Bachelor Thesis Nr. 317
Design and Development of a Location Privacy-preserving Android Application

Manuel Haag

Course of Study: Informatik B. Sc.

Examiner: Prof. Dr. Kurt Rothermel
Supervisor: M. Sc. Zohaib Riaz
Commenced: 7.3.2016

CR-Classification: D.1.5, H.3.4, I.2.8
Abstract

Imagine walking by a cosmetics store and having an advertisement pop up on your smartphone urging you to buy something right on spot. Imagine visiting a friend in a hospital and having a social network sharing your location without your knowledge. Within this work, an Android application was developed to protect your location privacy by detecting out-of-routine events and notifying you dependent on your privacy settings. No online connection or excessive data hoarding is required, it uses a stay location algorithm to determine whereabouts and an instantaneous entropy predictor to calculate an out-of-routine measurement.
Acknowledgements

I want to thank my supervisor Zohaib Riaz for his continued support during both the research and development phase of this bachelor thesis. Furthermore, I want to thank the people of the Institute of Parallel and Distributed Systems for giving me the opportunity to use university facilities and devices during the application development.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>Integrating the optimized algorithms</td>
<td>24</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Data model</td>
<td>24</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Algorithm implementation</td>
<td>26</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Debugging tools</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Evaluation</td>
<td>28</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction of the data set</td>
<td>28</td>
</tr>
<tr>
<td>6.2</td>
<td>Pre-processing</td>
<td>29</td>
</tr>
<tr>
<td>6.3</td>
<td>Performance researches</td>
<td>30</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Comparison of instantaneous entropy values</td>
<td>30</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Comparison of calculation times</td>
<td>31</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Data growth progression [1] .......................... 1
1.2 App categories - usage distributions at various entropy levels .......................... 3
2.1 Flowchart of privacy protection options .......................... 5
2.2 Comparison of $\log_2(i)$ and $H_i$ for the sample sequence .......................... 8
4.1 Sample map of a user’s path (Source: Google Maps) .......................... 14
4.2 UML diagram of the EntropyValue class .......................... 17
5.1 The main map frame .......................... 22
5.2 Extra actions from the status bar .......................... 23
5.3 Settings screen of the app .......................... 23
5.4 Out-of-routine event notification .......................... 24
5.5 UML diagram showing the symbolizable DataNode relation model .......................... 25
5.6 UML diagram showing the implementation of StayLocation-Series .......................... 26
6.1 Stay location calculation times .......................... 29
6.2 Instantaneous entropy values evaluation .......................... 30
6.3 Instantaneous entropy calculation times .......................... 31
List of Tables

2.1 Instantaneous entropy algorithm example steps . . . . . . . . 8
3.1 Usability heuristics . . . . . . . . . . . . . . . . . . . . . . 11
4.1 Stay location sliding window functions . . . . . . . . . . . . 14
List of Abbreviations

API Application Programming Interface
APIK Android Application Package
GC Garbage Collector
GPS Global Positioning System
JSON JavaScript Object Notation
LBA Location Based Application
LZE Lempel-Ziv Estimator
POI Point of Interest
UML Unified Modeling Language
Chapter 1

Introduction

In the last decades, the demand of the industry for user data has increased many times over. Today’s mobile big data controls some of the largest modern processes, as collecting more and more data opens up new perspectives in terms of user-targeted services. App creators are currently in the process of discovering the advantages of context aware apps, which are able to detect a user’s environment and offer services based on the user’s circumstances.

Figure 1.1: Data growth progression [1]
As the above figure shows, the global data collection is increasing from year to year, estimations for the next years assume an exponential growth. This data comes from the user, often without his knowledge or approval. This work will focus on privacy concerns regarding location based applications (LBA). This chapter will introduce the reader to the current situation on context aware apps, specific privacy concerns and how this work contributes to the status quo.

1.1 Context aware apps

Context aware apps are defining our lives more and more with every day, as their usability has sparked keen interest in companies all around the world. One of the most popular modern examples would be Apple, as they introduced location-based app suggestions with iOS 8 [5]. This feature recommends the user certain apps depending on the users position, such as a navigation system when being at work around closing time. Another big field of context aware uses is for advertising purposes. Imagine walking by a cosmetic store and have an advertisement popping up on your smartphone urging you to buy something right on the spot. This kind of context aware apps depends on detecting your environment by accessing location services on your phone. In addition to that, it might be of interest to the reader that app usage distributions vary at different 'predictability' rates of the user [6]:
It is shown in particular the need for more search orientated apps in unusual environments, on the contrary, with more habitual conditions, recurring functionalities as setting a timer are used more often. We will take a deeper look at this in a later chapter.

