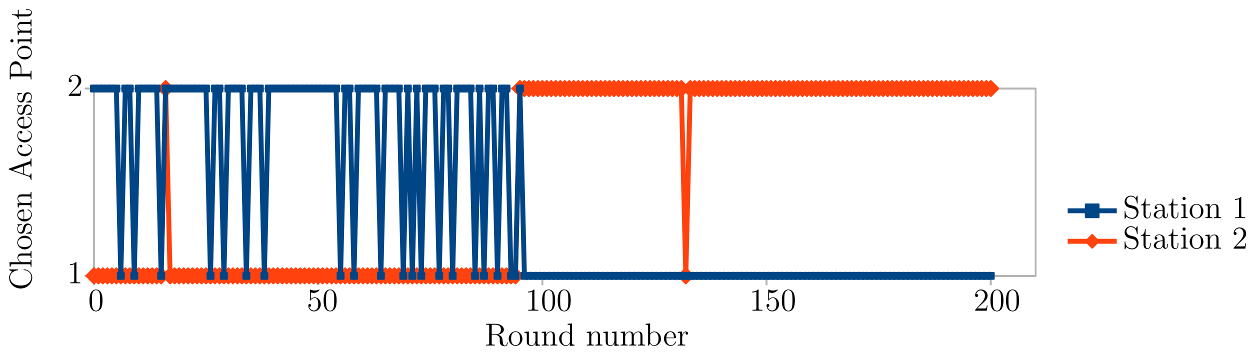

Figure 23: Changes in associations to Access Points over 200 rounds when ε-sticky was used – case 3

The results obtained for the third case are a perfect synthesis of the two previous ones. For the first 95 rounds, both figures resemble their equivalent from case 1. Station 1 explores with a similar frequency as in the case 3 from Chapter 4.4.2 where ε value was also set to 0.5. On the other hand Station 2 behaves fully as it was intended in the sticky part of ε-sticky algorithm. The only three moments when it explored the other possible association was in rounds 16, 95 and 132, from which the middle one is the most remarkable of them all. From round 95 everything changed and both stations behaved like in the first case from this chapter. The reason behind this was that once each station was granted 100% of the required throughput, the Sticky Counter was used and only moment when the stations would again explore was when the fluctuations of the output results were so high during two consecutive rounds that the counter could not have been set and ε-greedy algorithm would choose exploration behaviour.

When the reason behind this scenario from the point of view of the user is clear, it is also important to analyse the state of the network from its own perspective. Table 8 contains the rewards calculated for the first 12 rounds, while Table 9 summarises the rewards calculated just before the crucial point of the simulation and a few rounds after.

Table 8: Average reward over time for ε–sticky – case 3. Bolded rewards show the chosen Access Point for each association round

Reward													
Assoc. round		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Station 1	**AP 1**	0	0	0	0	0	0	**0.72**	0.72	0.72	**0.73**	0.73	0.73
	AP 2	**0.9**	**0.91**	**0.91**	**0.91**	**0.91**	**0.91**	0.91	**0.91**	**0.91**	0.91	**0.91**	**0.9**
Station 2	**AP 1**	**1**	**1**	**1**	**1**	**1**	**1**	**0.93**	**0.94**	**0.95**	**0.9**	**0.91**	**0.92**
	AP 2	0	0	0	0	0	0	0	0	0	0	0	0

Table 9: Average reward over time for ε–sticky – case 3, rounds 94 – 105. Bolded rewards show the chosen Access Point for each association round

Reward													
Assoc. round		**94**	**95**	**96**	**97**	**98**	**99**	**100**	**101**	**102**	**103**	**104**	**105**
Station 1	**AP 1**	**0.72**	0.72	**0.74**	**0.75**	**0.76**	**0.77**	**0.78**	**0.79**	**0.79**	**0.8**	**0.81**	**0.81**
	AP 2	0.9	**0.89**	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.89	0.89
Station 2	**AP 1**	**0.9**	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
	AP 2	0.76	**0.75**	**0.83**	**0.87**	**0.9**	**0.92**	**0.93**	**0.94**	**0.94**	**0.95**	**0.95**	**0.96**

The first six rounds are a perfect examples for both of the algorithms. Station 1 took greedy approach and kept its associations to an Access Point with the highest reward, while Station 2 sticked to Access Point 1, as it was able to grant it 100% of the required throughput. This approach did not change even when Station 1 decided to explore the other possibility as Sticky Counter during this simulation was set to 2 and the unsatisfying period lasted only for one round at a time in both round number 6 and 11. This changes when both stations decided to associate to Access Point 2 in round 95. Although neither of them obtained a 100% of the request throughput, thanks to 1–in–2 chance of choosing randomly in the next round, both stations managed to choose the optimal network–wise associations. This triggered a Sticky Counter setting condition and both stations managed to maintain an optimal associations for a long enough time, that even after switching to greedy approach due to the results fluctuations they were able to maintain an optimal connection. The situation when reward for the flow from each station to the optimal Access Point exceeded the value of the other reward happened in round 99 for Station 2 and 128 for Station 1.

Although the input parameters from the previous case helped to reach the optimal state of the network, there is also one parameter which has not been changed – *stickyCounter*. While it is obvious, that greater values would not help to reach the optimal state faster, because of the slow rate of changes and small number of possible associations, it is still unknown whether the smaller value would help or not. As the *stickyCounter* must be a positive integer number (numbers smaller than 1 would effectively change the algorithm to its greedy version) the only value which could be examined is 1. The results of the simulations performed with $\varepsilon = 0.5$ and *stickyCounter* $= 1$ parameters are presented in Figures 24 and 25.

