Implementation of predictive algorithm for minimizing the number of hand-offs in mobile wireless networks

Julio Navarro Lara

UCLA EE Student

Under the University of Granada Exchange Program

Under the mentorship of M.S. Pooya Monajemi

Advisor: *Prof. John Villasenor*

1. Preface

This project was developed at the UCLA Electrical Engineering Image Processing lab under Professor John Villasenor supervision. It was developed during the Winter and Spring quarters, and I had to turn in two reports in that period to the UCLA EE department, one at the end of each quarter.

This report is the final capitulation and it includes information about the complete process, the result of joining those two reports already turned in.

2. Introduction

Along with the introduction of femtocells, also called Home NodeB's, to the cellular market multiple new aspects arise in problems that are considered solved in the conventional macrocell networks. This work addresses the issue of hand-off management, which has been classically approached in multiple different ways.

The state of the art approaches have mostly focused on issues such as continuity of service, load balancing between base stations, and providing minimum QoS requirements.

The classical method for managing the hand-offs in mobile (cellular) networks is choosing the Base Station perceived with higher power by the Mobile Station. This method does not take into account the number of hand-offs made by the Mobile Station, an important figure for optimizing the performance of the Mobile Station.

We aim to address a specific issue that arises in the context of femtocell deployments, which is the possibility of large and unnecessary overhead in a network

where femtocells are densely deployed. We propose a predictive algorithm with the aim of minimizing the number of hand-offs in such scenarios. This performance measure is used to compare the proposed algorithm against a primitive method for managing the hand-offs in mobile networks, namely connecting at any time to the Base Station perceived with higher power by the Mobile Station, as mentioned earlier. We will refer to this primitive scheme in this report with the name of Instantaneous Pilot.

Even though the first goal was implementing this new predictive algorithm in scenarios where femtocells are deployed, the results can be totally applied in scenarios with any kind of base stations. The coverage radius of the base stations is not an issue, and the algorithm is independent to this parameter.

Actually, another kind of scenario where it can be very effective to apply this predictive scheme, as the reader could notice as he go deeper in the analysis of this algorithm, is one with a high density of base stations, with the coverage areas overlapping between them.

Work in predictive algorithms for mobile networks is not extensive. In many of the studies prediction is used as a means to pre-allocate resources on base stations prior to the arrival of a mobile station into the cell and the prediction is done entirely by the Base Station [1] [2] [3].

However, in our approach the predictive algorithm is executed in the Mobile Station itself, assuming necessary information has been communicated between the network and the Mobile Station that allows for computation to be performed.

Additionally, the mobile station’s positioning ability can facilitate the prediction of the user trajectory. The mobile devices are becoming smarter by the day, and most of

them have the feature of positioning. Our technique is aided by real world data about road topology, road-traffic statistics and vehicle speeds.

An important measure of call quality is minimal call drop probability. Incorporated in the model is a minimum receive power requirement, which is aided as additional constraints to the algorithm to ensure that any subsequent hand-offs will be providing the minimum QoS requirement with a high probability.

The theoretical framework of our algorithm is based on Markov model. This model is applied to a statistical traffic model incorporating road information in a scenario, whose efficiency will be proved in simulation.

First we are going to introduce the theory behind this algorithm and then we will explain the algorithm step by step, introducing examples for easing the understanding. Finally, we will present some results from the simulations we runned.

3. Theory

The final objective of our technique is minimizing the number of hand-offs while maintaining the QoS required for a decent communication.

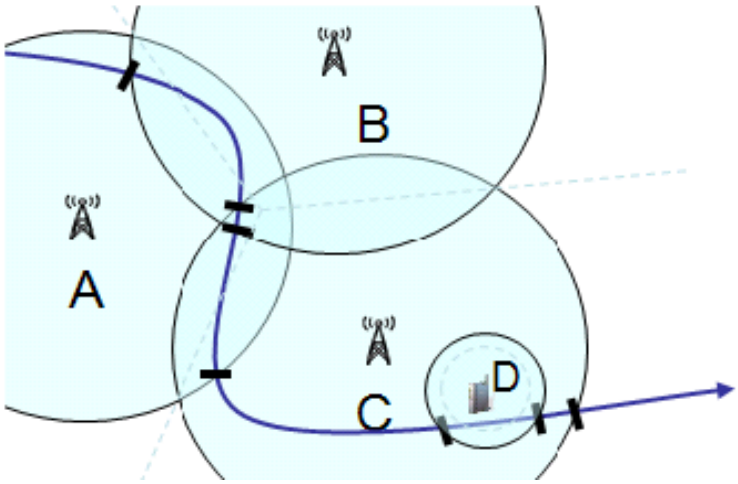


Figure 1. Example of a Scenario

In Figure 1 we can see a simple scenario where the path followed by the Mobile Station is known and indicated by the arrow. A, B, C and D are the areas under the coverage of certain Base Stations where the QoS is satisfied. In the overlapping areas, the QoS is satisfied by more than one Base Station. The QoS threshold will be a certain value of signal strength.

The first step is dividing the path into N segments, each of them associated with a determined set of Base Stations. We have 6 segments along the Mobile Station path. Our goal is to choose the best combination of Base Stations to minimize hand-offs.

With the conventional hand-off technique of choosing the most powerful signal, the Base Station sequence would be as follows: A-B-A-C-D-C. However, the optimal assignment for minimizing the number of Base Station changes would be A-A-A-C-C-C.

In the following subsection we are trying to explain all the mathematical formulation that will lie behind our algorithm.

3.1. Mathematical formulation. Continuous case

We can model the motion of the Mobile Station as a continuous-time Markov process. Hence, the probability density of the future paths is predicted given its recent history of locations and speeds as:

$$f\big(x(\tau, t < \tau < t + T)\big) = f\big(x(\tau, t < \tau < t + T) \mid x(\tau', t - T_p < \tau' < t)\big)$$

We want to minimize the number of hand-offs for the future T units of time, choosing the proper Base Station:

$$k^* = \arg\min_{k \in B} \left\{ \mathrm{E}_{p \in \phi} \left[\Lambda\big(r_{opt}(p \mid r_1 = k)\big) \right] \right\}$$

where,

B is the set of all Base Stations,

p represents one path,

ϕ is the set of all predicted paths for the future T_f seconds,

$\Lambda(r)$ indicates the number of hand-offs in a given sequence of Base Station assignments *r*,

$r_{opt}(p \mid r_1 = k)$ is the optimal sequence of Base Station assignments conditioned on the first Base Station being *k*:

$$r_{opt}(p \mid r_1 = k) = \arg\min_{r \in R_p \mid r_1 = k} \left\{ \Lambda\left(r_1 = k, r_2, \dots, r_{N_p}\right) \right\}$$

where,

R_p represents the set of permissible Base Station assignments for path *p*

$$R_p = \left\{ r \mid r_n \in \Gamma_{p_n}, \forall n: 1 \le n \le N_p \right\}$$

N_p represents the number of segments in path *p*

Γ_{p_n} is the set of Base Stations meeting the QoS target for the nth segment of path *p*

The continuous time formulation is over-constrained, due to the significantly change in the environment of the Mobile Station. We can consider this change is not continuous and just happens after discrete intervals of time.

3.2. Discrete Approach

We are using a discretized model because the continuous space-time model is computationally intensive and not very applicable to real systems. In this case, we divide the space into tiles in which a known set of Base Stations will meet the QoS requirements. Each path is divided into segments according to the tiles.

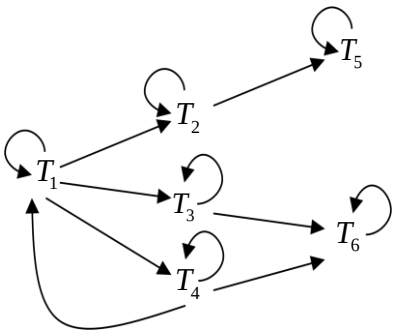


Figure 2. Tile graph based on Markov model (discrete time)

In Figure 2 we have the state graph generated using the Markov model, with each node representing the presence of the Mobile Station in a tile, and each arrow representing a tile crossing event (or lack of it) in ΔT seconds with a known probability. For all states there is a non-zero probability of staying in the same tile (no transition). The depth of the graph or number of stages in each branch theoretically determines the trade-off between better accuracy of prediction and computational complexity.

The model is now as follows:

$$k^* = \arg\max_{k \in B} \left\{ \mathrm{E}_{p \in \Phi} \left[I\left(r_1^{opt}(p) = k \right) \right] \right\}$$

where,

B is the set of all Base Stations,

p represents one path along the graph,

Φ is the set of all predicted paths over the graph with a length of $T_f/\Delta T$

I is the indicator function,

$r_1^{opt}(p)$ is the first Base Station in the optimal sequence of Base Station assignments for path p

$$r_{opt}(p) = \arg\min_{r \in R_p} \left\{ \wedge \left(r_1, r_2, \dots, r_{N_p} \right) \right\}$$

where,

R_p represents the set of permissible Base Station assignments for path p

$$R_p = \left\{ r \mid r_n \in \Gamma_{p_n}, \forall n: 1 \leq n \leq N_p \right\}$$

N_p represents the number of nodes in path p, $T_f/\Delta T$

Γ_{p_n} is the set of Base Stations meeting the QoS target for the nth node of path p

3.3. Presence Constraint

The constraint in this decision making process is QoS, which as we said is based on signal power received by the mobile station. We have another constraint when applying the predictive algorithm that we can call "Presence Constraint" and is based on the probability of a Base Station being supported in the next step of the prediction. When we choose a Base Station to hand over to, we check to make sure it has a large enough likelihood to be present in the next node of the tree. This constraint for the continuous time model is given by:

$$P_{QoS} = \mathop{\mathrm{E}}_{p \in \Phi} [I(k \in \Gamma(p_1))]$$

, where,

P_{QoS} is the probability of meeting the QoS target in terms of presence,

I represents the indicator function,

Γ is the set of Base Stations that meet the QoS target in a given path segment,

p_1 is the first segment of the path p.

4. Simulation program

For implementing this algorithm and obtaining some results about its performance, we used a simulation program, already in development when I arrived to the lab.

The program uses two programming languages: Matlab and C++. The Matlab part is used for the user interface. That includes the insertion of all the parameters needed for configuring the simulation, the creation of the simulation scenario and the visualization of the results. C++ is used for the simulation itself, processing all the movement of the Mobile Stations, the fading and interferences of signals, the scheduling, the hand-off schemes, etc. The interface between the two codes is text files, from where the configuration parameters and the results can be directly read.

You can see the present configuration window in Figure 3. The fields are described below:

- **General:** Here we can change the number of *time slots* of the simulation (each one corresponding to 1 millisecond). We can also choose if we want roaming Mobile Stations, roads or verbose output. It is recommended to uncheck this option in long simulations, for avoiding memory overflow. Anyways, we have also the option of change the "output sampling", which is the period between all the verbose information is gathered.