1.2 Privacy concerns

While context aware apps can certainly be of use for the majority of people, it also raises several ethical questions regarding data privacy. It is well-known that the location of a user reveals a lot about his habits and practices, and letting the user pre-define situations where he does not want his location to be shared is nearly impossible - as predicting unpredictable situations is a paradox on itself.

The contextual environment information also has to be stored in some way, making it vulnerable to data theft. Often the data is exchanged with an online web service endpoint or even published without the user’s clear agreement or knowledge. There is almost always a secondary, unanticipated use of the data by companies being taken over by another one or simply the data collecting company selling your data for business purposes.
1.3 Out-of-routine detection

This work tries to answer the need for privacy with automated processing. More precisely, we will build an algorithm to detect out-of-routine situations, as those are the most important to keep private. Then we will give the user a notification to turn off location services, this preserves the users decision-making. The goal is to achieve a situation where the user is satisfied with the background work of the app, but has the final word about important decisions. Our algorithm will be placed within an mobile app, as for real-time privacy preserving a portable device is necessary.

1.4 Contributions

The architecture of our app requires it to operate in the background and give notice on out-of-routine events, as taking over the systems location services is neither a suitable nor achievable target. Therefore, this app acts as a simple notification system and not as an executive power. One may argue, that our algorithm will need access to the user’s location at all times as well, but it is impossible to detect sensitive data without knowing the data itself. Nonetheless, we will try to detect private events as soon as possible and attempt to notify the user before any harmful damage occurs and let our algorithm work independently, thus no online connection will be built to send private data against an API and the like.
Chapter 2

Background and related work

In this chapter, we will talk about existing work and try to give an overview on current ways to protect the user’s privacy. We will go deeper into the automated kind of privacy preserving algorithms and analyze a real-time predictability estimator based on a Lempel-Ziv estimator.

2.1 A high-level classification of approaches

There are several kinds of privacy protection available. The flowchart below tries to dissect the various possibilities:

Figure 2.1: Flowchart of privacy protection options
It appears from the flowchart that there are two main possibilities of privacy protection, one being the manual way by letting the user define his own preferences beforehand and the other one being the automated process of detecting private events and letting the user know right on place.

2.2 User defined privacy preferences

If the user wants to define personal settings prohibiting location sharing on his own, he could do that based on either temporal, semantic or spatial properties. For example, he could schedule a deactivation of location services every day between 10.00 pm and 6.00 am at sleeping times (temporal), or could prevent social networks from sharing his location at certain POI categories like medical facilities (semantic).

For this kind of privacy protection, there are already solutions available notifying you at certain times (alarm clock) or certain locations (navigation systems, Foursquare).

2.3 Automated privacy protection

To automate the user’s privacy protection, one has to either implement an artificial intelligence feeding on the user’s input data or create an out-of-routine detection mechanism. While the machine learning part relies on large amounts of data to be able to work correctly, the out-of-routine detection can work right ahead but needs a learning period as well to study the user’s routine and improve prediction hit rates.

The advantage of automated approaches clearly is the independence from the user, he does not need to setup complex settings and such. The app gets shipped out in the same state to every unique user, but changes its behavior dependent on his location profile. Therefore, this work will focus on detecting out-of-routine events by checking the user’s time and location without requiring a configuration process.

2.4 Mobility prediction using instantaneous entropy

The instantaneous entropy approach is based on raw singleton symbols, therefore it only takes the order of sequences into consideration, not the time or location attributes. To formalize predictability of the user’s movement,
it deals with a random process $X = \{X_0, X_1, \ldots, X_N\}$. The definition of
Shannon entropy defines it as the independent entropy at each step, but we
want to take a look at conditional dependencies in between, as our steps are
related to each other.

There are many entropy rate estimators available, the Lempel-Ziv estimator
(LZE) from Song et al. [10] being the one of interest for us:

$$\hat{H}_N := \left( \frac{1}{N} \sum_{i=2}^{N} \frac{\Lambda_i}{\log_2(i)} \right)^{-1}, \quad (2.1)$$

where $\Lambda_i$ is defined as the length of the shortest subsequence starting at
position $i$ that did not previously occur in sequence $(x_1, \ldots, x_{i-1})$ [6]. This
increasing window LZE swiftly converges to the true entropy rate.