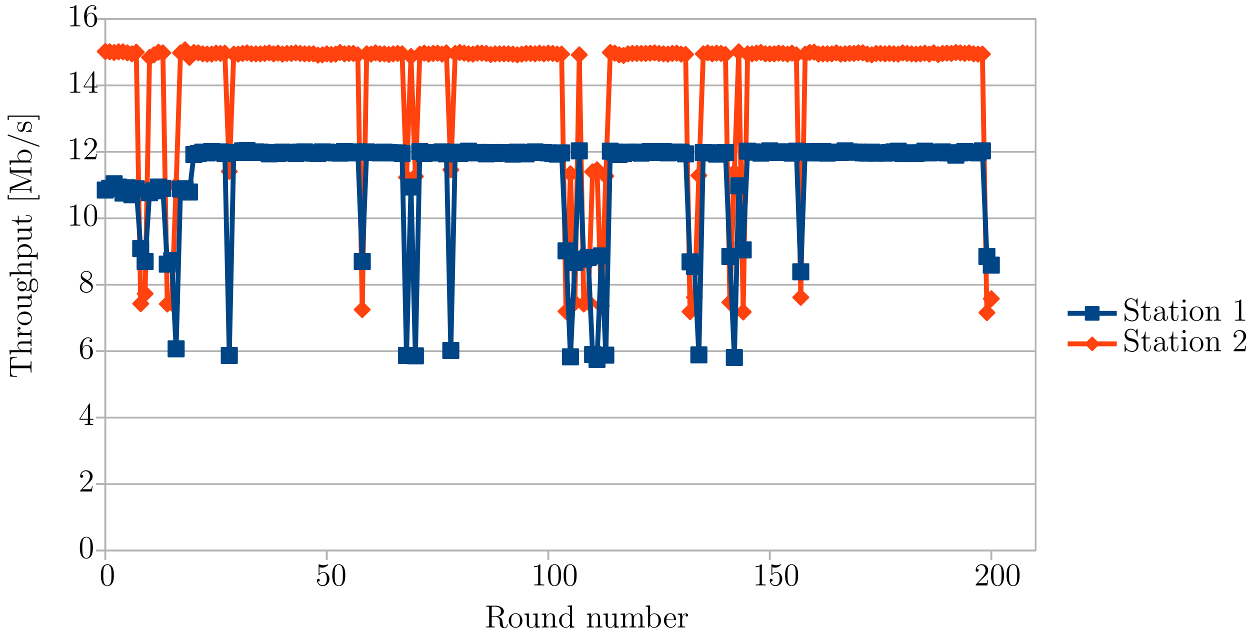

Figure 24: Throughput obtained by stations by associating to Access Points, using ε-sticky algorithm over 200 rounds – case 4

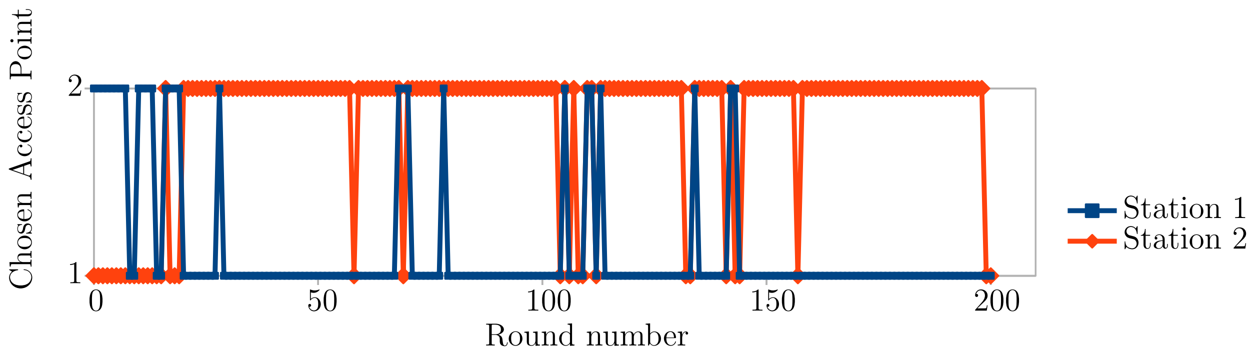

Figure 25: Changes in associations to Access Points over 200 rounds when ε-sticky was used – case 4

This time the optimal state of the network was reached just after 20 rounds of simulation. As it is pointless to analyse the results round after round, there are a few important notes to take from the course of this simulation. Setting a *stickyCounter* with such low value results in more rounds where stations explored and thus the user had lower available throughput. This is especially important in a network where the throughput tends to fluctuate. The Sticky Counter which should be a safety fuse for such situation will not work as intended as a single round of miss–association might result in the exploration of the network which now have 1–in–2 chance to occur. The other important thing to consider is why did stations reach an optimal state of the network in case 3? An answer for this question is simple: because there were only two rounds, during which Station 1 connected to the Access Point 2 while Station 2 was associated to it. The importance of this fact might be easily overlooked, but in reality the reward for the flow Station 2–Access Point 1 did not have a chance to be degraded and this was the main reason why the reward for the optimal association for Station 2 overtook the reward of the other flow much quicker, than for Station 1. In a scenario presented in case 4, if stations did not find the optimal associations withing the first rounds, the rewards for the optimal solution might have been in shatter and as the *stickyCounter* is set to such a low value, that the fluctuations in the output results might prevent finding the optimal state just like in case 2.

Table 10: Average reward over time for ε–sticky – case 4. Bolded rewards show the chosen Access Point for each association round

Reward													
Assoc. round		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Station 1	**AP 1**	0	0	0	0	0	0	0	0	**0.76**	**0.74**	0.74	0.74
	AP 2	**0.9**	**0.91**	**0.91**	**0.91**	**0.91**	**0.91**	**0.91**	**0.91**	0.91	0.91	**0.91**	**0.91**
Station 2	**AP 1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0.94**	**0.9**	**0.91**	**0.92**
	AP 2	0	0	0	0	0	0	0	0	0	0	0	0

Table 11: Average reward over time for ε–sticky – case 4, rounds 15 – 26. Bolded rewards show the chosen Access Point for each association round

Reward													
Assoc. round		**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
Station 1	**AP 1**	**0.73**	0.73	0.73	0.73	0.73	**0.78**	**0.82**	**0.85**	**0.86**	**0.88**	**0.89**	**0.9**
	AP 2	0.91	**0.88**	**0.88**	**0.88**	**0.88**	0.88	0.88	0.88	0.88	0.88	0.88	0.88
Station 2	**AP 1**	**0.87**	0.87	**0.88**	**0.89**	**0.89**	0.89	0.89	0.89	0.89	0.89	0.89	0.89
	AP 2	0	**0.73**	0.73	0.73	0.73	**0.87**	**0.91**	**0.93**	**0.94**	**0.95**	**0.96**	**0.96**

The rewards, gathered in Tables 10 and 11 are just another confirmation for the hypothesis above. When looking at round 20 after which the optimal associations were found it is very impressive that reward for flow Station 2–Access Point 2 overtook the reward for the second flow in just two rounds. The other station had to wait for this situation to happen for a little longer – six rounds – but it still a small number compared to the ones from case 3. But again, Station 1 used this connection just three times before and Station 2 only once, so when the reward, which is a mean throughput obtained during the rounds when the connection was used, was calculated the fraction denominator was still a low number and the sum was not much degraded compared to the maximum achievable sum of throughputs.

4.6 Performance Comparison

Apart from the quality research, described in Chapters 4.3– 4.5, a quantity research was conducted. Each algorithm was used 20 times with a random initial configuration. Used input parameters are: $\varepsilon = 0.3$ and $stickyCounter = 2$. The average throughput granted for each station during the course of all the algorithm–specific examinations are depicted in Figure 26.