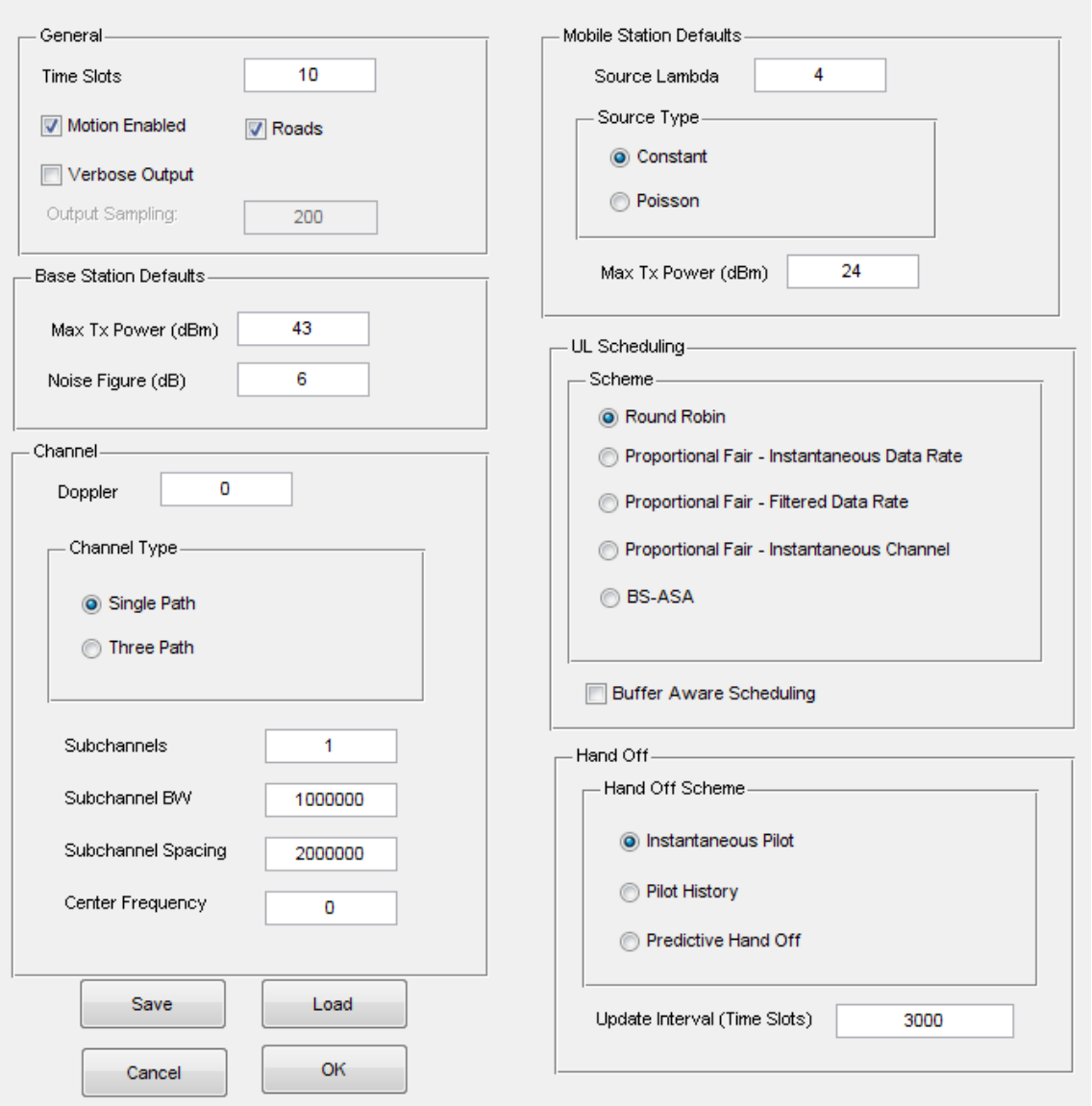


Figure 3. Configuration window of the simulation program

- **Base Station Defaults:** Here we can fix two parameters of the Base Stations: its *transmitted power* and its *noise figure*.
- **Channel:** In this section we can modify how the channel behaves: its *doppler effect*, its *type* (single path or three paths), the *number of subchannels*, their *bandwidth* and *spacing*, and the *center frequency*.

- **Mobile Station Defaults:** We can configure here the average rate of packet transmission in the Mobile Stations (*Source Lambda*), its *maximum power* and if the source is constant or Poisson type.
- **UL Scheduling:** This part is for selecting the Scheduling Scheme and if we want to simulate with the Buffer Aware option. We will always use Round Robin without Buffer Aware. The interest of this report is focused in the hand-off schemes and has nothing to do with the scheduling, so we are not going to spend more time explain the details of this.
- **Hand Off:** This last option allows us to choose the hand-off scheme used in the simulation. *Instantaneous Pilot* just choose the Base Station with higher power in that location, *Pilot History* is similar to the first one but with more considerations (not used here) and *Predictive Hand Off* is our new predictive algorithm. We can also set the number of time slots between every hand-off update.

Apart from this, we can change manually many other configuration parameters, in the text files that work as interface between Matlab and C++.

After clicking "Ok" in this window, we can create the scenario in the next one. That window appears in Figure 4, with the scenario that we used for obtaining the results to analyze in this report, *Scn_15May*. Of course, we tried many different scenarios during the development of the project.

In it we can draw roads of different density (freeways, streets and small streets), place the Mobile Stations (in blue) and the Base Stations (in black). We have run simulations also with Femto Base Stations, but they will be shown later in this report.

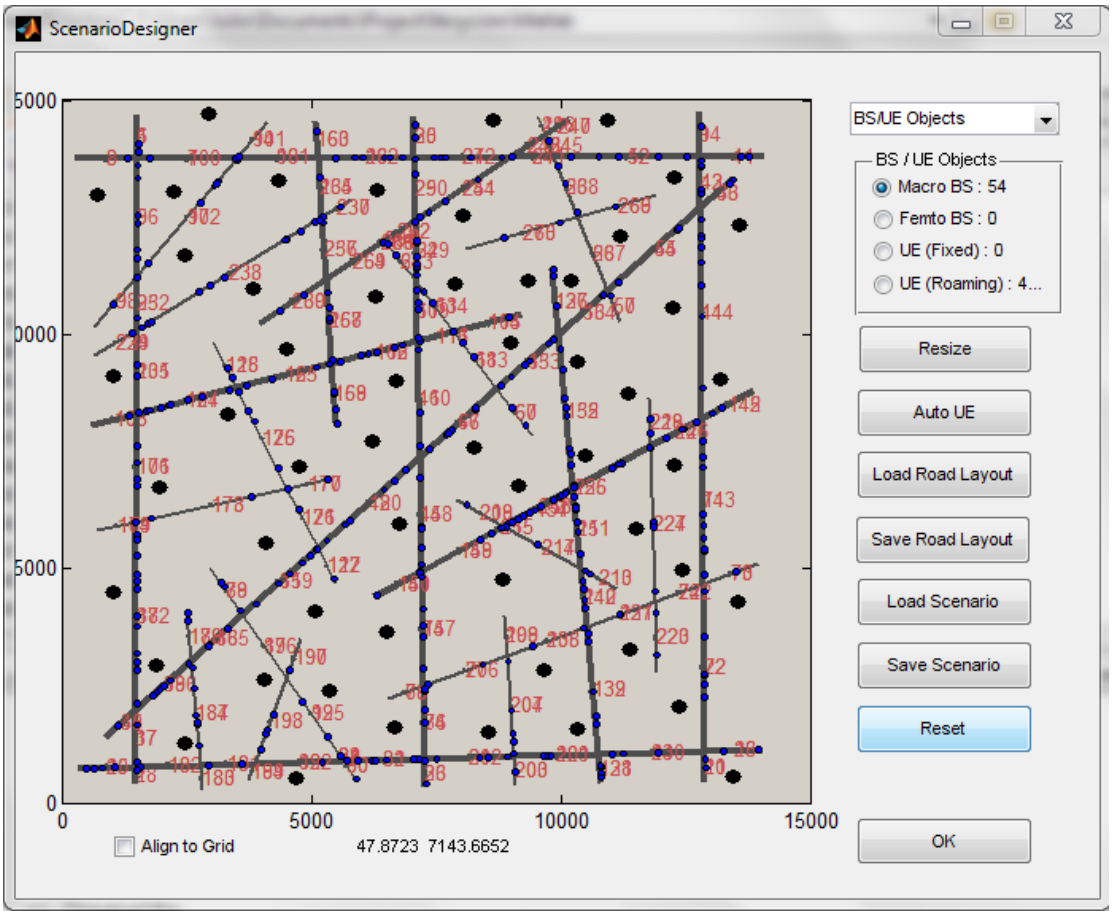


Figure 4. Design window with the scenario Scn_15May loaded

The streets are defined by their extreme points (start and end), speed, next roads and their probabilities and ID. When a road is drawn, both directions are included in the scenario, but the two are considered as different roads. Moreover, when the Base Stations are included and we add more roads intersecting the existent ones, the roads are split in segments, associated with the tiles considered in the predictive algorithm theory.

The probabilities for the next roads are determined according to the road type, related with the traffic density. The roaming Mobile Stations, which are moving along the roads, will change their trajectory according to the probabilities assigned to the roads.

During run-time, the program updates each of the entities (Mobile Stations, Base Stations, channel, etc.) at each time slot. The hand-off scheme will be executed

according to the selected interval. The time at which every Mobile Station starts the hand-off process is randomly distributed for avoiding them doing it at the same time.

5. Improving the simulator

As a I mentioned earlier, the mobile simulator I used during the project was already being used in the lab when I arrived. I had to develop my project over that tool, so I spent some time becoming familiar with it. Soon, I realized some of the simulation actions were done incorrectly, like simulation of the movement of the users.

As my advisor told me, the last person working on it left her work incomplete as she left the lab. So I realized that my first task before starting my project had to be improving the program so I could try the algorithm I was going to develop and see if it could work in real life or not. The simulator was a totally required tool, so if I could not fix the existent one, I would have to develop a new one.

In this section I am going to describe my work in the code of the simulation program for improving it beside the core of my work, the predictive algorithm. I have chosen to split the section in tasks instead of following a logbook format with chronological order for easing the reading and comprehension of the entire job done. For example, the motion of the road and the method for predicting the motion of the Mobile Stations were done simultaneously, but they are considered in different subsections.

I should say that I am going only to talk about the main work I did by myself. Apart from it, I have been helping Monajemi in many small tasks related to the code and the final results.

During the development of the project, I work mostly on the core of the simulator, the C++ code, because it was the part I needed for implementing the predictive

algorithm. Monajemi meanwhile continued fixing the Matlab part, the interface with the user, trying to solve all the problems we found on it so I could keep developing and trying the algorithm.

5.1.First Steps

As I mentioned, the first thing I had to do when I arrived to the lab was, of course, becoming familiar with the program used for simulate the mobile networks. It is a highly complex program with a lot of classes and dependences between them, as we saw in Section 0, so it is not straight-forward to start working on it.

I had never programmed in C++ before, but I had a good background in programming routers and microcontrollers in C, so it was not difficult to update my programming knowledge. Moreover, I had been coding in Java since I started the University and I was very familiar with object-oriented programming. For updating myself in all the changes from C to C++, I read [4] the week before starting.

I was supposed to work just in the C++ code, in the simulation itself. As we saw in Section 0, the Matlab code is used for creating the interface to configure the simulation and it was going to be managed by Monajemi.

My first task was testing the program and fixing any bug that could happen during the simulation. I spent around one and a half week in this and it was a great way for learning how the simulation worked.

I fixed several bugs during this process, all of them minor errors. Most of them were solved just changing one line of code, but they were difficult to find and I spent a lot time debugging step by step.

As an example, some of the method for updating simulation variables did not consider the difference between roaming mobile station and fixed ones, or the initialization of some objects was not done correctly.

Another example is related with the generation of random number for creating Poisson traffic in the mobile station. The reason was the rounding of float numbers converted to integers, which made us lose all the information contained in it.

After this, I checked that all the scheduling and hand-off schemes (except the Predictive Handoff, which would become my main contribution to the program) were working properly. For that, I modified the code for making it print the Base Station history of each Mobile Station as part of the results and I run simple scenarios. Doing this I could check the path of the Mobile Stations step by step and it was easier to detect general bugs.

All the schemes were properly working, but I realized I could not say the same about the motion of the Mobile Stations.

5.2. Mobile Stations' Motion

In the program, when you add a road to the scenario, you are actually adding four roads instead: one for each direction and two small segments at both ends connecting them. If you keep adding roads they are going to be split in more segments (called tiles), but that is not relevant here.

With the existing code, I realized that when a Mobile Station arrived to the end of one segment, instead of continuing to the opposite direction, it was teleported to the other extreme of the road (from point 2 to 4 in Figure 5).

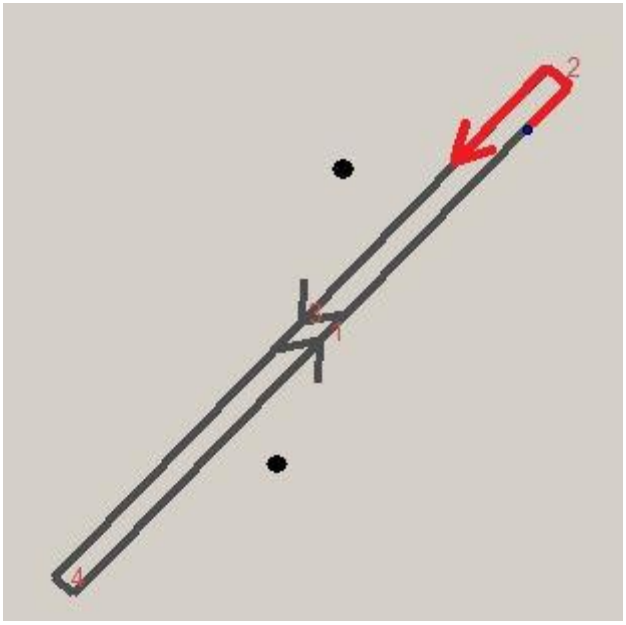


Figure 5. Example of path followed by a roaming Mobile Station in the simulation program.

The number of time slots in the testing simulations was not really high, because it was not worthy to spend a lot of time simulating when we just needed to see if the program was working properly. So I did not notice this bug until I increased the speed of the Mobile Stations for letting them cover a longer path.

The old code written for moving the MSs was kind of complicated. In it, the decision of the final position was determined by trigonometry from the road angle, which was computed using the arctangent function. This method was perfectly valid and pretty elegant but there was a problem: the ambiguity of the arctangent function.

The arctangent function is not injective in the interval $[0, 2\pi)$, and we need to state the quadrant where the angle is using another complementary criteria. For solving this, in the old code a bunch of nested *if* conditions were used after the distance was calculated, resulting in a very complicated algorithm that did not work well in every occasion.

I decided to build the whole method again, using the sign of the numerator inside the arctangent function for determining the real angle before any displacement is computed. I made a huge reduction in the length and complexity of the code, going from more than 100 code lines to less than 50. And the method was now working perfectly fine.