The issue with this estimator is it need for future data, which is not available
on a real-time mobile device. McInerney et al. [6] therefore developed a real-
time LZE for the instantaneous entropy using a reverse of equation 2.1:

$$\tilde{H}_i := \log_2(i), \quad (2.2)$$

where $\Gamma_i$ is defined as the length of the shortest subsequence ending at
position $i$ that did not previously occur in sequence $(x_1, \ldots, x_{i-\Gamma_i})$. They
point out that the reverse of a time series has the same entropy rate as the
original one.

To give an example, we are calculating the values for the sample sequence
$AABABCAB$: 

\begin{align*}
\hat{H}_N &:= \left( \frac{1}{N} \sum_{i=2}^{N} \frac{\Lambda_i}{\log_2(i)} \right)^{-1}, \\
\tilde{H}_i &:= \log_2(i).
\end{align*}
Table 2.1: Instantaneous entropy algorithm example steps

<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>$\Gamma_i$</th>
<th>$i$</th>
<th>$\log_2(i)$</th>
<th>$\hat{H}_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>1 (A)</td>
<td>0</td>
<td>-Infinity</td>
<td>-Infinity</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>2 (AA)</td>
<td>0</td>
<td>-Infinity</td>
<td>-Infinity</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>1 (B)</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>2 (BA)</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>2 (AB)</td>
<td>3</td>
<td>1.585</td>
<td>0.792</td>
</tr>
<tr>
<td>6</td>
<td>C</td>
<td>1 (C)</td>
<td>5</td>
<td>2.322</td>
<td>2.322</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>2 (CA)</td>
<td>5</td>
<td>2.322</td>
<td>1.161</td>
</tr>
<tr>
<td>8</td>
<td>B</td>
<td>3 (CAB)</td>
<td>5</td>
<td>2.322</td>
<td>0.774</td>
</tr>
</tbody>
</table>

Plotting $\log_2(i)$ and $\hat{H}_i$, we can quickly see a peak for new events and decreasing values, the more of the sequence is known:

Figure 2.2: Comparison of $\log_2(i)$ and $\hat{H}_i$ for the sample sequence

We can also see the $\log_2(i)$ being an upper limit for the final $\hat{H}_i$ value. This information will be valuable when looking for a threshold limit to de-
tect out-of-routine events.
Notice that this approach only takes the order of sequence into consideration and no additional parameters, therefore it is independent from shifts of the whole sequence or one-time events delaying further symbols. This is important if we apply this to a user’s daily path, if he moves to somewhere unexpected right before work and every following event on that day is delayed by a certain amount of time, only the unexpected event will have an exceptional high entropy value and not the normal course of actions afterwards.
Chapter 3

Objectives

In the previous chapters, we have clearly shown the need for a privacy-preserving mobile approach. For our application, we set the following targets:

1. **Background algorithm(s)** detecting out-of-routine events based on the users location and time

2. **Notification** of out-of-routine events by displaying a pop-up message

3. **Visualization** of the users stay location history using a map frame

4. **Settings** to configure algorithms and notifications

For the algorithm part, we have a few approaches already available from existing work — as demonstrated in the previous chapter. More precisely, we are going to implement an stay location detection algorithm and a out-of-routine resolution based on the instantaneous entropy prediction. Notifications will be realized by displaying a pop-up and visualization of the user’s stay locations will use a map fragment to display the users main spatial positions.

3.1 Optimization of algorithms

As both processing power and memory space is limited on a mobile device, all parts of the out-of-routine detection framework have to be optimized regarding to real-time usage. Large data sets have to be tested against the back-end, thresholds for out-of-routine events have to be dynamically adjusted according to the user’s specific behavior.
Time spans ranging from a few minutes to several months have to be considered as well as different phone performance setups. Since the app serves as a utility tool and not as an entertainment function, short processing times are a requirement. If we notice a user’s approaching to an unusual location, an information should be displayed within less of a second.