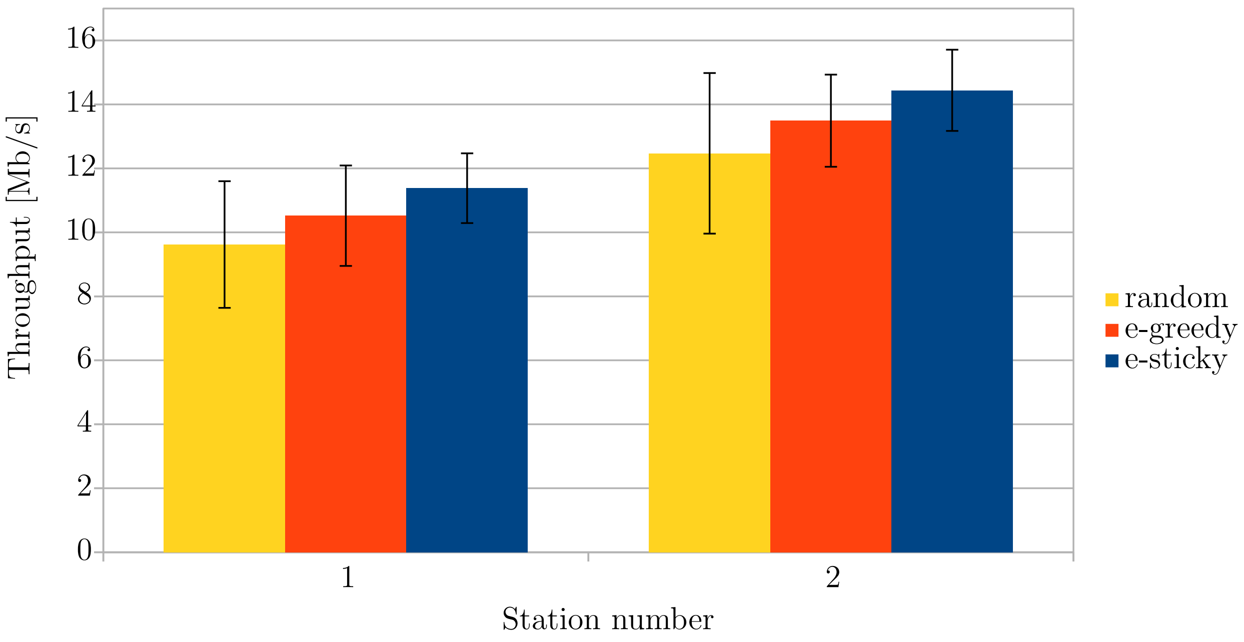

Figure 26: Average throughput granted for each station with the use of the analysed algorithm

As a reminder, the requested throughput for Station 1 is 12 Mb/s and 15 Mb/s for Station 2. As shown in Figure 26, ε-sticky algorithm grants the biggest available throughput for both stations. What is also important this algorithm has also the smallest standard deviation, which means that statistically the network should be the least susceptible to fast and random changes in the offered speed, when this algorithm is deployed in the network. The second important thing in the analysis of these protocols is the final per–flow reward – value of the reward for each flow after 200 rounds.

The results of the second quantitative research are similar to the first ones. Although, there are a few things, visible in Figure 27, worth considering. The random algorithm has the smallest standard deviation of the output results, but this does not mean, that this algorithm is the most reliable one. The results are random, which means that there are no extreme cases in the network and all the scenarios has the same probability of occurrence. On the other hand, if the configuration is to be made based on the final rewards from the 200 rounds, this algorithm can provide a reliable results in this particular case, but it is more scenario– and configuration–specific then algorithm–specific. In bigger networks, these data would be gibberish, as there are more possible associations and more interfering with each other flows. The most interesting thing about the ε algorithms are two flows from the sticky approach – Station 1–Access Point 2 and Station 2–Acceess Point 1. The standard deviation in both cases is very high, but repetition of simulations will not reduce this to much small value. The reason behind is that these two flows are very susceptible to extreme cases in the network. Either these two flows are chosen and are used for almost the whole time, with the reward 0.9 for the first one and 1 for the second, or are almost not used with the reward either 0 or around 0.5.

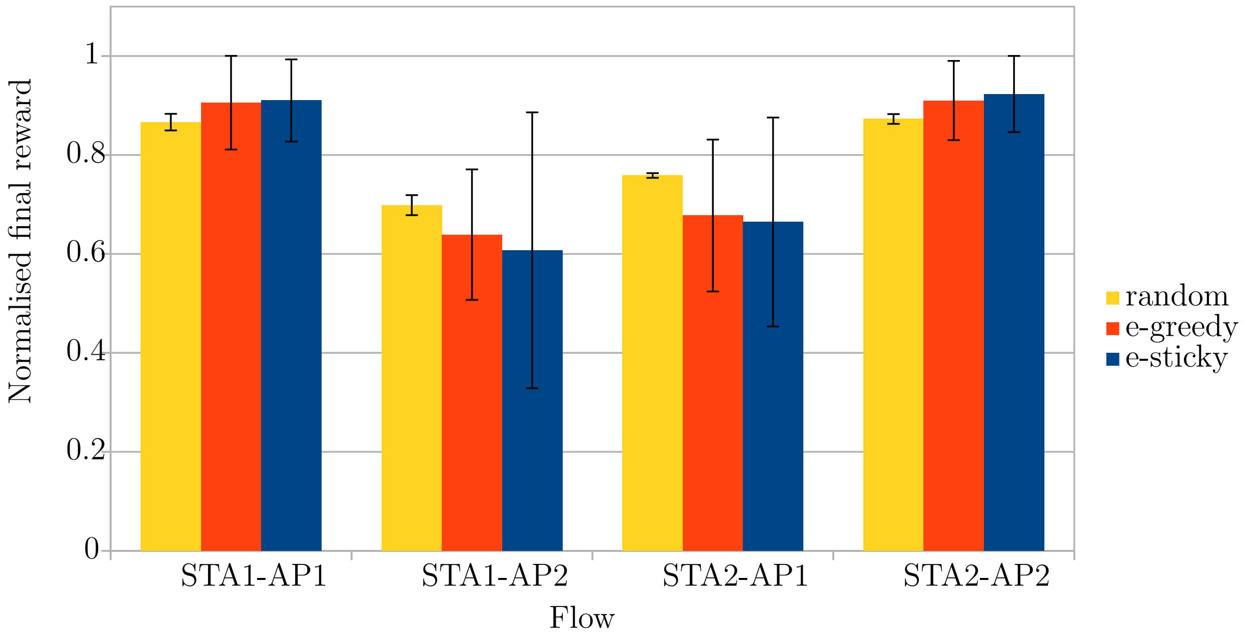

Figure 27: Average final reward calculated for each flow

4.7 Applicability in Other Research

This chapter contains an exemplary configuration which has to be done in order to perform Monte Carlo simulations with examined library. Configuration of the scenario from the reference paper [19] should be made by following below steps:

1. Configure the network topology;
2. Ensure the connectivity between required nodes;
3. Install queues on the appropriate nodes;
4. Implement a state diagram, shown in Figure 1, which is a behaviour method for this simulation;
5. Prepare the arrays to store results: Waiting time, Processing time, Sojourn time, # of people in System, # of people in Queue;
6. Prepare the result calculation method, which gathers data from appropriate queues;
7. Prepare the warm–up statistics collector method, which gathers the statistics after *warmupTime*;
8. Invoke MonteCarloSimulator constructor and pass behaviour method, information about duration of rounds, warm–up time and number of rounds, decide whether to print re-

sults in the console and pass *useDefaultRewardCalculation* as *false*;

9. Pass warm–up collector as a parameter to *SetWarmupStatisticsCollectionFunction* method;
10. Pass result calculation method as a parameter to *SetRewardCalculationFunction* method;
11. Run simulations;
12. Analyse the results and prepare appropriate diagrams.

The list of the required steps might seem long and the implemented library does not seem to appear in many of them, but it does not mean that its usefulness can be marginalised. With so much to do to examine a single scenario, MonteCarloSimulator eases the process by providing out–of–the–box solution, which automatically schedules appropriate events in a desired timeframe.

5 Summary

The main objective of this thesis was to implement a new library capable of conducting Monte Carlo simulations in *ns-3* simulator and to validate its implementation. To answer the question if these two goals were met, one should take a look back in the Chapter 4, where three different algorithms were implemented. The results showed that the algorithms does not converge into an optimal solution in every analysed case. Some limitations, preventing this convergence are not covered by the algorithm and no suitable solution was proposed. The algorithms are heavily dependant on the initial state of the network and generated random numbers.

These inconveniences are not a result of the implementation of the library itself, but are more algorithm–specific. This fact does not influence the output verdict – the implemented Monte Carlo library works as intended. Paradoxically these divergences visible in the output data prove, that the implemented solution does not always provide an optimal output data, but rather is able to point out the possible negative scenarios which should be taken care of in the next iteration of the simulation model.

The implemented solution supports both out–of–the–box methods which can be used by default and options for the customisation of the analysed model. Most importantly this library keeps the most important part of the Monte Carlo simulations – it provides the user with the meaningful output results which can be directly analysed and used in the algorithm optimisation. What is also important both papers related to networking and cited in Chapter 2.2 and article [24] can easily examine the proposed solutions with the use of MonteCarloSimulator.

As this solution answers all the needs which appeared during an implementation of Monte Carlo simulation library, it is not expected to be fixed in the near future. The source code is available on GitHub[1] and is reported in the official *ns-3* wiki page[2]. However with new requirements, some additional customisation options, which were not obvious during the initial implementation might be added in the future. The most obvious future work which should be done based on the results of this thesis is the complex analysis of the ε algorithms in small networks, which was outside the scope of this thesis.