The code is presented below, with all the comments included.

```
void Mobile_Station::MSposition_Roads(void)
{
   // Road Model:
   // The mobile MS has to travel along the Roads i.e the Roads
   // In this model speed of the user travelling on certain Road(Road segment)
   // is determined by the allowed speed on that Road. We consider the stationary
   // and pedestrian users to have zero speed

   // Assume the mobile MS are intially configured to be on any of the Roads

   // 1. Refer the speed of the argument CurrentRoad calculate the next point
   //    on that Road
   // 2. If that point is on the CurrentRoad, return pointer to the same Road
   // 3. If that point is beyond the end point of CurrentRoad, according to probability
  //of transition to next Roads, choose the next Road and return the pointer
  //to that Road

   /* generate uniform random number in the interval [1,100] */
   int temp = rand() % 100 + 1;
   double p = 0;

   //Extremes of the road segment
   double x1 = CurrentRoad->StrtEndPts[0];
   double y1 = CurrentRoad->StrtEndPts[1];
   double x2 = CurrentRoad->StrtEndPts[2];
   double y2 = CurrentRoad->StrtEndPts[3];

   //Distance to travel
   double d = (CurrentRoad->speed)*Sim->simCfg.slot_time;

   double theta = atan((y1-y2)/(x1-x2)); //Angle of the road
   //It is going to be always between -pi/2 and pi/2

   //We solve the ambiguity of atan checking the denominator sign
   if(x1-x2 > 0)
          theta = theta + 3.14159;

   //We add the distance to each coordinate
   X = X + d*cos(theta);
   Y = Y + d*sin(theta);

   if(X > max(x1,x2) || X < min(x1,x2) || Y > max(y1,y2) || Y < min(y1,y2))
   {
          //If the MS is out of the road after the movement,
          //choose next Road depending on the given probabilities

          // If there is no next Road i.e deadend of a Road, it will stop moving
          if (CurrentRoad->nextRoads.size()<=0)
```

```
        {
                X = x2;
                Y = y2;
                isMobile = false;
                return;
        }

        //We need to check all the roads that are connected to this end
        for (int i=0; i < (int) CurrentRoad->nextRoadProbs.size(); i++)
        {
                // Assuming that the nextRoads are sorted ascendingly
                // by their Probabilities

                //Find the cumulative probability
                p = p + CurrentRoad->nextRoadProbs[i];

                //If the cumulative probability is greater than the
                //random number found before,
                //we have found the next road to follow
                if (temp <= 100*p)
                {
                        CurrentRoad = CurrentRoad->nextRoads[i];
                        break;
                }
        }

        //Update the coordinates to the end of the road
        X = x2;
        Y = y2;
        return ;
    }
}
```

We need to have in mind that due to the short duration of a time slot, we do not need to worry about the distance that the Mobile Station would still need to move in the next road and the method stop the Mobile Station at the end of the current road until the following time slot. Doing this, we avoid making the code more complex. Anyways, it is a realistic assumption to consider that the Mobile Stations spend some time turning back.

6. Predictive Handoff Algorithm

In this section, I am going to explain how the Predictive Handoff algorithm developed during the elaboration of this project works. This will help the reader to understand the process I follow in the creation of the algorithm.

The general algorithm for taking the information of all the possible cases that can happen in the future is a double recursive method composed by three main functions. I will explain these functions in a bottom-top approach, from the lowest and simplest abstraction level to the more general one. But first we need to explain the contents of the Position class, which is a key part in this algorithm.

6.1. Position class

Position could be called a "container class", because the objects of it act as containers of information about a certain location, without any function.

This class contains the following information in its variables:

- **X and Y (doubles)**: They contain the coordinates of the position in the XY plane.
- **road (*Road)**: Pointer to the road segment to which that position belongs.
- **prob (double)**: Probability of being in that position.
- **BSsequences (vector<vector<int>>)**: Matrix whose elements are the number of times that combination of next Base Station ID (rows) and number of hand-off (columns) occurs. The purpose of this will make sense when we explain the predictive algorithm.

6.2. Movement

The first thing we need is a function that, given the position of a Mobile Station and a certain time, is able to give us a list with all the possible different positions of the Mobile Station after that time.

This function will be inspired in the movement function used for controlling the motion of the Mobile Stations in the simulator, but with substantial differences:

a) Now it does not make sense to stop the Mobile Station at the end of a road if that road ends. We need to calculate which distance it should travel in the new road and do it. In this case we are working with longer times (that of one step in the Predictive Handoff algorithm), so it would not be rare if we need to repeat this process several times because the Mobile Station can travel through several roads in the given time. That is why we need the function to be recursive, for being repeated for subsequent roads until the time is over. The time left will be the distance already travelled over the speed of the road where it was moving.

b) Instead of taking the most probable path as we do calculating the motion, we need to cover now all the possibilities. The output of the function will be a list of all the possible final destinations of the Mobile Station.

The code of this function is presented below, with a detailed explanation in the comments.

```
vector<Position> Predictive_Handoff::movement(Position* currentPos, double ts)
{
	vector<Position> positions;
	vector<Position> newPositions;

	iteration++;

	//Extremes of the current road
	double x1 = currentPos->road->StrtEndPts[0];
	double y1 = currentPos->road->StrtEndPts[1];
	double x2 = currentPos->road->StrtEndPts[2];
```

```
double y2 = currentPos->road->StrtEndPts[3];

double theta = atan((y1-y2)/(x1-x2)); //Angle of the road
//It is going to be always between -pi/2 and pi/2

//We need eventually to include a correction because of the atan ambiguity
if(x1-x2 > 0)
	theta = theta + 3.14159;

//Distance = speed * time
double d = (currentPos->road->speed)*ts;

//Trigonometry for finding the new position of the MS
double X = currentPos->X + d*cos(theta);
double Y = currentPos->Y + d*sin(theta);

Position* pos;

//We need to check if this position is outside the road or not
if(X > max(x1,x2) || X < min(x1,x2) || Y > max(y1,y2) || Y < min(y1,y2))
{
	// ------------Out of the Road-----------//
	// -------- Choose next Road depending on the given probabilities ------- //

	if (currentPos->road->nextRoads.size()<=0) // If there is no next Road i.e deadend
	                                           //of a Road, it will stop moving
	{
		X = x2;  //We stop at the end
		Y = y2;
		pos = new Position(X,Y,currentPos->road,currentPos->prob);
		positions.push_back((*pos));
		delete pos;
		return positions;
	}

	double p;
	double timeleft;
	Road* newRoad;

	//Going through all the possible next roads
	for (int i=0; i < (int) currentPos->road->nextRoads.size(); i++)
	{
		//The probability of this new position will be affected by the
		//prob. of the current position and the prob. of change
		//We suppose independent events
		p = currentPos->prob * currentPos->road->nextRoadProbs[i];

		//According to the distance from the current point to the end of the road,
		//we need to calculate the time left for the rest of the movement
		timeleft = sqrt(pow(X-x2,2)+pow(Y-y2,2))/((currentPos->road->speed));

		//With the new position (end of the current road) and new road we call
		//recursively the movement method again

		newRoad = currentPos->road->nextRoads[i];

		pos = new Position(newRoad->StrtEndPts[0],newRoad->StrtEndPts[1],newRoad,p);

		newPositions = movement(pos, timeleft);
		iteration--;

		//Finally, we take all the positions together
```

```
                positions.insert(positions.end(),newPositions.begin(),newPositions.end());

                delete pos;
            }
            return positions;

        }else{
            if(Sim->sim_stats->MS_stats[MS->Ind].VerboseMode && iteration>2){
                MS->Iterations.push_back(iteration);
            }
            //If we are in the same road, we just change the coordinates of position
            pos = new Position(X,Y,currentPos->road,currentPos->prob);
            positions.push_back((*pos));
            delete pos;
            return positions;
        }
}
```

6.3. Gathering the Data for the Prediction

For making the prediction, we need to gather first all the information about the possible hand-offs in the future. For that, I created a recursive method that, using the movement function of Section 6.2, goes into the future and return the number of hand-off "jumps" needed for each path.

The idea is obtaining the prediction in each step of the recursion from all the possible Base Station paths we are finding. For example, in the hand-off tree shown in Figure 6, we would have the following list of Base Stations paths:

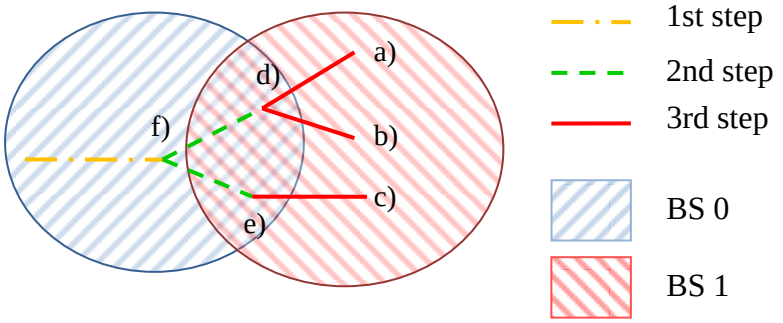
$$[(0 \quad 0 \quad 1), (0 \quad 1 \quad 1), (0 \quad 0 \quad 1), (0 \quad 1 \quad 1), (0 \quad 0 \quad 1), (0 \quad 1 \quad 1)]$$

, which are all the possible Base Station paths we can have, having in mind all the three different end locations.

It is evident that we have redundant information in that vector. We do not need to know to which Base Station we could switch three steps ahead for choosing a Base Station now. We just need to know the number of jumps we will have for each path if we choose one certain Base Station now.

Actually, we could not implement the algorithm including in each step the whole list of Base Stations: the computer memory was easily overflowed with a medium-sized scenario. We had to simplify and erased all the non-essential information if we wanted it to work properly.

The implemented algorithm is based in matrices. The only thing the method returns is all the final positions in the prediction, which are objects of the Position class. But



a) $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$ b) $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$ c) $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

d) $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$ $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$ e) $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

f) $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ $\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$

Figure 6. Example for better understanding the algorithm of gathering the data for the prediction (3 steps). Below the image we have the BSsequence matrix of all the object Position gathered in each position.

inside those objects, the information about the number of hand-offs is contained in a matrix, *BSsequences*. Its rows correspond to each Base Station, and the columns are the number of hand-offs (0, 1, 2, etc.). The number of column is equal to the depth of the recursive algorithm (how many steps ahead do we want in the prediction) plus one, because we are not going to have a higher number of hand-offs. Each element represents how many times we have that number of hand-off if the Mobile Station goes to that final position and we now change to the Base Station indicated in the row.

In the Figure 6, we have a simple example of the generated matrices where you can follow the construction of the "recursive tree" from the leaves to the root.

The matrix is created in one of the final positions. It will have all the elements as zero except those elements of the first column corresponding to Base Stations giving coverage to that final position.

The procedure for changing the number of the matrix in other steps is simple. We first start with an all-zeros matrix (*tempvector*) and we update the elements with the information read from the matrix of each specific final position, gathered from the last recursive call. Actually, we will add numbers from this matrix (let's call it *sonMatrix*) to tempvector.

The algorithm goes through each Base Station we have giving coverage to the actual position. We are going to call BSa to that Base Station's ID. We follow then two rules:

a) We add the row number BSa of *sonMatrix* to the row number BSa of *tempvector*. If we have BSa in both positions, there are no new hand-offs.
b) We add all the other rows of *sonMatrix* to the row number BSa of *tempvector* but displaced one position to the right. That means the second element of *sonMatrix*'s row is added to the first element of *tempvector*'s row and so on. That

means we are considering a hand-off considering we are changing from BSa in next step.

If we are in the root of the tree, we also need to compute the probability of appearance of each Base Station in the following step and store this data in the *nextBS* vector. This information will be used later in the prediction.