### 3.2 App design requirements

A few usability heuristics have to be kept in mind during the development of a mobile app. Jakob Nielsen worked out a set of ten basic rules which we are trying to match with our app’s elements [8]:

<table>
<thead>
<tr>
<th>Visibility of system status</th>
<th>Map frame displaying the user’s path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match between system and real word</td>
<td>Real-time location processing</td>
</tr>
<tr>
<td>User control and freedom</td>
<td>Configurable privacy settings</td>
</tr>
<tr>
<td>Consistency and standards</td>
<td>Usage of well-known APIs</td>
</tr>
<tr>
<td>Error prevention</td>
<td>Fail-safe exception handling</td>
</tr>
<tr>
<td>Recognition rather than recall</td>
<td>Detection of unrecoverable events</td>
</tr>
<tr>
<td>Flexibility and efficiency of use</td>
<td>Focus on the main necessary parts</td>
</tr>
<tr>
<td>Aesthetic and minimalist design</td>
<td>Up-to-date interface elements</td>
</tr>
<tr>
<td>Help users recognize, diagnose, and recover from errors</td>
<td>Error logging</td>
</tr>
<tr>
<td>Help and documentation</td>
<td>This written work</td>
</tr>
</tbody>
</table>

Table 3.1: Usability heuristics

Furthermore, we will want the user to have the final decision regarding
whether to hide his location or not. The app should also be able to work independently on the device, so no data is shared with any online host. The only input data taken from the user should be time and location, to keep the risk of having vulnerable sensible data as low as possible.

3.2.1 Input pre-processing

Our input is defined as time-stamped locations. As those data nodes come with no semantic information at all, we have to calculate a symbol for each location using a stay location algorithm. The user’s most common locations have to be detected and merged if approximated again. After all, we want to focus on actual visit’s of places rather than the traveling time in between.

3.2.2 Detection of out-of-routine

To actually detect out-of-routine events, we need some kind of value to measure the user’s routine. If a value exceeds the usual amount, it will be thrown as an out-of-routine event. Identification of those cases will be done based on the stay locations calculated in the previous step. If the user is currently staying and the mentioned ‘routine value’ is above a dynamically adjusted average threshold, a notification will be sent.
Chapter 4

Algorithms analysis and optimization

Following the positioning of our objectives in the previous chapter, we are going to analyze the requirements more deeply and optimize the objectives for our usages. This chapter will focus on dissecting and optimizing all algorithmic parts of our privacy-preserving framework, from stay location parsing to the actual out-of-routine detection and notification.

4.1 Stay location detection

For the purpose of this section, we are going to assume that user data is already available in the form of temporal and spatial fields, or to be more precise, single nodes with a timestamp and geocode with a fixed time interval between each.

An example route would look like this:
Our goal is to detect stay points of the present path history, to achieve that we are merging popular areas of the user into one significant geocode. To achieve this, we construct a sliding window based on a LinkedList for incoming data nodes with the following properties:

<table>
<thead>
<tr>
<th>Return type</th>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long</td>
<td>getTimespan()</td>
<td>Calculates the time span between the first and last data node in the sliding window</td>
</tr>
<tr>
<td>LatLng</td>
<td>getAverageGeocode()</td>
<td>Calculates the average geocode of all data nodes in the sliding window</td>
</tr>
<tr>
<td>List of DataNode</td>
<td>getInRadius()</td>
<td>Returns all data nodes within the average geocode radius</td>
</tr>
<tr>
<td>Integer</td>
<td>getSatisfaction()</td>
<td>Calculates the amount of data nodes within the average geocode radius in percent</td>
</tr>
</tbody>
</table>

Table 4.1: Stay location sliding window functions

Now we have to set three limits regarding our sliding window:

1. **Time limit**: If the sliding window time span exceeds this amount,
update the average geocode and satisfaction values. This is effectively
the minimum amount of time for the user to be considered staying at
one place. We set this to 5 minutes.

2. **Radius limit**: The `getInRadius`-function will return all nodes within
this radius of the average geocode. Basically the radius the user can
freely move within and still be considered as one stay location. We set
this to 100 meters.

3. **Satisfaction limit**: We will assume the sliding window is ‘satisfied’
if more than this percent of data nodes within it are within the radius
of the average geocode. We set this to 90%.