[1]`https://github.com/chelminiak/MonteCarloSimulator`

[2]`https://www.nsnam.org/wiki/Contributed_Code`

A Library Source Code

```cpp
#include "MonteCarloSimulator.h"

#include "ns3/packet-sink.h"
#include "ns3/log.h"
#include "ns3/names.h"
#include "ns3/simulator.h"
#include "functional"

namespace ns3
{

NS_LOG_COMPONENT_DEFINE("MonteCarloSimulator");

MonteCarloSimulator::MonteCarloSimulator(ApplicationContainer*
                                             sinkApplications,
                                         double numberOfRounds,
                                         double roundTime,
                                         double roundWarmup,
                                         std::string outputName,
                                         uint32_t resultsPrinting,
                                         bool useDefaultRewardCalculation,
                                         std::function<void()>
                                             BehaviourFunction)
{
    sinks = sinkApplications;
    rounds = numberOfRounds;
    time = roundTime;
    warmup = roundWarmup;
    outputFileName = outputName + ".csv";
    printing = resultsPrinting;
    useDefaultCalculation = useDefaultRewardCalculation;
    if (useDefaultRewardCalculation)
    {
        Simulator::Schedule(Seconds(warmup),
                            &MonteCarloSimulator::GetWarmupStatistics,
                            this);
    }
    for (int round = 0; round < rounds + 1; ++round)
    {
        if (useDefaultRewardCalculation){
            Simulator::Schedule(Seconds((round + 1) * time),
                                &MonteCarloSimulator::
                                    DefaultRewardCalculation,
                                this);
            Simulator::Schedule(Seconds((round + 1) * time + warmup),
                                &MonteCarloSimulator::GetWarmupStatistics,
                                this);
            Simulator::Schedule(Seconds((round + 1) * time),
                                &MonteCarloSimulator::HandleResults,
                                this);
        }
        Simulator::Schedule(Seconds((round + 1) * time),
                            BehaviourFunction);
    }
    for (int i = 0; i < 100; ++i)
    {
        for (int j = 0; j < 500; ++j)
        {
            throughputArray[i][j] = 0;
            throughputSumArray[i] = 0;
            totalBytes[i] = 0;
            chooseArray[i] = 0;
            rewardArray[i][j] = 0;
        }
    }
}

bool
MonteCarloSimulator::FileExists(const std::string& filename)
{
    std::ifstream f(filename.c_str());
    return f.good();
}

void
MonteCarloSimulator::GetWarmupStatistics()
{
    for (uint32_t applicationIndex = 0; applicationIndex < sinks->GetN();
         ++applicationIndex)
    {
        totalBytes[applicationIndex] =
            DynamicCast<PacketSink>(sinks->
                                    Get(applicationIndex))->GetTotalRx();
    }
}

void
MonteCarloSimulator::DefaultRewardCalculation()
{
    uint64_t totalBytesThroughput[sinks->GetN()];
    for (uint32_t applicationIndex = 0; applicationIndex < sinks->GetN();
         ++applicationIndex)
    {
        totalBytesThroughput[applicationIndex] =
            DynamicCast<PacketSink>(sinks
                                        ->Get(applicationIndex))
                ->GetTotalRx();
        throughputArray[applicationIndex][currentRound] =
            (totalBytesThroughput[applicationIndex] -
             totalBytes[applicationIndex]) * 8 /
            ((time - warmup) * 1000000.0);
        totalBytes[applicationIndex] =
            totalBytesThroughput[applicationIndex];
    }
    for (uint32_t applicationIndex = 0; applicationIndex < sinks->GetN();
         ++applicationIndex)
    {
        if (throughputArray[applicationIndex][currentRound] > 0)
        {
            chooseArray[applicationIndex] += 1;
            throughputSumArray[applicationIndex] +=
                throughputArray[applicationIndex][currentRound];
            rewardArray[applicationIndex][currentRound] =
                throughputSumArray[applicationIndex] /
                chooseArray[applicationIndex];
        }
    }
}

void
MonteCarloSimulator::HandleResults()
```

```
{
    if (currentRound >= printing)
    {
        std::cout << "Results for round " << currentRound << ": "
                  << std::endl;
        for (uint32_t applicationIndex = 0;
             applicationIndex < sinks->GetN();
             ++applicationIndex)
        {
            std::cout << "Reward for application number "
                      << applicationIndex << ": "
                      << rewardArray[applicationIndex][currentRound]
                      << std::endl;
        }
    }
    std::ofstream outputFile;

    std::string outputCsv = outputFileName;
    if (FileExists(outputCsv))
    {
        // If the file exists, append to it
        outputFile.open(outputCsv, std::ios::app);
    }
    else
    {
        // If the file does not exist, create it and set the header line
        outputFile.open(outputCsv, std::ios::app);
        outputFile << "StageNumber";
        for (uint32_t applicationIndex = 0; applicationIndex <
                                            sinks->GetN();
             ++applicationIndex)
        {
            outputFile << ",Reward" << applicationIndex;
        }
        outputFile << std::endl;
    }

    outputFile << currentRound;
    for (uint32_t applicationIndex = 0;
         applicationIndex < sinks->GetN();
         ++applicationIndex)
    {
        outputFile << "," << rewardArray[applicationIndex][currentRound];
    }
    outputFile << std::endl;

    outputFile.close();
    currentRound += 1;
}

int*
MonteCarloSimulator::GetCurrentRound()
{
    return &currentRound;
}

double (*MonteCarloSimulator::GetRewardArray())[500]
{
    return rewardArray;
}

double *MonteCarloSimulator::GetChooseArray()
{
    return chooseArray;
}

u_int32_t *MonteCarloSimulator::GetTotalBytes()
{
    return totalBytes;
}

double (*MonteCarloSimulator::GetThroughputArray())[500]
{
    return throughputArray;
}

double *MonteCarloSimulator::GetThroughputSumArray()
{
    return throughputSumArray;
}

void
```

```
MonteCarloSimulator::SetRewardCalculationFunction
    (std::function<void()> RewardCalculationFunction)
{
    if (!useDefaultCalculation)
    {
        for (int round = 0; round < rounds + 1; ++round)
        {
            Simulator::Schedule(Seconds((round + 1) * time),
                                RewardCalculationFunction);
            Simulator::Schedule(Seconds((round + 1) * time),
                                &MonteCarloSimulator::HandleResults,
                                this);
        }
    }
}

void
MonteCarloSimulator::SetWarmupStatisticsCollectionFunction(
    std::function<void()> WarmupStatisticsFunction)
{
    if (!useDefaultCalculation)
    {
        for (int round = 0; round < rounds + 2; ++round)
        {
            Simulator::Schedule(Seconds((round)*time + warmup),
                                WarmupStatisticsFunction);
        }
    }
}

void
MonteCarloSimulator::SetEndConditionFunction
    (std::function<bool()> EndConditionFunction)
{
    for (int round = 0; round < rounds + 1; ++round)
    {
        Simulator::Schedule(Seconds((round + 1) * time),
                            [EndConditionFunction]()
                            {if (EndConditionFunction()){
                Simulator::Stop();
            }});
    }
}

}
```