I think the best way for understanding the algorithm is following the code, which is shown below.

```
vector<Position> Predictive_Handoff::prediction(int depth, Position* pos)
{
	vector<Position> paths;
	vector<Position> newPaths;

	//We repeat the recursive method until 'depth' reaches 0
	if(depth > 0){

		iteration = 0;

		PROCESS_MEMORY_COUNTERS_EX pmc;
		int result;
		int ma, mb;
		result = GetProcessMemoryInfo(GetCurrentProcess(),
			(PROCESS_MEMORY_COUNTERS*)&pmc,sizeof( pmc ));
		ma = pmc.PrivateUsage;

		//Prediction of the different positions of the MS after the sampling period
		vector<Position> positions = movement(pos,samplingPeriod);

		PROCESS_MEMORY_COUNTERS_EX pmc2;
		result = GetProcessMemoryInfo(GetCurrentProcess(),
			(PROCESS_MEMORY_COUNTERS*)&pmc2,sizeof( pmc2 ));
		//printf( "M: %i\n", (pmc2.PrivateUsage-ma));

		//For each position we need to do another prediction
		for(int i=0; i<(int)positions.size(); i++)
		{
			//We call the method recursively changing the depth value and
			//from the new position
			newPaths = prediction(depth-1, &positions[i]);

			//Data processing of each final prediction
			for(int j=0; j<(int)newPaths.size(); j++)
			{
				vector<vector<int> > tempvector (Sim->numBS,
					vector<int>(maxDepth+1,0)); //Temporal vector

				//We need to add the information of each possible BS in this tile to
				//the compiled information from each prediction
				for(int k=0; k<(int)pos->road->SelfTilePtr->QoS_BS.size(); k++)
				{
```

```
                        /*The matrix BSsequences has a summary of the number of
                        jumps of each possible path
                        The rows represent the BS index and the columns are the
                        number of jumps. The maximum number of jumps cannot be
                        greater than the current depth of our recursive method,
                        so we use the depth as a reference for going through the
                        columns of the matrix.
                        The number in each position is the number of handoff paths
                        we have with that number of jumps (column) and starting with
                        that BS (row), so we can later determine which is the best
                        handoff path for each final position of the MS.
                        */

                        //Going through the columns
                        for(int l=0; l<depth; l++)
                        {
                                //Going through the rows
                                for(int h=0; h<Sim->numBS; h++)
                                {
                                        //If that BS index is in the current tile, we
                                        //dont have any new jump and we clone the number
                                        //of paths with that amount of handoff jumps
                                        if(h==pos->road->SelfTilePtr->QoS_BS[k])
                                        {
                                                tempvector[h][l] +=
                                                        newPaths[j].BSsequences[h][l];
                                        }
                                        else
                                        {
                                        //In any other case, we have a new jump and we
                                        //add the information to the next column and
                                        //to the row of the BS that we are exploring
                                        tempvector[
                                                pos->road->SelfTilePtr->QoS_BS[k]][l+1] +=
                                                newPaths[j].BSsequences[h][l];
                                        }
                                }
                        }
                }

                //Storage of the matrix
                newPaths[j].BSsequences = tempvector;
        }

        //We include the new paths in the path array
        paths.insert(paths.end(),newPaths.begin(),newPaths.end());

        //If we are in the root of the tree (beginning), we need to take
        //information of the BS appearance in the next prediction step. We will
        //use this information later in the BS update.
        if(depth == maxDepth)
        {
                for(int l=0;
                        l<(int)positions[i].road->SelfTilePtr->QoS_BS.size(); l++)
                {
                        nextBS[positions[i].road->SelfTilePtr->QoS_BS[l]] +=
                                        positions[i].prob;
                }

        }
}
```

```
			return paths; //Return of all the possible final positions
		}
		else
		{
		//If we are at the end of the predicition tree, we need to create the Position object
			Position* endPosition = new Position(pos->X,pos->Y,pos->road,pos->prob);
			vector<vector<int> > tempvector (Sim->numBS, vector<int>(maxDepth+1,0));

			//We have just routes with 0 jumps so far
			for(int k=0; k<(int)pos->road->SelfTilePtr->QoS_BS.size(); k++)
			{
				tempvector[pos->road->SelfTilePtr->QoS_BS[k]][0]++;
			}

			endPosition->BSsequences = tempvector;
			paths.push_back(*endPosition);
			delete endPosition;
			return paths;
		}
}
```

Even if the information gathered by the method is very reduced, we still have redundancies on it. For example, at some points of the recursive tree we can have some specific Base Station sequences that we know are going to beat up all the other possibilities with much more hand-offs, so we could keep just those for the prediction.

However, with this method the allocation of memory is static, because the matrices have always the same size, which is very good for the program speed, even if most of the elements have useless information. Otherwise, we would end with an even more complicated code, difficult to debug.

Anyways, this recursive code is, surprisingly, very efficient. In fact, we almost cannot feel the difference in time between the Predictive Handoff and the Instantaneous Pilot schemes in a normal scenario.

6.4. The Prediction Itself

From all the finals positions contained in the vector returned by the *prediction* function, and the results stored in their matrices, we have to determine which is the best Base Station for being connected to.

Our main purpose is minimizing the number of hand-offs, and we will based our prediction in the most probable situations. Of course, the program's predictions won't be always correct, because even if the probability of taking certain route is very small and we do not minimize the number of hand-offs for that route, it is still possible that the Mobile Station travels in that direction.

We will have a number of possible routes, each one associated with one final position and with a certain number of Base Station paths, or Base Stations sequences the Mobile Station can take until arriving to the final position. The information of all this paths will be summarized in the matrix we studied in Section 6.3.

For doing the prediction, the program needs first to assign different probabilities to all the Base Stations in the Mobile Station's scope right now. The probabilities of the Base Stations in each path will be weighted by the probability of being in that final position, and then all the probabilities for all the paths will be summed up. The program will only analyze the rows corresponding to Base Station that are reachable in this point, for reducing the computational effort, and it will follow certain conditions for gathering the probabilities:

a) The program will analyze just the first column where some of the elements are not zero. Each column represents a number of hand-offs from 0 to *maxDepth*+1, where *maxDepth* is the maximum number of steps of the algorithm, so the information about the best possibilities is contained in the first non-null column.

b) Once we find that column, the program applies the following conditions to its elements:

   a. If the element of the row associated with the Base Station the Mobile Station is currently connected to has non-zero value, we will assign to that Base Station 100% of chances. Have in mind that the present hand-off, from the Base Station associated to the Mobile Station in this moment, was not included in the information contained in the matrix. We should then consider the elements of the row corresponding to this Base Station as shifted to the left from the original matrix (one hand-off less is needed), whose columns are considered now to start in 1 jump instead that in 0.

   b. If the element of the row associated to the current Base Station is zero, then we have to calculate the probability of each Base Station between all those which are reachable now and have the minimum number of handoffs. Remember that the current Base Station needs a hand-off less than the others, so we take the number from the next column.

      The calculation of the probabilities is summarized below, with *k* for the indexes of the reachable Base Stations and being *c* the selected column:

$$P(k) = \begin{cases} \dfrac{BSsequences[k][c+1]}{BSsequences[k][c+1] + \sum_{n \neq k} BSsequences[n][c]} & if\ k = currentBS \\ \dfrac{BSsequences[k][c]}{BSsequences[k][c+1] + \sum_{n \neq k} BSsequences[n][c]} & otherwise \end{cases}$$

For example, imagine this situation:

i. There are 3 Base Stations (0,1 and 2), and only 1 and 2 are reachable right now
ii. The current Base Station is the one with index 2.
iii. The matrix of the analyzed final position is: $\begin{pmatrix} 0 & 2 & 6 \\ 0 & 1 & 4 \\ 0 & 0 & 3 \end{pmatrix}$
iv. The probability of being in that final position is 0.5.

The selected column is the second one. The element corresponding to the current Base Station is zero, so the probabilities are computed. The probability vector would be: $\begin{bmatrix} \frac{1}{4} \cdot 0.5 & \frac{3}{4} \cdot 0.5 \end{bmatrix} = \begin{bmatrix} \frac{1}{8} & \frac{3}{8} \end{bmatrix}$

The Base Station 0 is not considered, because it is not reachable right now.

Once we have all the probabilities, it is time for doing the prediction. The program is not going to simply pick up the most probable Base Station. Remember that the previous method, *prediction*, gathered also the probability of appearance of each Base Station in the first step and stored it in a global variable. Our condition will require the Base Station is the most probable and its probability of appearance in the next step is above a certain threshold.

In other words, the program is going to choose the most probable Base Station in terms of minimizing the number of hand-offs among all the Base Stations whose probability of appearance in the next step is above that certain threshold. If it is done in

this way, we make sure that the program does not make the Mobile Station switch to a Base Station that does not appear very often in the possible paths.

For instance, imagine we have just two final positions, P1 and P2. As best Base Station for minimizing the number of hand-offs we have A in P1 and B in P2, both reachable in the current position. B is also reachable in the first prediction step of P1 path, but the Mobile Station cannot connect to A in any point of P2 path.

If the Mobile Station has a probability of 0.55 for going to P1 and 0.45 for P2, the basic algorithm will choose A as next Base Station. But P2 probability is really high, and the option of the Mobile Station taking P2 path is likely enough for discarding A and taking B, which appears also in P1. The number of hand-offs for P1 is not minimized but the overall result is better.

We want a really high threshold of appearance, 0.9. For being sure the program is deciding about the Base Station in a predictive way even if there is no Base Station matching this condition, I came up with the idea of repeating the decision process lowering the threshold in steps of 0.1 until a Base Station is found or the threshold reaches 0. Any of the Base Stations found using this method is better than the Instantaneous Pilot scheme if we look for a minimization in number of hand-offs.

If even after doing this any Base Station is meeting the criteria, the predictive algorithm is not useful in this case and we have to apply the Instantaneous Pilot scheme, connecting the Mobile Station to the Base Station with the most powerful signal among the reachable ones.

There is one exception to this: if the Base Station connected right now to the Mobile Station is still reachable, the Mobile Station stays attached to it. At least, this will avoid the Mobile Station a hand-off in this moment.

This was the final method of the whole algorithm of prediction. We will see in Section 8 that it is working pretty well. Its theoretical complexity is not reflected in the time of execution, which is very similar to the case with no prediction as soon as we do not have a complicated knot of roads.

Again, all this explanation does not make sense if the reader does not have the opportunity to have a look to the code, which is shown below.

The algorithm of decision described is inserted in the method *update*, which is the one called in every time slot for updating the movement of the Mobile Station and the election of the Base Station associated to it.

```
void Predictive_Handoff::update()
{
	vector<double> temp (Sim->numBS,0);
	nextBS = temp;
	MS->Iterations.clear();
	MS->HOflag = 1;

	//Current position of the MS, with (relative) probability = 1.0
	Position *pos = new Position(MS->X,MS->Y,MS->CurrentRoad,1.0);

	//Calculation of all the possible final positions
	vector<Position> paths = prediction(maxDepth,pos);

	//Vector for storing the probability of changing to a BS
	vector<double> BSprobs (Sim->numBS,0.0);

	//Going through all the final positions
	for(int i=0; i<(int)paths.size(); i++)
	{
		//Going through the columns of the "handoff matrix" in that route
		//We will stop in the first column we find with non-zero numbers
		//(minimum number of jumps)
		for(int k=0; k<maxDepth+1; k++)
		{
			int counter = 0; //Counter for summing up all the element of a column
			//Going through the rows, but just through those which represent a BS
			//presented in this tile
			//We do this for saving time
			for(int j=0; j<(int)MS->CurrentRoad->SelfTilePtr->QoS_BS.size(); j++)
			{

				if(MS->CurrentRoad->SelfTilePtr->QoS_BS[j]==MS->Connect_BS_ind)
				{
					if(paths[i].BSsequences[MS->Connect_BS_ind][k]!=0)
					{
					//If the BS checked is the same than the current BS
					//and the number of paths is != 0, we are going to stay
```

```
                        //in that BS for minimizing the energy
                                BSprobs[MS->CurrentRoad->SelfTilePtr->QoS_BS[j]] =
                                        paths[i].prob;
                                counter = -1;
                                break;
                        }else if(k<maxDepth){
                        //If the number of paths is 0, we would need to take the
                        //number from the next column, because we need a jump less for
                        //that BS due to we already are attached to it
                                counter +=
                                        paths[i].BSsequences[
                                        MS->CurrentRoad->SelfTilePtr->QoS_BS[j]][k+1];
                        }
                }
                else
                {
                        //If we are not attached to that BS, we just
                        //increment the counter
                        counter +=
                                paths[i].BSsequences[
                                MS->CurrentRoad->SelfTilePtr->QoS_BS[j]][k];
                }
        }

        //If counter == 0, we just go to the next column.
        //We are going to choose a path from the first column where counter == 0
        //All the next columns represent a higher number of jumps
        if(counter==-1)
                break;
        else if(counter!=0)
        {
                //We go again through all the BS for computing the probabilities
                //using the total value in counter
                for(int j=0; j<(int)MS->CurrentRoad->SelfTilePtr->QoS_BS.size(); j++)
                {
                //Remember that if we are in the current BS, we need to look to the
                //next column for taking into account that we dont need the first
                //jump
                        if(MS->CurrentRoad->SelfTilePtr->QoS_BS[j]==MS->Connect_BS_ind
                                && k<maxDepth)
                        {
                                BSprobs[MS->CurrentRoad->SelfTilePtr->QoS_BS[j]] +=
                                        ( ((double)paths[i].BSsequences[
                                        MS->CurrentRoad->SelfTilePtr->QoS_BS[j]][k+1]) /
                                        (double)counter )*paths[i].prob;
                        }
                        else
                        {
                                BSprobs[MS->CurrentRoad->SelfTilePtr->QoS_BS[j]] +=
                                        ( ((double)paths[i].BSsequences[
                                        MS->CurrentRoad->SelfTilePtr->QoS_BS[j]][k]) /
                                        (double)counter )*paths[i].prob;
                        }
                }
                break; //We already have what we want, so we leave
        }
    }
}

delete pos;

int mostlikely = -1;                    //Most likely BS index
```

```
	double mostlikelyProb = 0.0;			//Most likely BS probability
	double BSPresence = BSMinPresence;	//Minimum presence of the BS in the next step
					//for being useful

	for(int i=0; i<(int)MS->CurrentRoad->SelfTilePtr->QoS_BS.size(); i++)
	{
	//For each BS in this tile, if the probability of appearance of that BS in the future
	//is higher than the previous one's probability and the presence in next step is higher
	//than our limit, we choose as far that BS as the most likely
		if((BSprobs[MS->CurrentRoad->SelfTilePtr->QoS_BS[i]] > mostlikelyProb ||
			(BSprobs[MS->CurrentRoad->SelfTilePtr->QoS_BS[i]] = mostlikelyProb &&
			MS->CurrentRoad->SelfTilePtr->QoS_BS[i]==MS->Connect_BS_ind))
			&& nextBS[MS->CurrentRoad->SelfTilePtr->QoS_BS[i]]>BSPresence)
		{
			mostlikely = MS->CurrentRoad->SelfTilePtr->QoS_BS[i];
			mostlikelyProb = BSprobs[MS->CurrentRoad->SelfTilePtr->QoS_BS[i]];
		}
	//If we already checked all the base station and we didnt find any result, we would
	//need to low our presence limit (by 0.1)
		if(i==(int)MS->CurrentRoad->SelfTilePtr->QoS_BS.size()-1 && mostlikely == -1
			&& BSPresence > 0)
		{
			i=0;
			BSPresence -= 0.1;
			if(BSPresence < 0)
				BSPresence = 0;
		}
	}

	if(mostlikely==-1)
	{
	//If we dont have any most likely BS and the current BS is reachable in this
	//tile, the best option is staying attached to it
		for(int i=0; i<(int)MS->CurrentRoad->SelfTilePtr->QoS_BS.size(); i++)
		{
			if(MS->CurrentRoad->SelfTilePtr->QoS_BS[i]==MS->Connect_BS_ind)
			{
				return;
			}
		}

		//If we need to change, we use the Instantaneous Pilot scheme
		MS->Connect_BS_ind = Sim->network->FindMaxPilotInst(MS->Ind);
	}
	else
	{
		//Connection to the most likely BS
		MS->Connect_BS_ind = mostlikely;
	}

	if(Sim->sim_stats->MS_stats[MS->Ind].VerboseMode){
		MS->BSprobs = BSprobs;
		MS->BSPresence = nextBS;
	}
}
```