We can now construct a *Stay Location* algorithm using the constituted
sliding window. It will process nodes one at a time and assign a stay location
to every node belonging to a stay location:

```java
void addNode(final DataNode node) {

    // Add the node to end of the slidingWindow
    slidingWindow.offer(node);

    // Proceed if the time span is above the threshold
    if (slidingWindow.getTimespan >= TIME_LIMIT) {

        // Check if the user is currently at a stay location
        if (slidingWindow.getSatisfaction() >= SATISFACTION_LIMIT) {
            satisfied = true;
        }

        // If the user is leaving a stay location, close the
        slidingWindow
        else if (satisfied) {

            // Assign a stay location to each node in radius
            slidingWindow.getInRadius().forEach(DataNode::
                setStayLocation(StayLocation.find(slidingWindow
                    .getAverageGeocode())));

            // Renew the window and set satisfied to false
        }
    }
}
```
By waiting until the sliding window is not satisfied anymore instead of closing it right ahead, we make sure that if a user is staying at a location for more than 5 minutes, the average geocode and the list of nodes in its radius will still be correct. Notice that the static function `StayLocation.find(geocode)` either selects an existing stay location object – if the given geocode is within the radius limit of it – or constructs a new one. Every StayLocation has a unique symbol field representing the location’s symbol. We also keep track of all found stay locations in an `alphabet` collection.

### 4.2 Out-of-routine detection

Now that we are able to delay incoming data nodes with a symbol according to their stay location existence, we can parse those data nodes to detect out-of-routine movement by filtering nodes with a high entropy value.

To accomplish this objective, we worked out two approaches. The algorithms shown in chapter 2 are already advanced enough to be built on, though they require some fine-tuning regarding performance and memory usage.

#### 4.2.1 Instantaneous entropy

The instantaneous entropy approach measures the predictability of the user with entropy values. Opposed to the raw char data used in chapter 2, we now have to deal with stay location objects. We handle this by assigning a symbol to each stay location and concatenating stay locations in a list. Thereby the length of a string relates to the size of a list.

We construct an `EntropyValue` object to store the fields we need:
Every data node will be mapped to an associated `EntropyValue`. Our code to examine a single data node therefore will look like this:

```java
// Initialize a map to assign an EntropyValue to every DataNode
private final LinkedHashMap<T, EntropyValue> entropyValues = new LinkedHashMap<>();

// Initialize a collection to store the list of previously seen sequences
private final Set<String> sequenceLibrary = new HashSet<>();

void addNode(final DataNode node) {
    // Initialize a new EntropyValue for the node
    EntropyValue entropyValue = new EntropyValue();

    // Set the node's symbol as the end of the shortestSequenceEndingHere
    entropyValue.shortestSequenceEndingHere = node.getSymbol();

    // Trace the sequences backwards
    for (int backwardsIndex = entropyValues.size() - 1; backwardsIndex >= 0; backwardsIndex--) {
        // Push the node stay location symbol to the beginning of the shortestSequenceEndingHere
```
Now that we are able to calculate the $h$ for a given data node, we can determine the user’s ‘out-of-routine value’. And if we compute the average $h$ of all nodes, we can compare a given node with it and figure out whether it is out-of-routine (above average) or in-routine (below average).

### 4.3 Efficiency problems

For large data sets, the instantaneous entropy approach leads to inevitable performance problems as the library of previous sequences eventually grows too fast if the user is constantly visiting new places. With every new incoming data node, the library of previously seen sequences has to be crawled through comparing it with the current sequence, making the process slower with every new location. As the sequence library is stored using a RAMDirectory, unlimited growing
is not desired plus Java’s garbage collector will slow down the entire system at high RAM load, even OutOfMemory errors are highly possible.

### 4.4 Performance optimization

To counter the efficiency problems discussed in the previous section, we try to improve the algorithm’s conduct by implementing two attenuations:

1. Limit the size of the library of previously seen sequences
2. Remove old, rare sequences

The first item is almost self-explaining, as the actual size of the shortest subsequence ending there, which was not previously seen, does not matter — as long as it is large enough. We do not mind the absolute entropy value, as we just check whether it is above or below the average. After a short 'learning period', the spikes from out-of-routine values are large enough to be determined easily. Therefore, we limit the library’s size to

\[
LibrarySizeLimit = \frac{\text{StayLocationTimeLimit}}{\text{DataNodeInterval}}
\]  

(4.1)

Accordingly, we insert the following conditional break into our code fragment at line 20:

```java
1 if (entropyValue.shortestSequenceEndingHere.size() >= LIBRARY_SIZE_LIMIT) {
2   break;
3 }
```

Secondly, to prevent the influence of one-time events, we are setting an expiration timestamp on every sequence. If the sequence occurs again, the timer gets refreshed. If not, it gets removed from the library after a fixed amount of time — this way, we prevent infinite storage of dead sequences. After all, if the sequence should occur again, it will be over the entropy threshold anyway. As our first improvement kind of prevents searching for those old sequences already, this is more of a memory enhancement then a performance upgrade.