B Simulation Scenario Source Code

```
#include "ns3/command-line.h"
#include "ns3/config.h"
#include "ns3/uinteger.h"
#include "ns3/boolean.h"
#include "ns3/double.h"
#include "ns3/string.h"
#include "ns3/log.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/ssid.h"
#include "ns3/mobility-helper.h"
#include "ns3/internet-stack-helper.h"
#include "ns3/ipv4-address-helper.h"
#include "ns3/udp-client-server-helper.h"
#include "ns3/packet-sink-helper.h"
#include "ns3/on-off-helper.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/packet-sink.h"
#include "ns3/yans-wifi-channel.h"
#include "ns3/propagation-loss-model.h"
#include "ns3/propagation-delay-model.h"
#include "ns3/abort.h"
#include "ns3/mobility-model.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/csma-module.h"
#include "ns3/config-store.h"
#include "iostream"
#include "algorithm"
#include "random"
#include "ns3/MonteCarloSimulator.h"

using namespace ns3;
using namespace std;

NS_LOG_COMPONENT_DEFINE ("thesis");

NodeContainer wifiApNodes;
NodeContainer wifiStaNodes;
ApplicationContainer sinkApplications;
std::string epsilonType = "sticky";
```

```
double epsilonValue = 0.3;
int stickyCounter = 2;
double stickyCounterArray[4];
double (*rewardArray)[500];
double (*throughputArray)[500];
int* roundNum;
int dataRate[2] = {12, 15};
bool stickyUsed[2] = {0, 0};

// Enable or disable network interfaces in order to switch
// station's association
void StatoAP1(Ptr<Node> sta){
    sta->GetObject<Ipv4>()->SetDown(2);
    sta->GetObject<Ipv4>()->SetUp(1);
}

void StatoAP2(Ptr<Node> sta){
    sta->GetObject<Ipv4>()->SetDown(1);
    sta->GetObject<Ipv4>()->SetUp(2);
}

void RandomizeConnection(Ptr<Node> wifiNode){
    std::random_device randomDevice;
    std::mt19937 gen(randomDevice());
    std::uniform_real_distribution<> dis(0.0, 1.0);
    double randomValue = dis(gen);
    if (randomValue < .5){
        StatoAP1(wifiNode);
    } else {
        StatoAP2(wifiNode);
    }
}

void EpsilonGreedy(Ptr<Node> wifiNode, uint32_t nodeIndex){
    // Generate random values with null-device as seed to obtain different
    // values in short timeframe
    std::random_device randomDevice;
    std::mt19937 gen(randomDevice());
    std::uniform_real_distribution<> dis(0.0, 1.0);
    double randomEpsilon = dis(gen);
    if (randomEpsilon < epsilonValue){
        RandomizeConnection(wifiNode);
    } else
    {
        if (rewardArray[2 * nodeIndex][*roundNum] >
            rewardArray[2 * nodeIndex + 1][*roundNum])
        {
            StatoAP1(wifiNode);
        }
        else
        {
            StatoAP2(wifiNode);
        }
    }
}

void ChooseAP(){
    if (epsilonType == "sticky")
    {
        int currentAssociation[2] = {0, 0};
        if (throughputArray[0][*roundNum-1] >
        throughputArray[1][*roundNum-1])
        {
        currentAssociation[0] = 0;
        }
        else
        {
        currentAssociation[0] = 1;
        }
        if (throughputArray[2][*roundNum-1] >
        throughputArray[3][*roundNum-1])
        {
        currentAssociation[1] = 2;
        }
        else
        {
        currentAssociation[1] = 3;
        }
        for (uint32_t nodeIndex = 0; nodeIndex < wifiStaNodes.GetN();
             ++nodeIndex)
        {
        if (stickyUsed[nodeIndex])
        {
            if (throughputArray[currentAssociation[nodeIndex]]
                                [*roundNum-1] >
                dataRate[nodeIndex] * .996)
            {
                stickyCounterArray[currentAssociation[nodeIndex]] =
                    stickyCounter;
            }
            else
            {
                stickyCounterArray[currentAssociation[nodeIndex]] =
                    stickyCounterArray[currentAssociation[nodeIndex]]
                    - 1;
            }
        }
        else
        {
            if (throughputArray[currentAssociation[nodeIndex]]
                                [*roundNum-1] >
                dataRate[nodeIndex] * .996)
            {
                stickyCounterArray[currentAssociation[nodeIndex]] =
                    stickyCounter;
            }
        }
        if (stickyCounterArray[currentAssociation[nodeIndex]] > 0)
        {
            stickyUsed[nodeIndex] = 1;
        }
        else
        {
            EpsilonGreedy(wifiStaNodes.Get(nodeIndex), nodeIndex);
        }
        }
    }
    else if (epsilonType == "greedy")
    {
        for (uint32_t nodeIndex = 0; nodeIndex < wifiStaNodes.GetN();
             ++nodeIndex)
        {
        EpsilonGreedy(wifiStaNodes.Get(nodeIndex), nodeIndex);
        }
    }
    else
    {
        for (uint32_t nodeIndex = 0; nodeIndex < wifiStaNodes.GetN();
             ++nodeIndex)
        {
            RandomizeConnection(wifiStaNodes.Get(nodeIndex));
        }
    }
}


int main (int argc, char *argv[]){
    double roundTime = 2;
    double numRounds = 10;
    double roundWarmup = 1;
    uint32_t printing = 0;
    std::string outputName = "example-algorithms";

    CommandLine cmd;
    cmd.AddValue("roundTime", "Duration of single round", roundTime);
    cmd.AddValue("numRounds", "Number of rounds", numRounds);
    cmd.AddValue("printing", "Number of stage from which results "
                             "will be printed", printing);
    cmd.AddValue("roundWarmup", "Warmup time for each round", roundWarmup);
    cmd.AddValue("epsilonType", "Type of epsilon algorithm. Available "
                                "types: none, greedy, sticky", epsilonType);
    cmd.AddValue("epsilonValue", "Value of epsilon parameter",
                 epsilonValue);
    cmd.AddValue("stickyCounter", "Sticky counter, used in "
                                  "epsilon sticky algorithm",stickyCounter);
    cmd.AddValue("outputName", "Name of the output file with results",
                 outputName);
    cmd.Parse (argc,argv);

    if (roundWarmup >= roundTime){
        std::cout << "Warmup time cannot exceed time of a single round"
                  << std::endl;
        return 1;
    }

    if (!(epsilonType == "none" || epsilonType == "greedy" ||
          epsilonType == "sticky")){
        std::cout << "Unsupported epsilon algorithm" << std::endl;
        return 1;
    }

    if (epsilonValue < 0 || epsilonValue > 1){
        std::cout << "Epsilon parameter should be in range <0, 1>"
                  << std::endl;
        return 1;
    }

    // Zero values in used arrays
    for (int i = 0; i < 4; ++i){
        stickyCounterArray[i] = 0;
    }


    // Create AP and stations
    wifiApNodes.Create(2);
    wifiStaNodes.Create(2);

    // Configure mobility
    MobilityHelper mobility;
    Ptr<ListPositionAllocator> positionAlloc =
        CreateObject<ListPositionAllocator> ();

    positionAlloc->Add (Vector (0.0, 2.0, 0.0)); // position of AP1
    positionAlloc->Add (Vector (6.0, 2.0, 0.0)); // position of AP2
    positionAlloc->Add (Vector (2.0, 5.0, 0.0)); // position of station 1
    positionAlloc->Add (Vector (2.5, 0.0, 0.0)); // position of station 2

    mobility.SetPositionAllocator (positionAlloc);

    mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");

    mobility.Install (wifiApNodes);
    mobility.Install (wifiStaNodes);

    // Configure wireless channel
    Ptr<MatrixPropagationLossModel> lossModel1;
    Ptr<MatrixPropagationLossModel> lossModel2;
    Ptr<YansWifiChannel> wifiChannel1 = CreateObject<YansWifiChannel>();
    Ptr<YansWifiChannel> wifiChannel2 = CreateObject<YansWifiChannel>();
    WifiPhyStateHelper phyStateHelper;
    lossModel1 = CreateObject<MatrixPropagationLossModel>();
    lossModel1->SetDefaultLoss(0);
    lossModel2 = CreateObject<MatrixPropagationLossModel>();
    lossModel2->SetDefaultLoss(0);

    // Create & setup wifi channel
    wifiChannel1->SetPropagationLossModel(lossModel1);
    wifiChannel1->SetPropagationDelayModel
        (CreateObject<ConstantSpeedPropagationDelayModel>());

    wifiChannel2->SetPropagationLossModel(lossModel2);
    wifiChannel2->SetPropagationDelayModel
        (CreateObject<ConstantSpeedPropagationDelayModel>());

    YansWifiPhyHelper phy;
    phy.SetChannel (wifiChannel1);

    // Set channel width for given PHY
    phy.Set("ChannelWidth", UintegerValue(20));

    // Set channel number for AP1
    phy.Set("ChannelNumber", UintegerValue(36));

    // Prepare MCS values, which will be used for Data and Control Modes
    // in Wi-Fi network
    WifiMacHelper mac;
    WifiHelper wifi;
    wifi.SetStandard (WIFI_STANDARD_80211ax);

    std::string oss3 = "HeMcs3";
    std::string oss4 = "HeMcs4";
    std::string oss5 = "HeMcs5";
    std::string oss6 = "HeMcs6";
    wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager","DataMode",
                                  StringValue (oss3), "ControlMode",
                                  StringValue (oss3)); //Set MCS

    Ssid ssid = Ssid ("toy-scenario-AP1"); //Set SSID
    mac.SetType ("ns3::AdhocWifiMac");

    // Connect nodes to Wi-Fi network
    NetDeviceContainer apDevices1;
    apDevices1 = wifi.Install (phy, mac, wifiApNodes.Get(0));

    //    mac.SetType ("ns3::StaWifiMac", "Ssid", SsidValue (ssid));
    NetDeviceContainer staDevices1;
    staDevices1 = wifi.Install (phy, mac, wifiStaNodes.Get(0));

    wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager","DataMode",
                                  StringValue (oss6), "ControlMode",
                                  StringValue (oss6)); //Set MCS
    staDevices1.Add(wifi.Install(phy, mac, wifiStaNodes.Get(1)));

    phy.Set("ChannelNumber", UintegerValue(44));
    phy.SetChannel (wifiChannel2);

    NetDeviceContainer apDevices2;
    apDevices2 = wifi.Install (phy, mac, wifiApNodes.Get(1));

    wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager","DataMode",
                                  StringValue (oss4), "ControlMode",
                                  StringValue (oss4)); //Set MCS
    NetDeviceContainer staDevices2;
    staDevices2 = wifi.Install (phy, mac, wifiStaNodes.Get(0));

    wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager","DataMode",
                                  StringValue (oss5), "ControlMode",
                                  StringValue (oss5)); //Set MCS
    staDevices2.Add(wifi.Install(phy, mac, wifiStaNodes.Get(1)));

    // Set guard interval on all interfaces of all nodes
    Config::Set ("/NodeList/*/DeviceList/*/$ns3::WifiNetDevice/"
                 "HeConfiguration/GuardInterval",
                 TimeValue (NanoSeconds (800)));

    // Install an Internet stack
    InternetStackHelper stack;
    stack.Install (wifiApNodes);
    stack.Install (wifiStaNodes);

    // Configure IP addressing for Wi-Fi
    Ipv4AddressHelper address;
    address.SetBase ("192.168.1.0", "255.255.255.0");
    Ipv4InterfaceContainer staNodeInterface;
    Ipv4InterfaceContainer apNodeInterface;

    staNodeInterface = address.Assign (staDevices1);
    apNodeInterface = address.Assign (apDevices1);

    address.SetBase ("192.168.2.0", "255.255.255.0");

    staNodeInterface.Add(address.Assign (staDevices2));
    apNodeInterface.Add(address.Assign (apDevices2));

    // Set transmitted power for all Station nodes in the network
    Config::Set ("/$ns3::NodeListPriv/NodeList/2/$ns3::Node/DeviceList/0/"
                 "$ns3::WifiNetDevice/Phy/$ns3::YansWifiPhy/TxPowerEnd",
                 DoubleValue(-79.5)); //-76.68
    Config::Set ("/$ns3::NodeListPriv/NodeList/2/$ns3::Node/DeviceList/0/"
                 "$ns3::WifiNetDevice/Phy/$ns3::YansWifiPhy/TxPowerStart",
                 DoubleValue(-79.5));
    Config::Set ("/$ns3::NodeListPriv/NodeList/2/$ns3::Node/DeviceList/1/"
                 "$ns3::WifiNetDevice/Phy/$ns3::YansWifiPhy/TxPowerEnd",
                 DoubleValue(-76.45)); //-79.85
    Config::Set ("/$ns3::NodeListPriv/NodeList/2/$ns3::Node/DeviceList/1/"
                 "$ns3::WifiNetDevice/Phy/$ns3::YansWifiPhy/TxPowerStart",
                 DoubleValue(-76.45));
    Config::Set ("/$ns3::NodeListPriv/NodeList/3/$ns3::Node/DeviceList/0/"
                 "$ns3::WifiNetDevice/Phy/$ns3::YansWifiPhy/TxPowerEnd",
                 DoubleValue(-71.1)); //-72.52
    Config::Set ("/$ns3::NodeListPriv/NodeList/3/$ns3::Node/DeviceList/0/"
                 "$ns3::WifiNetDevice/Phy/$ns3::YansWifiPhy/TxPowerStart",
                 DoubleValue(-71.1));
    Config::Set ("/$ns3::NodeListPriv/NodeList/3/$ns3::Node/DeviceList/1/"
                 "$ns3::WifiNetDevice/Phy/$ns3::YansWifiPhy/TxPowerEnd",
                 DoubleValue(-72.3)); //-76.53
    Config::Set ("/$ns3::NodeListPriv/NodeList/3/$ns3::Node/DeviceList/1/"
                 "$ns3::WifiNetDevice/Phy/$ns3::YansWifiPhy/TxPowerStart",
            DoubleValue(-72.3));

    // Install applications (traffic generators)
    ApplicationContainer sourceApplications;
    uint32_t portNumber = 9;
    for (uint32_t staIndex = 0; staIndex < 2; ++staIndex){
        for (uint32_t APindex = 0; APindex < 2; ++APindex)
        {
            auto ipv4 = wifiApNodes.Get(APindex)->GetObject<Ipv4>();
            const auto socketAddress = ipv4->GetAddress (1, 0)
                                            .GetLocal ();
            InetSocketAddress sinkSocket (socketAddress, portNumber++);
            OnOffHelper onOffHelper ("ns3::UdpSocketFactory", sinkSocket);
            onOffHelper.SetConstantRate(DataRate(dataRate[staIndex]
                                                  * 10e6),1472);
            sourceApplications.Add (onOffHelper.Install (
                wifiStaNodes.Get (staIndex)));
            PacketSinkHelper packetSinkHelper ("ns3::UdpSocketFactory",
                                               sinkSocket);
            sinkApplications.Add (packetSinkHelper.Install
                                 (wifiApNodes.Get (APindex)));
        }
    }

    // Initialize simulation at random
    RandomizeConnection(wifiStaNodes.Get(0));
    RandomizeConnection(wifiStaNodes.Get(1));

    MonteCarloSimulator monteCarloSimulator = MonteCarloSimulator(
        &sinkApplications, numRounds, roundTime, roundWarmup,
        outputName, printing,true,
        &ChooseAP);
    rewardArray = monteCarloSimulator.GetRewardArray();
    throughputArray = monteCarloSimulator.GetThroughputArray();
    roundNum = monteCarloSimulator.GetCurrentRound();

    sinkApplications.Start (Seconds (0.0));
    sinkApplications.Stop (Seconds ((numRounds + 1) * roundTime));
    sourceApplications.Start (Seconds (0.0));
    sourceApplications.Stop (Seconds ((numRounds + 1) * roundTime));

    // Define simulation stop time
    Simulator::Stop (Seconds ((numRounds + 1) * roundTime));

    // Print information that the simulation will be executed
    std::clog << std::endl << "Starting simulation... " << std::endl;

    // Run the simulation!
    Simulator::Run ();

    //Clean-up
    Simulator::Destroy ();

    return 0;
}
```