6.5. Mathematical formulation of the algorithm

As a summary, we present below the pseudo-code describing the predictive hand-off algorithm. It includes all the process in the prediction: the simulation of possible future movements, the data recollection and the prediction itself from the probabilities gathered.

For understanding the algorithmic pseudo-code, we need to have in mind the meaning of each one of the variables:

- T_H is the time elapsed between two consecutive hand-offs (i.e. hand-off interval).
- n_{PH} and t_{PH} represent the number of steps and their temporal size in the prediction, respectively.
- $L(t)$ is the set of all possible final positions of the mobile station after time t.
- $K(l, b, \theta)$ is the number of all possible sequences of base stations for arriving to a final position l, starting with a base station b and with a certain number of hand-offs θ.
- $R_{p_l}(b, \theta)$ is the set of all the sequences of base stations given a final position l, the first base station in the sequence b and a number of hand-offs θ:

$$R_{p_l}(b, \theta) = \{r | r_n \in \Gamma_{p_n}, \forall n: 1 \leq n \leq N_p \wedge r_1 = b \wedge \Theta(r) = \theta\}$$

- $b_{best}(l)$ represents the best base station for switching to if the mobile station is going to the final position l.
- $b_{current}$ is the base station to what the mobile station is attached before the prediction.

- $\Psi(b)$ is the probability of finding base station b as the best one among all possible paths.
- L_b is the set of all final positions after time $n_{PH} \cdot t_{PH}$ (i.e. prediction interval) whose optimal base station for switching to in the current time slot is b:

$$L_b = \{l \in L(n_{PH} \cdot t_{PH}) | b_{best}(l) = b\}$$

- $P(l)$ is the probability of having the final location l as a destiny.
- B_{best} is the set of best base stations for switching to according to the results in $\Psi(b)$:

$$B_{best} = \{b \in B | \Psi(b) \geq \Psi(\alpha) \forall \alpha \in B\}$$

- $\Omega(b)$ represents the probability of base station b being present in the next step of the prediction, whose minimum threshold for a good prediction is Th.

Once we have defined every variable used, here it is the pseudo-code of the predictive algorithm.

Run this algorithm at time intervals of T_H

for all $l \in L(n_{PH} \cdot t_{PH})$ **do**

  for all $b \in B, 0 \leq \theta \leq n_{PH}$ **do**

    $K(l, b, \theta) = size\ of\ R_{p_l}(b, \theta)$

  end for

  $For\ \ K(l, B, \Theta)\ find\ b_{best}(l)$

end for

$$\Psi(b) = \sum_{l \in L_b} P(l)$$

$Select\ elements\ in\ B_{best}$

for all $b \in B_{best}$ **do**

if $\Omega(b) > Th$ **then**

if $b = b_{current}$ **then**

$No\ hand\text{-}off$

END

else

$Hand\text{-}off\ to\ first\ element\ in\ B_{best}$

END

end if

end if

if $\Omega(b) < Th\ \ \forall\ b \in B_{best}$ **then**

$Low\ Th\ until\ \exists b\ \in\ B_{best} : \Omega(b) > Th$

end if

end for

if $No\ BS\ has\ been\ selected$ **then**

if $b_{current}\ still\ meets\ the\ QoS$ **then**

$No\ hand\text{-}off$

END

else

$Choose\ BS\ using\ Instantaneous\ Pilot\ Scheme$

end if

end if

7. Bugs that appeared in the process and how we solved them

7.1.Bug 1: Double Calculation of Probability

The first thing I did was solving two bugs of the code. Even if the program seemed to work fine, a closer look made us realize that something was wrong during the execution.

We first notice that in the graphs I presented as a result in the Winter Report, which is shown here in Figure 7, the first values of number of handoffs were higher when we used the Predictive Algorithm than when we used Instantaneous Pilot.

At that moment we thought that was very strange. If you design a method having in mind the objective of minimizing the number of handoffs, it has necessarily to return a number of handoffs lower than in any other of the methods.

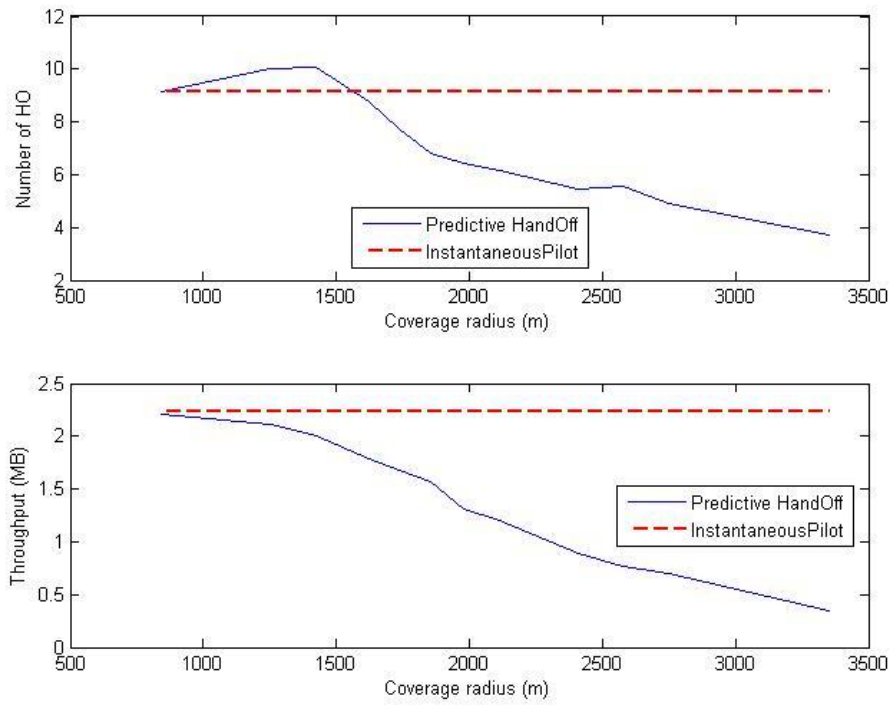


Figure 7. Results from old code. Average number of hand-offs and throughput as a function of coverage radius. Scn_Freeway scenario, predictive algorithm depth = 2

Anyways, we didn't have time for correcting it in the Winter Quarter and, because we thought the code was correct, I considered that problem scenario-dependent.

Once the Spring Quarter started and we had more time for working in this project, we analyzed the key points of the code. I repeated some simulations storing the vectors *BSprobs* and *nextBS*, already presented in the last report.

Analyzing those vectors, we discovered that sometimes the probabilities of choosing one Base Station or other summed up to 2, instead of 1 as it should be. Why 2? The only reason I could think of was that we were considering the probability for each Base Station twice. But the surprising thing was that the probabilities were different, not just twice the value it should be.

Going back to the code, it was not difficult to find where the problem was: when the code was studying all the Base Stations one by one in each possible path for obtaining the probabilities of choosing them in the next handoff.

In the cases when the code noticed that the analyzed Base Station was the current Base Station to which the Mobile Station was attached *and* that Base Station was not the first one in the list of Base Stations, the search is supposed to finish, because if a Mobile Station has the current Base Station in the next prediction period the optimal solution in terms of number of handoffs is staying attached to it.

But the problem here was that we trusted in a counter for skipping the next steps in the search, and that counter was incremented each time a Base Station different to the current one was analyzed. If the counter was 0, the code was supposed to find the current Base Station in the next prediction lapse. However, if that Base Station was not the first one in the list of Base Station, the next step in the search was not skipped and the calculation of probability was repeated again, so the sum of probabilities was 2.

This happened in just a few cases, but they were enough for affecting the results. Solving the problem was easy: we just needed a third value for the counter, -1 (in addition to 0 and non-0), which meant that the current Base Station was found in the next prediction lapse.

Anyways, we found another bug in the program which needed to be solved for obtaining perfect results.

7.2. Bug 2: Bad election when all the Base Stations have the same probability

As all the probabilities of choosing the Base Stations were analyzed for every path, the information was stored in the vector *BSprobs*, which finally has the global probabilities of choosing those Base Stations for minimizing the number of handoffs. From that vector, the Base Station with higher likelihood is chosen.

But we thought about what happened if there were some Base Stations with the same probability and this was higher than the others, a really common situation. In that case, the first Base Station in the list with that high probability was strictly chosen.

This is completely logical: if all the possible Base Stations are going to give us the same results according to the statistics it does not matter which method we use for selecting one of those.

At that moment we realized that not all of the “top” Base Stations had an equal status. What happen if one of those is the Base Station to which the Mobile Station is currently attached? In that case, that one clearly minimizes the number of handoffs, so it should be chosen for sure. This was not checked in the code and the selection was

purely based on the position of the Base Stations in the list, so the results were not optimal in terms of number of handoffs.

This problem was easy to solve. I just needed to add one additional condition in the search of the best Base Station for checking if the currently attached one is among the most likely ones.

7.3. Corrected Code

After solving those bugs, the final version of the predictive code was ready. We just needed to try it. The reader could find the complete code in the last section, when we described each of the method involved.

We have to say that some lines of the code were actually included for testing purposes in order to solve a big memory problem. We will talk about it in the next subsection.

The next step was checking if the method was now working perfectly. I repeated the simulations for the same scenario I was using for testing purposes (*Scn_Freeway*) and the results were totally successful. The average number of handoffs and the throughput as a function of the coverage radius are shown in Figure 8 for both codes, old and new.

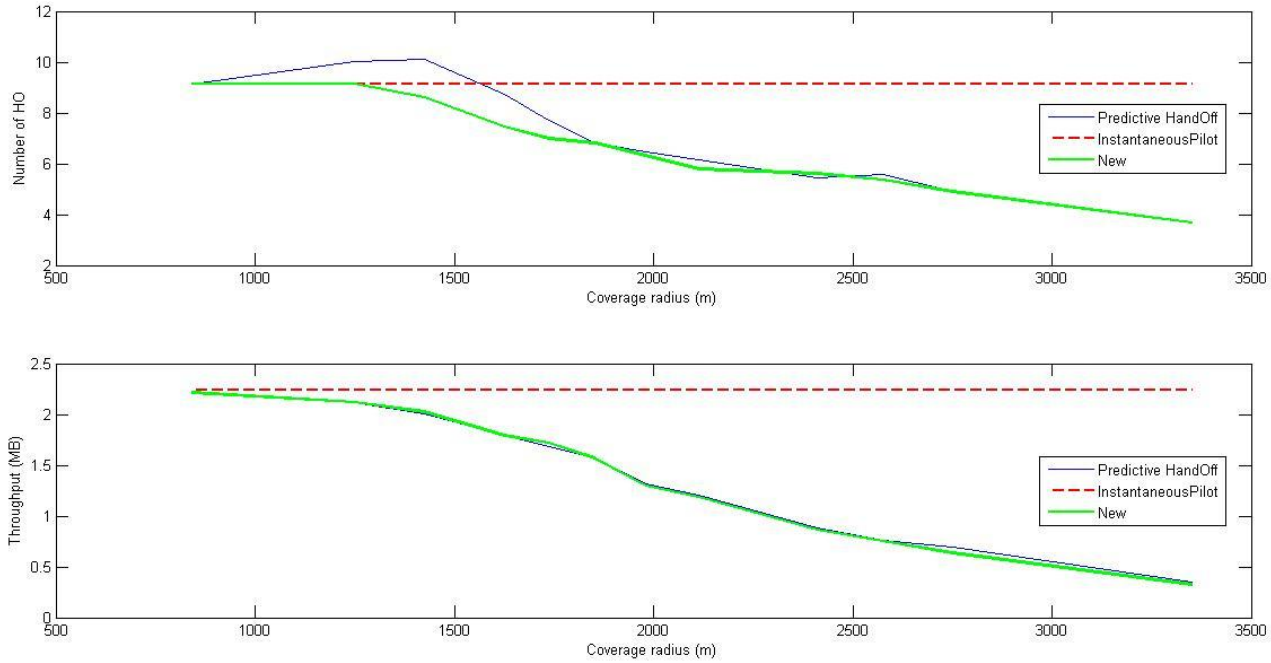


Figure 8. Results from old code vs. those from new code. Average number of hand-offs and throughput as a function of coverage radius. Scn_Freeway scenario, predictive algorithm depth = 2.