We achieve the sequence expiration by using a Cache structure as the Google Guava library offers it [3]:

With this new cache structure replacing our old `sequenceLibrary` collection, we can now store a sequence together with the timestamp of incidence. Entries are automatically being removed after a two weeks period, as this amount of time should be suitable for recurring in-routine events.
For the actual app, there are two main system architectures to choose from nowadays: Android from Google or iOS from Apple. For this work, Android was chosen as the desirable platform because of the large reach of Android-compatible devices and the easy portability due to the Java programming language. For the target platform version, Android 6.0 (Marshmallow) with API Level 23 was selected, as it offers the most up-to-date tools regarding maps and JDK features. Currently, 15.2\% of all Android devices use Android 6.0 according to the Android Developers Dashboard [4].

5.1 Interface

The main layout of our app will consist of a Android-typical status bar at the top and a map frame below. An ‘Extras’ button will be placed at the usual position for the settings activity — on the right of the status bar. To notify the user of important system alerts and out-of-routine events, Android Toasts will be used.

5.1.1 Google Maps API

As the app was generated for the Android OS, using Google Maps API to display the user’s location was the natural choice. Android devices usually come with the Google Play Services pre-installed, therefore the functionality for maps was already there and ready to be used. On rooted devices, there
are multiple providers for Google Apps as well. To visualize the user’s path and stay locations, map nodes were used with an *OnClick* feature displaying more information about the selected geocode like the internal ID, exact position and overall time spent there.

![Figure 5.1: The main map frame](image)

5.1.2 Configuration options

Three main configuration options had to be considered while creating the settings activity. At first, the user needs to be able to save and load his path history to enable portability of profiles. Secondly, algorithm parameters like time, radius and satisfaction limit should be customizable. Thirdly and lastly, the possibility to define personal notification settings was imple-
mented.

![Location Privacy](image1)

**Figure 5.2:** Extra actions from the status bar

The following screen shows the options given to the user:

![Settings screen of the app](image2)

**Figure 5.3:** Settings screen of the app

Settings are stored using the Android built-in `PreferenceManager`, user
profiles are ex- and imported using the JSON data interchange language.

5.1.3 Notification

Notifications are realized using Android Toast messages. As seen in Figure 5.3, the user can select to be notified about the following events:

1. New stay location detected
2. Out-of-routine event
3. In-routine event

An example notification can be seen in the following figure:

Figure 5.4: Out-of-routine event notification

5.2 Integrating the optimized algorithms

Now that we have the visual front-end settled, we need to integrate our optimized algorithms into our application. The following subsection will take a look at the whole data node model, the implemented algorithms and finally the benchmarking tools used to measure the algorithm’s performance.

5.2.1 Data model

To keep our application easily extendable for other data formats, a generic data structure regarding our entropy algorithms was needed. The UML diagram below shows the raw hierarchical structure:
The `Symbol` interface serves as the input class for our entropy algorithms, with its only function being the `getId()` one returning the unique identifier of the symbol as a String. Our `StayLocation` class object implements that interface and defines a geocode as an additional field.

The `Symbolizable` interface serves as a superior class to the `Symbol` one, while `DataNode` implementing it is our real timestamped location node representing the user’s movement.

Using those interfaces, we can construct a `StayLocationSeries` as follows:
Figure 5.6: UML diagram showing the implementation of StayLocationSeries

Notice the StayLocationSeries having a NIL_SYMBOL for unimportant data nodes. It also keeps track of all nodes in the nodes collection and the set of stay locations within the alphabet set.

With the discussed data structures we are also flexible about the input data, we could use raw text data as well — as long as it has a getSymbol method for each node.

5.2.2 Algorithm implementation

Our algorithm from the previous chapter have to be modified a little to make it work in real-time. The sliding window currently only assigns stay location symbols to the nodes within after it has closed, we need to copy the command to set the stay locations from the location-leaving part to the satisfaction recognition.

Additionally, to be able to save and reload paths we add a functionality in the settings menu to store the location history using the JSON serializer GSON from Google.