Bibliography

[1] A. Barbu and Song-Chun Zhu. Introduction to Monte Carlo Methods. In Monte Carlo Methods, pages 1–17. Springer Singapore, Singapore, First Edition edition, 2020. ISBN 978-981-13-2970-8. doi:10.1007/978-981-13-2971-5.

[2] R. L. Harrison. Introduction to Monte Carlo Simulation. AIP Conference Proceedings 5, 1204:17–21, 2010. doi:10.1063/1.3295638.

[3] S. Raychaudhuri. Introduction to Monte Carlo simulation. 2008 Winter Simulation Conference, 2008. doi:10.1109/WSC.2008.4736059.

[4] Amazon Web Services. What is the monte carlo simulation? URL `https://aws.amazon.com/what-is/monte-carlo-simulation/`.

[5] I. Pinkovetskaia, Y. Nuretdinova, I. Nuretdinov, and N. Lipatova. Mathematical modeling on the base of functions density of normal distribution. Journal of the University of Zulia, 12 (33):34–49, 2021. doi:10.1109/WSC.2008.4736059.

[6] C. Rondero-Guerrero, I. González-Hernández, and C. Soto-Campos. An extended approach for the generalized powered uniform distribution. Computational Statistics, pages 1–24, 2022. doi:10.1007/s00180-022-01296-3.

[7] M. S. Mirković. Triangular distribution and PERT method vs. payoff matrix for decision-making support in risk analysis of construction bidding: A case study. Facta universitatis - series: Architecture and Civil Engineering, 18:287–307, 2020. doi:10.2298/FUACE201117020M.

[8] B. Lutkevich. What is the monte carlo simulation? URL `https://www.techtarget.com/searchcloudcomputing/definition/Monte-Carlo-simulation`.

[9] V A Voevodin and D S Burenok and V S Cherniaev. Monte Carlo method for solving the problem of predicting the computer network resistance against DoS attacks. Journal of Physics: Conference Series, 2099(1):012069, 2021. doi:10.1088/1742-6596/2099/1/012069.

[10] H. Jie, L. Kezhong, and S. Huajie. Reliability Analysis of Swarm Self-security Intelligence System Based on Fault Tree and Monte Carlo Simulation. 2022 21st International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES), Chizhou, China, pages 184–187, 2022. doi:10.1109/DCABES57229.2022.00054.

[11] J. Wang, M. Neil, and N. Fenton. A Bayesian network approach for cybersecurity risk assessment implementing and extending the FAIR model. Computers and Security, 89: 101659, 2020. doi:10.1016/j.cose.2019.101659.

[12] W. Wang and A. Cammi and F. Di Maio and S. Lorenzi and E. Zio. A Monte Carlo-based exploration framework for identifying components vulnerable to cyber threats in nuclear power plants. Reliability Engineering and System Safety, 175:24–37, 2018. doi:10.1016/j.ress.2018.03.005.

[13] T. Fagade and K. Maraslis and T. Tryfonas. Towards Effective Cybersecurity Resource Allocation: The Monte Carlo Predictive Modelling Approach. International Journal of Critical Infrastructures, 13(2-3):152–167, 2017. doi:10.1504/IJCIS.2017.088235.

[14] D. Sarrut, A. Etxebeste, E. Muñoz, N. Krah, and J. M. Létang. Corrigendum: Artificial Intelligence for Monte Carlo Simulation in Medical Physics. 2021. doi:10.3389/fphy.2021.738112.

[15] H. Sahiner and X. Liu. Gamma spectral analysis by artificial neural network coupled with Monte Carlo simulations. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 953:163062, 2020. doi:10.1016/j.nima.2019.163062.

[16] F. Verhaegen, J. Seco. Artificial Intelligence and Monte Carlo Simulation. In Monte Carlo Techniques in Radiation Therapy, pages 251–259. CRC Press, Boca Raton, Second Edition edition, 2021. doi:10.1201/9781003211846.

[17] J. O. Rico-Contreras and A. A. Aguilar-Lasserre and J. M. Méndez-Contreras and J. J. López-Andrés and G. Cid-Chama. Moisture content prediction in poultry litter using artificial intelligence techniques and Monte Carlo simulation to determine the economic yield from energy use. Journal of Environmental Management, 202:254–267, 2017. doi:10.1016/j.jenvman.2017.07.034.

[18] R. Pirayeshshirazinezhad and S. G. Biedroń and J. A. D. Cruz and S. S. Güitrón and M. Martínez-Ramón. Designing Monte Carlo Simulation and an Optimal Machine Learning to Optimize and Model Space Missions. IEEE Access, 10:45643–45662, 2022. doi:10.1109/ACCESS.2022.3170438.

[19] S. Hande, P. Patidar, S. Meena, and S. Banerjee. Network flow Optimization through Monte Carlo Simulation. 2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC), Solan, India, 2018. doi:10.1109/PDGC.2018.8745965.

[20] J. Konorski and S. Szott. Mitigating Traffic Remapping Attacks in Autonomous Multi-hop Wireless Networks. IEEE Internet of Things Journal, 9, no. 15:13555–13569, 2022. doi:10.1109/JIOT.2022.3143713.

[21] A. M. Voicu, L. Simić, and M. Petrova. Modelling Broadband Wireless Technology Coexistence in the Unlicensed Bands. IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), Pisa, Italy, pages 129–138, 2021. doi:10.1109/WoWMoM51794.2021.00026.

[22] P. Y. Omole-Matthew, G. J. Arome, A. Favour-Berthy Thompson, and B. K. Alese. Monte Carlo Simulation Approach to Network Access Control. Journal of Internet Technology and Secured Transactions, 9:726–729, 2021. doi:10.20533/jitst.2046.3723.2021.0088.

[23] A. M. Voicu, L. Simić, and M. Petrova. Modelling Large-Scale CSMA Wireless Networks. 2021 IEEE Wireless Communications and Networking Conference (WCNC), Nanjing, China, pages 1–6, 2021. doi:10.1109/WCNC49053.2021.9417319.

[24] M. Carrascosa and B. Bellalta. Multi-Armed Bandits for Decentralized AP selection in Enterprise WLANs. Computer Communications, 159:108–123, 2020. doi:10.1016/j.comcom.2020.05.023.

[25] G. F. Riley and T. R. Henderson. The *ns-3* network simulator. In K. Wehrle, M. Güneş, and J. Gross, editors, Modeling and Tools for Network Simulation, chapter 2, pages 15–34. Springer, Berlin, Heidelberg, 2010. doi:10.1007/978-3-642-12331-3_2.