Now, as we supposed, the average number of handoffs is always lower for the predictive case than for the instantaneous pilot. And the throughput is almost the same, while the number of handoffs dropped a considerable amount for small coverage radius.

Apart from this graph, I created one (Figure 9) where the average number of handoffs and the throughput are represented as a function of the density of Base Station.

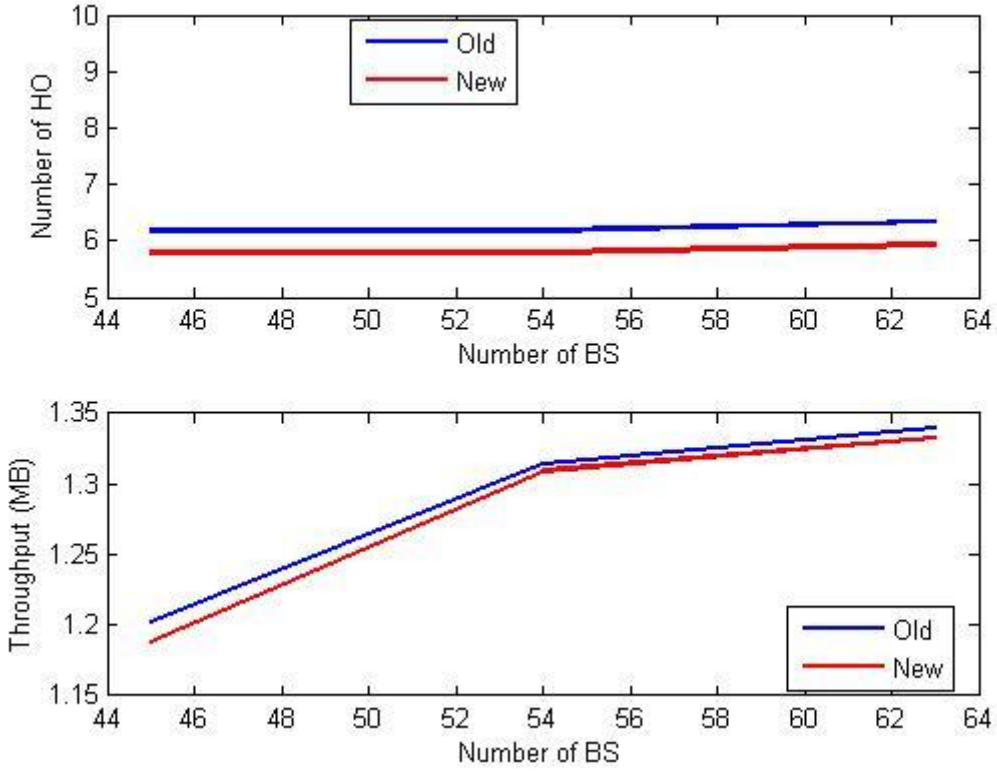


Figure 9. Results from old code vs. those from new code. Average number of hand-offs and throughput as a function of the number of BS. Scn_Freeway scenario, predictive algorithm depth = 2, coverage radius ≈ 2 km

In this graph, the algorithm's performance improvement is also shown. The average number of handoffs is lower and the fall in the throughput is not high. Our objective is minimizing the number of handoffs and keeping the throughput above certain threshold. In this case, the throughput is much higher than the minimum needed for a cellular phone call.

7.4. New Problem: Memory Overflow

Once the code was working perfectly, we needed to start the simulations and obtain some results again. As we saw in Figure 8 and Figure 9, we were able to simulate with no problems for different coverage radius and different Base Station density.

But apart from changing those parameters, it would be interesting to check the behavior of the program for different depths of the predictive algorithm. When we tried to simulate for depth higher than 2 steps, all the simulations stopped running at some point.

Something was happening during the execution time which makes the simulations fail. Finding the real problem took us several weeks, and I will try to summarize the process in the following subsections.

7.4.1. Checking the Stack

The first possible reason that came to my mind was the stack. All recursive programs are eventually affected by a stack overflow problem, as they use the stack for keeping track of the successive calls to the method.

Because I planned the algorithm based on a double recursive function, it was likely that this kind of problem could be present. Once the stack is overflow, the program usually stops working.

The first thing to do was checking what the maximum occupancy of the stack was. For doing that, I located some counters inside the method which kept track of the stack level and returned the result in the output file.

The results are shown in the histogram of Figure 10 for a depth of the predictive algorithm of 3. Remember that all the simulations with depth over 2 steps were crashing at some point.

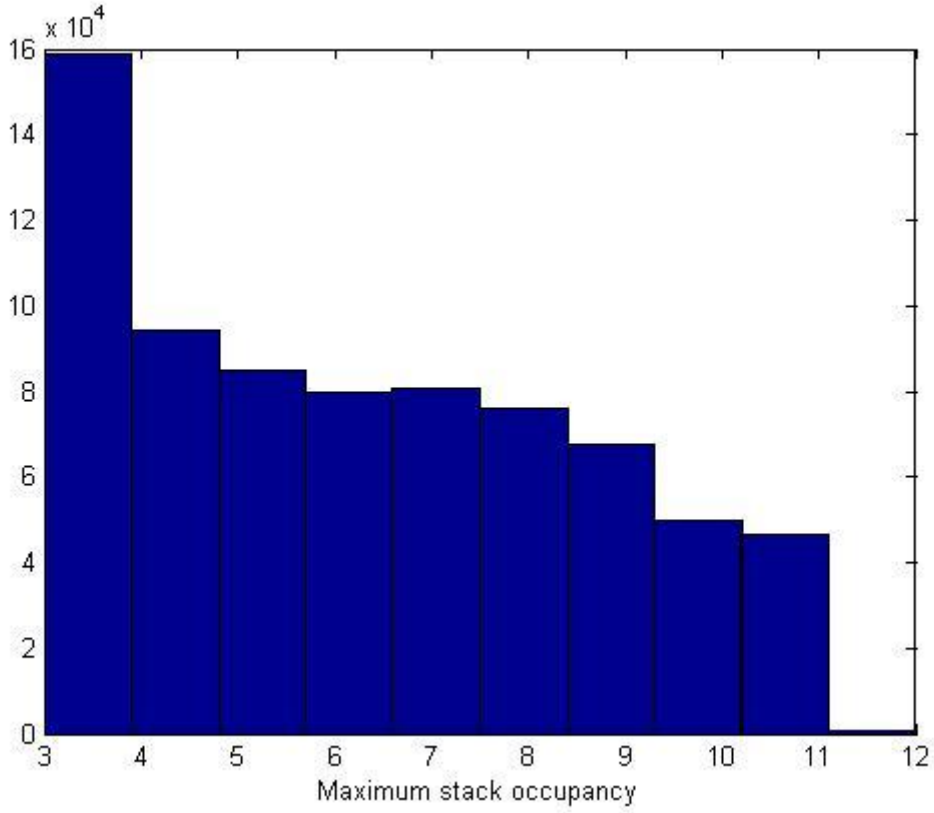


Figure 10. Histogram of the maximum stack occupancy for predictive depth = 3.

The maximum stack occupancy during all the execution was 12 levels, a quantity that does not justify the failing of the program. Anyways, we could not thing in other problem apart from that, so we found ourselves wandering if the stack overflow was not registered in these statistics.

Monajemi suggested that maybe the problem was in how the crossroads were designed. So far, every time a road was drawn in the scenario, two parallel roads were created, one for each direction and joint in their extremes.

The separation between both was little but non-zero. Now think about any crossroad. In the intersection of two roads a small square was created and there was any restriction for turning to any direction.

This means that when the prediction algorithm is applied and the Mobile Station is arriving to a crossroad, the path of looping around that square is there, as the sides of it are very small and the traveled distance during the prediction period is high.

In this case, it is easy to have a problem of stack overflow. I had two good reasons for believing that the stack was not overflow: a) the statistics did not show that so far and b) even if the traveled distance during the prediction was high, it was divided in steps, and the call to the recursive methods were independent for each step and the stack was reset.

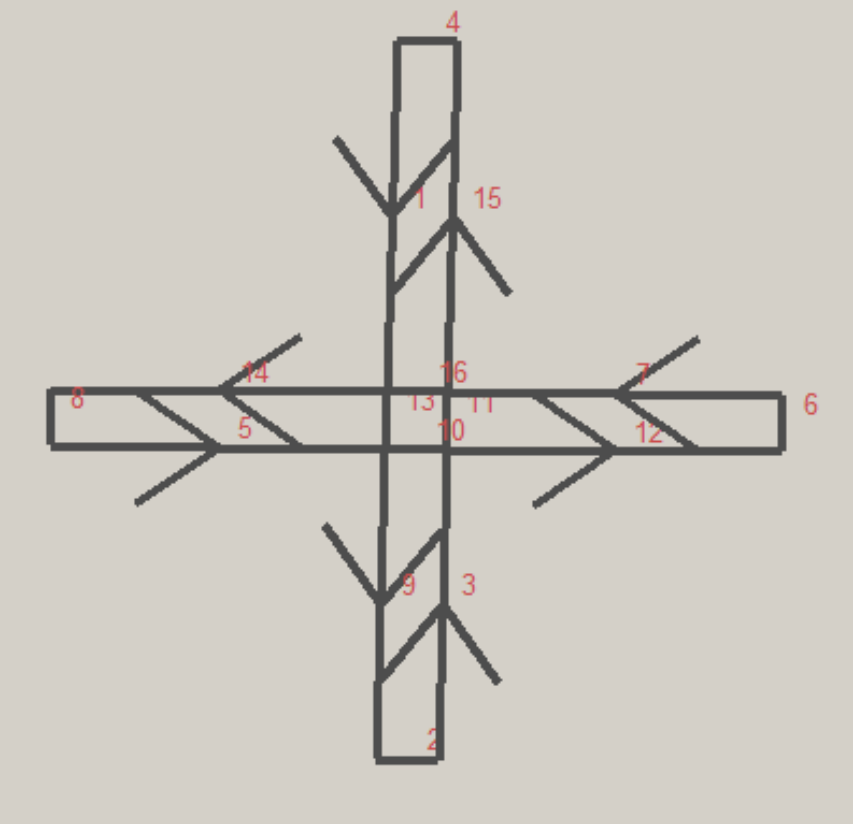


Figure 11. Example of crossroad in the old scenario designer.

Figure 12. Example of crossroad in the new scenario designer.

Anyways, even if it did not cause a stack overflow problem, the crossroad problem was important by itself. It is a waste of resources having those kinds of "square crossroads" in a simulator, as in real life no much Mobile Station are going to turn around in a crossroad.

That's why we decided to adapt the roads for making both directions coincident in space (Figure 12). This change had to be made in Matlab, in the code for creating the scenario. As I had been working just in the C++ code, this modification was made by Monajemi. The C++ code did not need to be modified, as all the methods were just based on the location of each direction separately.

After apply this change, we could no longer work with the same scenarios, based on the old distribution, so for obtaining the results shown in next section we created a new scenario, which will be shown in that section.

7.4.2. Checking the Memory

Trying to run the failed simulations in other computer, I realized they crashed in a different point. That told us the problem was machine-dependent. Immediately, I think about a problem with the memory of the computer. But how was that possible if every method was independent from each other?

I checked the memory anyways. For doing that, I used the C++ *GetProcessMemoryInfo*, which returns an object with information about the memory used by one process, in this case the simulation process.

Inside the *PROCESS_MEMORY_COUNTERS_EX*, the information we need is contained in *PrivateUsage*, the Commit Charge value in bytes for this process. Commit Charge is the total amount of memory that the memory manager has committed for a running process.

Using this method, I gathered memory usage for every step in the simulation. The analyzed steps were the following ones:

- **Beginning.** Memory usage at the beginning of the simulation.
- **Network update, BS update, BS interference, BS scheduler, MS update, Channel update, Stats update, Output update.** Increasing in memory usage during the updating of the different parts of the simulation.
- **Final.** Memory usage at the end of the simulation.

- **Total.** The difference between the memory usage at the end and at the beginning.

After simulating, the results were as I expected: a lot of memory was used during the Mobile Station update. The number was so big that the memory was overflow and the simulation crashed.

For being sure that the problem was in the predictive algorithm, I run a simulation for Instantaneous Pilot. In this case the increase in memory was even negative for the Mobile Station update, so the problem was clearly in the prediction.

The solution was very simple, and I came with it after looking the code step by step. For collecting the information about the likelihood of each Base Station we used the class Position, as it was explained in the Winter Report. I though the objects of that class were erased once we exit the code, but I learnt that you need to destroy them explicitly.

After placing some *delete* lines in the code (see Section 7.3), the memory leakage problem was completely solved. Now we could simulate any scenario with any depth in the prediction, and the program would not fail.

8. Results

As we said before, after the crossroad modification we got the results from a new scenario, called *Scn_15May*, which can be seen in Figure 13. The black dots are Base Stations and the blue ones, Mobile Stations. It has 400 Mobile Stations and 45 Base Stations, distributed more or less uniformly.

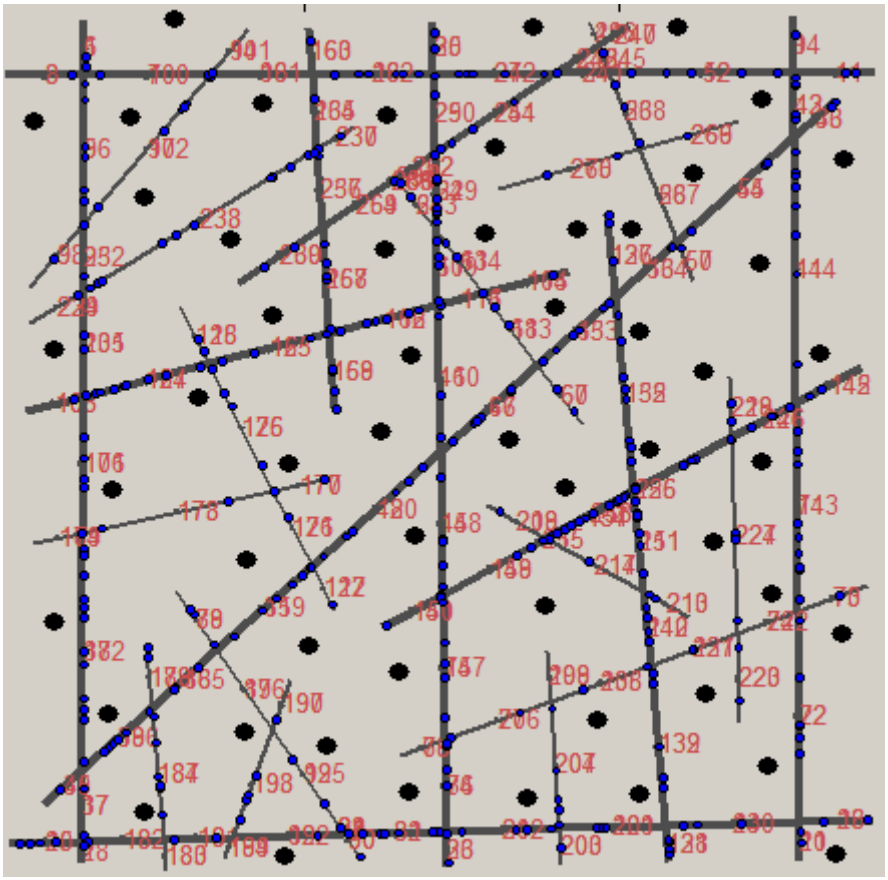


Figure 13. Scenario *Scn_15May*

We changed some parameters for studying the scenario's behavior in different circumstances so we can find the relationship between them and the results. We have changed the coverage radius, the depth of the predictive algorithm and the density of Femto Base Stations (0 in Figure 13).

We have simulated 600000 time slots, 10 minutes. For each single simulation we spent around 6 hours, depending also on some parameters as the depth or the density of Femto Base Stations (see following subsections).

8.1. Changing the Coverage Radius

Minimizing the number of hand-offs not only brings good consequences. There is at least a drawback: if the Mobile Station is not connected to the Base Station which gives it the maximum power, neither the throughput will be optimal, because both magnitudes are related. The point is that in mobile networks we just care about working above certain throughput threshold. As soon as the throughput is higher than that threshold, the communication will work perfectly and the objective is minimizing the number of hand-offs.

The first question to answer, as we did with the *Scn_Freeway* scenario was which could be the QoS threshold, i.e. Base Station's radius, for working above the minimum throughput limit. For that, we have simulated for different coverage radius with the predictive algorithm and instantaneous pilot. You can find the results in Figure 14.

As we increase the coverage radius, the number of hand-offs decrease. This is perfectly logical, because if the coverage radius is bigger, the Mobile Station will stay longer time under the coverage of the same Base Station, and it will need less hand-offs. The number of handoffs is now lower than in the instantaneous pilot case for every radius.

Moreover, the throughput is worse using the predictive algorithm. Furthermore, it is going down as the coverage radio is increased. That makes perfect sense as we said before, because if the coverage radius is bigger, we are allowing communications with less power, lowering the QoS threshold.

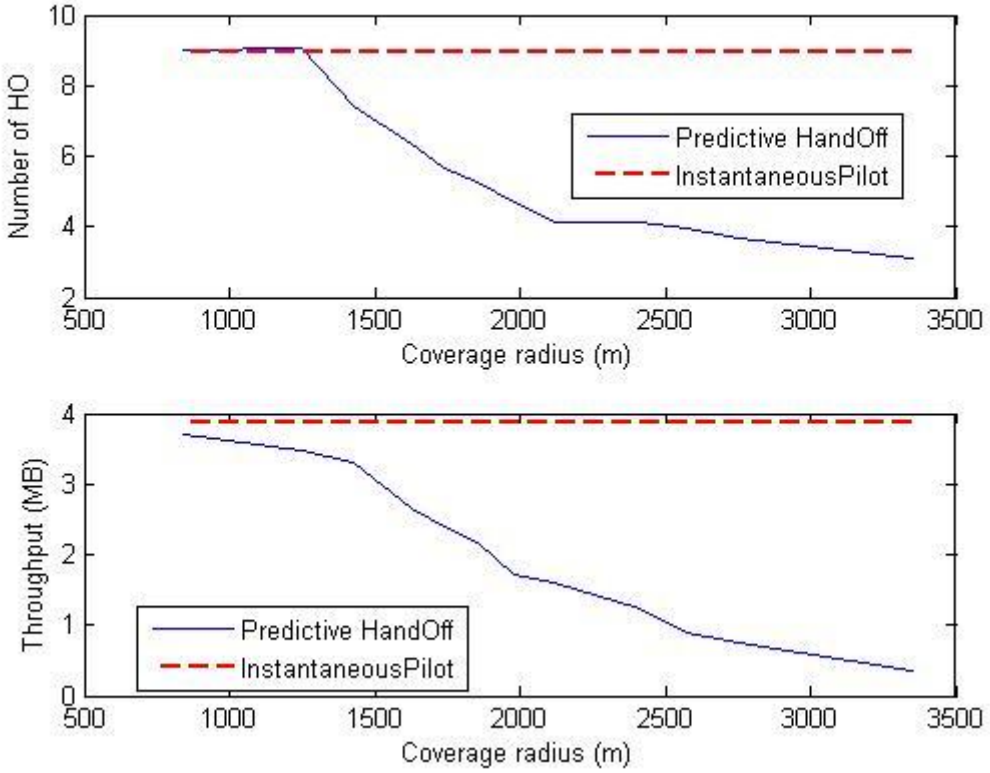


Figure 14. Average number of hand-offs and throughput as a function of coverage radius. Scn_15May scenario, predictive algorithm depth = 2

Looking at these results, we decided to take an standard radius of 2.1 km, which corresponds to a threshold of -108 dBm of minimum power, as we did with the old scenario before changing the way the road were linked.

8.2. Changing the Depth

Once the memory leakage problem is fixed, we can simulate for different depths of the predictive algorithm. The average number of handoffs and the throughput are shown in Figure 15 as a function of the depth.

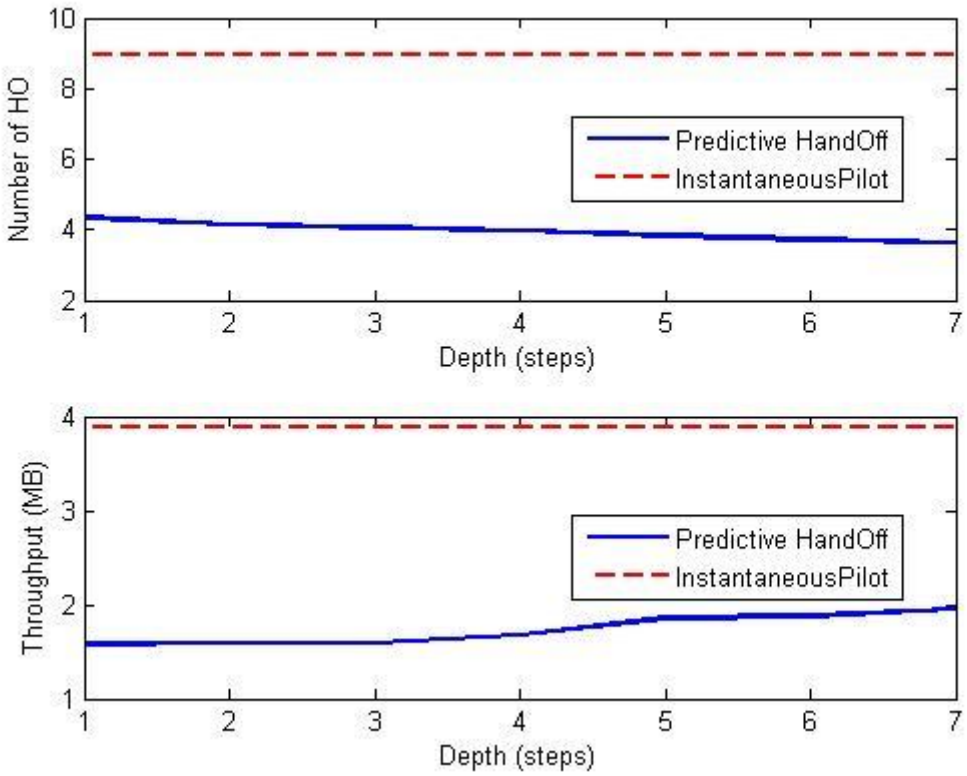


Figure 15. Average number of hand-offs and throughput as a function of algorithm depth. Scn_15May scenario, coverage radius ≈ 2.1 km

As we supposed, the number of handoffs is reduced as the depth of the algorithm is increased, because its accuracy in the prediction is better.

On the other hand, the throughput increases with the depth, which means that increasing the depth we obtain better results: less number of handoffs and more throughput. We do not know how to justify this theoretically, and it seems that it could be scenario-dependent.

We can see also a representation of the time elapsed for each simulation in Figure 16. It increases with the depth and it is higher than the time elapsed for the instantaneous pilot scheme, as we would expect. The time does not increase dramatically until depth 7.

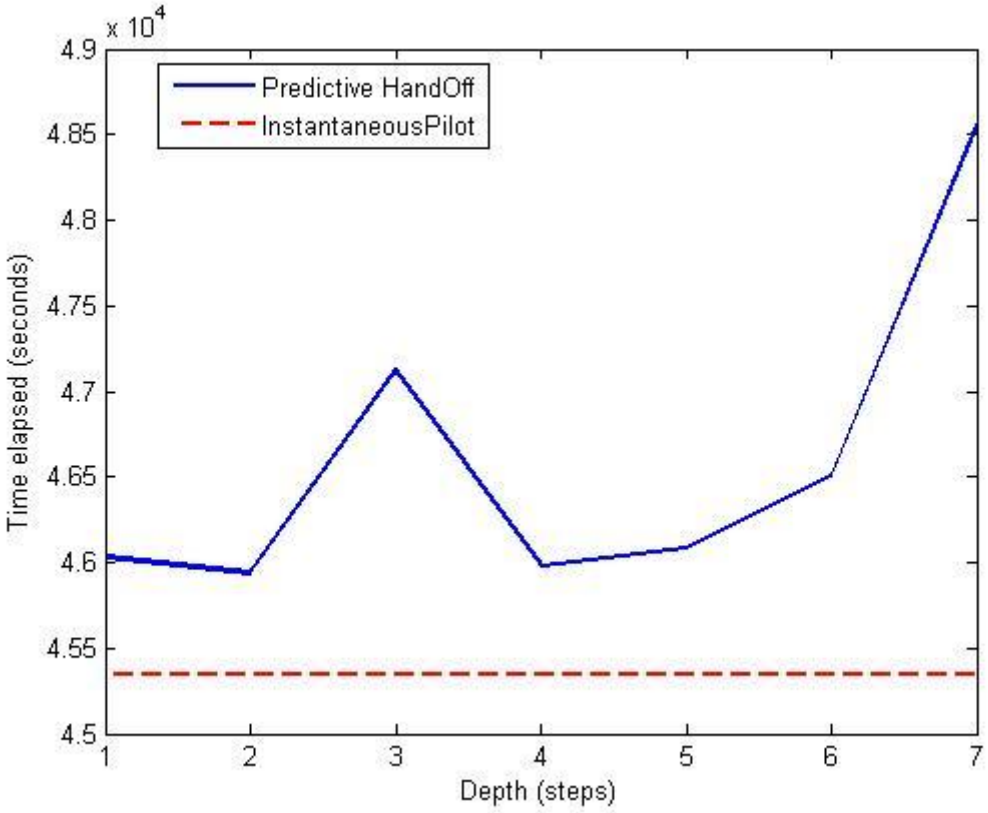


Figure 16. Time elapsed until the simulation is complete as a function of the depth of the predictive algorithm. *Scn_15May* scenario, coverage radius ≈ 2.1 km.

There is a strange peak at depth 3. We need to have in mind that this graph has to be read carefully, because the time elapsed does not just depend on the pure algorithm. It depends also on how many processes are running at the same time in the computer, how big they are, etc. Each time we run a simulation, the time elapsed is going to be different according to its environment, so probably when we run the depth 3 simulation, the machine was more busy than in the other cases. The simulations were very heavy and elapsed for a long time, so I could not repeat them before the academic year ended.

8.3. Changing the Number of Femto Base Stations

So far, our simulations did not include Femto Base Stations, base stations working at less power and, as a result, with a lower coverage ratio.

We have distributed uniformly the Femto Base Stations in the same scenario, *Scn_15May*, and we have simulated for different densities of Base Stations. For example, in Figure 17 the scenario with 54 Femto Base Stations is shown.

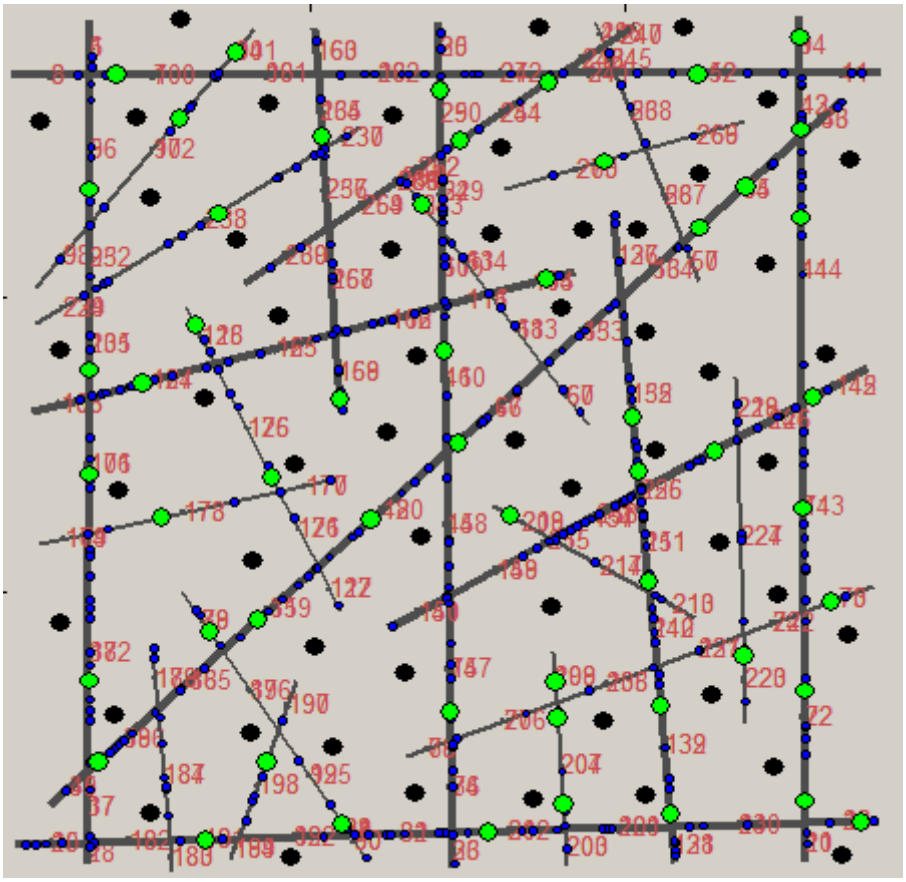


Figure 17. ***Scn_15May scenario with 54 additional Femto Base Stations***

The idea behind these simulations is proving that the predictive algorithm keeps the Mobile Stations always attached to the Macro Base Stations. We already know that under Instantaneous Pilot the Mobile Station is going to switch to a Femto Station as

soon as its power is the highest reachable. But Femto Base Stations has a really small coverage radius, and it is a waste of energy switch to them just for a little bit when the Mobile Station is going to leave its coverage area soon and there is another Macro Base Station giving coverage to the whole path.

The average number of handoffs and the throughput are shown in Figure 18. As we predict, the results for the Predictive case are always the same, while the number of handoffs and the throughput increase for the Instantaneous Pilot. We can say that the predictive algorithm works without caring about the femto level, which is useful when we are talking about roaming Mobile Stations, mainly when they are located in some kind of vehicle.

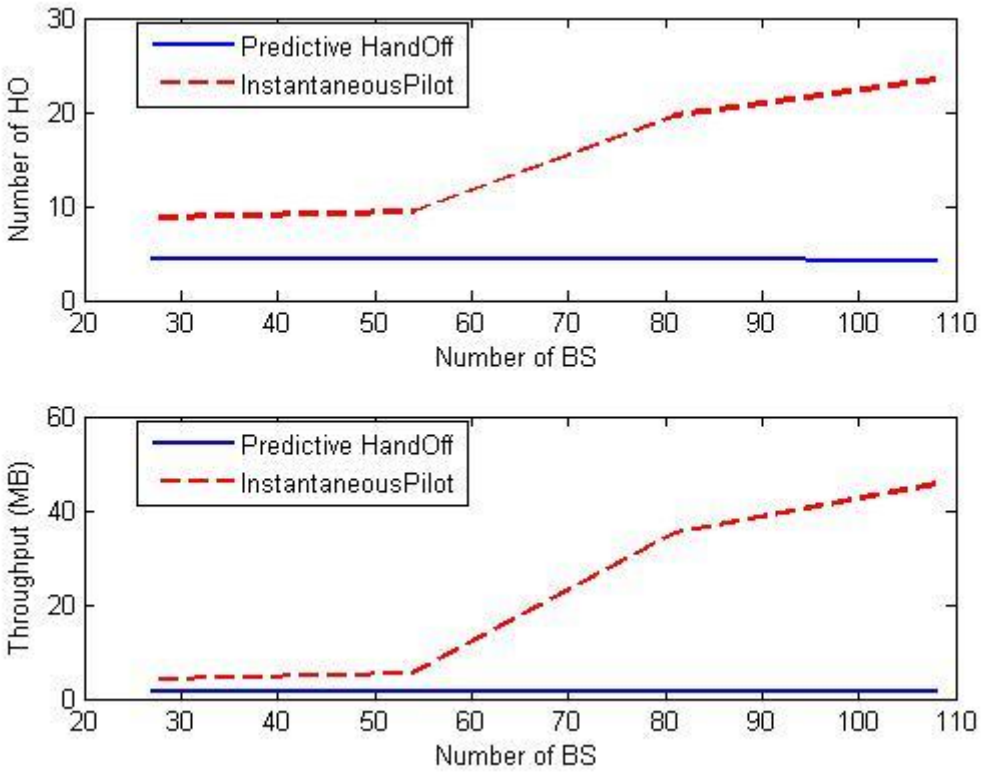


Figure 18. Average number of hand-offs and throughput as a function of the number of Femto Base Stations. Scn_15May scenario, coverage radius ≈ 2.1 km

9. Conclusion and Future Work

We have presented here a complete predictive algorithm for minimizing the number of hand-offs in mobile wireless networks, according to the work done during the Winter and Spring quarters. The results obtained from the simulations are totally successful, once we solved some issues from Winter quarter.

Our goal now is writing a journal report from the results obtained. We are currently working in it and if everything is ok we will finish it for the end of the academic year.

This simulation program allows us to implement many other details in the mobile networks for improving their performance in other aspects. For example, we planned to work in some new scheduling scheme for let the Mobile Stations share efficiently the bandwidth.

The seed is sown and the program is ready for being improved in order to obtain other important results. Now, it is time for us to handing over to others.

10. Bibliography

[1] Haitham Abu-Ghazaleh and Attahiru Sule, "Application of Mobility Prediction in Wireless Networks Using Markov Renewal Theory," *IEEE Transactions on Vehicular Technology*, vol. 59, no. 2, February 2010.

[2] Vicent Pla, José Manuel Giménez-Guzmán, Jorge Martínez, and Vicente Casares-Giner, "Optimal Admission Control Using Handover Prediction in Mobile

Cellular Networks," *Spanish Ministry of Sciences and Technology Documents*, 2003.

[3] Sherif Rashad, Mehmed Kantardzic, and Anup Kumar, "User mobility oriented predictive call admission control and resource reservation for next-generation mobile networks," *Journal of Parallel and Distributed Computing*, vol. 66, pp. 971-988, 2006.

[4] Eric Nagler, *Learning C++*, 3rd ed.: Thomson Brooks/Cole, 2004.

[5] J. Hou and Y. Fang, "Mobility-based call admission control schemes for wireless mobile networks," *Wireless Comm. Mobile Comput.*, vol. 1, no. 3, pp. 269-282, 2001.

[6] S. Chan and M. Zukerman Q. Huang, "Improving handoff QoS with or without mobility prediction," *IEE Electron. Lett.*, vol. 43, no. 9, pp. 534-535, 2007.

[7] J. François and G. Leduc, "Mobility prediction's influence on QoS in wireless networks: A study on a call admission algorithm," *Proc. WIOPT*, pp. 238-247, 2005.

[8] S. Zhou and A. Seneviratne J. Chan, "A QoS adaptive mobility prediction scheme for wireless networks," *Proc. IEEE GLOBECOM*, pp. 1414-1419, Nov. 1998.

[9] W.-Y. Kao, D. Shiung, C.-L. Liu, H.-Y. Chen and H.-Y. Gu H.-W. Ferng, "A dynamic resource reservation scheme with mobility prediction for wireless multimedia networks," *Proc. IEEE*, no. 60, pp. 3506-3510, Sept. 2004.

[10] H. Park and K. Lee S. Kwon, "A novel mobility prediction algorithm based on user movement history in wireless networks," *Proc. 3rd AsianSim*, pp. 419-428, Oct. 2004.

[11] S. Michaelis and C. Wietfeld, "Comparison of user mobility pattern prediction algorithms to increase handover trigger accuracy," *Proc. IEEE 63rd Veh. Technol. Conf.*, pp. 952-956, 2006.

[12] U. Deshpande, U. C. Kozat, D. Kotz and R. Jain L. Song, "Predictability of WLAN mobility and its effects on bandwidth provisioning," *Proc. IEEE INFOCOM*, pp. 1-13, 2006.

[13] M. H. Sun and D. M. Blough, "Mobility prediction using future knowledge," *Proc. 10th ACM Symp. Model., Anal. Simul. Wireless Mobile Sys.*, pp. 235-239, 2007.

[14] F. Yu and V. Leung, "Mobility-based predictive call admission control and bandwidth reservation in wireless cellular networks," *Comput. Netw.*, vol. 38, no. 5, pp. 577-589, 2002.

[15] I.F. Akyildiz and W. Wang, "The predictive user mobility profile framework for wireless multimedia networks," *IEEE/ACM Trans. Netw.*, vol. 12, no. 6, pp. 1021-1035, 2004.

[16] P. Bahl and I. Chlamtac T. Liu, "Mobility modeling, location tracking, and trajectory prediction in wireless ATM networks," *IEEE J. Sel. Areas Commun.*, vol. 16, no. 6, pp. 922-936, 1998.

[17] N. Samaan and A. Karmouch, "A mobility prediction architecture based on contextual knowledge and spatial conceptual maps," *IEEE Trans. Mobile Comput.*, vol. 4, no. 6, pp. 537-551, 2005.

[18] D. M. Rodriguez, C. Molina and K. Basu V. Bermudez, "Adaptability theory modeling of time variant subscriber distribution in cellular systems," *Proc. IEEE VTC*, vol. 3, pp. 1779-1784, 1999.

[19] J.-K. Lee and J. C. Hou, "Modeling steady-state and transient behaviors of user mobility: Formulation, analysis, and application," *Proc. 7th ACM MobiHoc*, pp. 85-96, 2006.

[20] H. Abu-Ghazaleh and A. S. Alfa, "Mobility prediction and spatial-temporal traffic estimation in wireless networks," *Proc. IEEE*, no. 67th, pp. 2203-2207.

[21] W. Soh and H. Kim, "QoS provisioning in cellular networks based on mobility prediction techniques," *IEEE Comm. Magazine*, pp. 86-92, 2003.