It needs to be considered that data capture is often paused or interrupted for a while. We will treat those missing data periods by ignoring them and going back to the last known location, for example in sequence AAB***CA
we will jump to B after backtracing from C, the star symbol representing either missing data or traveling information from the NIL_SYMBOL. Ultimately, we have to define our threshold for out-of-routine events. We do that by calculating the average $\hat{H}$ of all nodes in history and comparing it to the most recent value. If the value is above the average and larger than the one before, we will give the mentioned notification to the user reminding him to turn off his location services.

5.2.3 Debugging tools

To measure our algorithmic parts performances, we use the Java system command `System.currentTimeMillis()` before and after each processing part to calculate the time difference needed. Notice that the Java garbage collector (GC) influences the result by small amounts if a certain memory threshold is reached, though its impact is mainly negligible with modern memory space amounts.
Chapter 6

Evaluation

As a clear evaluation of all constructed parts is almost as important as the work itself, we will want to analyze the performance and behavior of each of our algorithmic fragments.

We will start by introducing sample data sets to test our app and inspect the stay location and out-of-routine detection part of the application.

All test were done using an emulated Nexus 5 device with 1536 MB of RAM and 64 MB of VM heap size.

6.1 Introduction of the data set

As predicting the user’s movement has been a valuable task in the past, luckily for us there are a few large data sets available for scientific research. The one we used is the GeoLife GPS Trajectories data set from Microsoft [7]. It ranges over a time span of over three years and has high variety of user profiles from the Microsoft Research Asia Geolife project.

The files are arranged as user directories containing multiple PLT files representing the user’s history. The PLT files are based on the GPX Exchange Format and contain one node per line with spatial and temporal information. We parse those files using a RegEx matcher and convert the lines to DataNodes. The lines are arranged in a 5 second interval, making it easy to gradually analyze their contents.

This way, we are able to test our application against large data banks without having to distribute it to actual people beforehand.
6.2 Pre-processing

The DataNodes generated from our PLT parser have to be convert to Stay-Locations before entropy values can be calculated. The following graph displays the calculation time per node in relation to the amount of stay locations found:

![Figure 6.1: Stay location calculation times](image)

It is clearly visible that there is no correlation between both. The spikes in calculation time are related to Java’s garbage collector, which cleans up the memory at certain satisfaction levels. As the memory space needed is nearly constant due to the sliding window used, shifts in calculation time can only be assumed to be dependent on other processes running. In average, 1.6 milliseconds were needed per data node. This is a suitable amount of time on a mobile device, as the memory amount is very small as well, we are satisfied with this result.
6.3 Performance researches

Next we will take a look at calculation times and value results of our instantaneous entropy out-of-routine detection algorithm. As we improved our algorithm by implementing limits on the mobile device, we are interested in seeing the gained performance.

6.3.1 Comparison of instantaneous entropy values

We already worked out the mechanism behind the instantaneous entropy calculation giving peaks on out-of-routine events and converging to zero on habitual behavior. The following chart illustrates the progress of $\tilde{H}_i := \log_2(i) / \Gamma_i$ in comparison to its dynamically adjusted average value:

![Instantaneous entropy values chart](chart.png)

If we take a look at the data behind, we recognize every high peak being a new stay location, every low peak being a long time abandoned location revisited. Our out-of-routine detection gives the alarm if a value is above the average and higher that the one before, after a certain learning period.
spikes of previously seen sequences become lower in value. In our example, the spike at about 1200 nodes origins from the same location as the one at around 170 nodes, being lower in value because it is not entirely new to the algorithm.

### 6.3.2 Comparison of calculation times

To compare the calculation time per node needed, we store the time difference between the `addNode()`-function beginning and end. While the blue line displays the original approach without any improvements, the orange line shows the optimized version.

![Instantaneous entropy calculation times](image)

**Figure 6.3: Instantaneous entropy calculation times**

We can see the spikes from the Java garbage collector being much lower with the optimized version, as the memory amount needed goes down by a factor of 10. From 450 MB of RAM needed for the calculation of 1000 entropy values to 50 MB in the optimized version, garbage collection events are much rarer and faster overall. With an even larger amount of data nodes available, the unoptimized approach will even run into `OutOfMemory` exceptions.
In average, the optimized algorithm takes around 1.4 milliseconds per data node on the Nexus 5 device used. Opposed to other calculation-heavy applications, this is a suitable time for a real-time background calculation. Keep in mind that our algorithm does not rely on large data sets to learn from, thus the overall space needed is very small.
Bibliography


Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Signature: