Filippo Bertozzi - 0000925096
filippo.bertozzi3@studio.unibo.it

Report LAM 2022/2023

Cellular Connectivity and Noise Map

Overview

L'elaborato si presenta in due principali activity: l'activity iniziale (MainActivity.kt) e una seconda activity, che alla sua base ha la familiare mappa di Google Maps (GridAndMapActivity.kt), nella quale troviamo la grandissima maggioranza delle funzionalità principali dell'applicazione.

Il progetto è stato sviluppato utilizzando come linguaggio di programmazione Kotlin senza l'uso dell'innovativo Jetpack compose, sfruttando invece il metodo più "*classico*" di utilizzo del file XML per la definizione del layout della user interface.

Main Activity

Questa activity ricopre un doppio ruolo: in primis, è la schermata iniziale dell'applicazione; in secondo luogo, è anche utilizzata per gestire i dati registrati e alcune impostazioni che influenzano il funzionamento della successiva activity, oppure la possibilità di ricevere notifiche o meno dall'applicazione.

La prima schermata che si presenta davanti all'utente all'apertura dell'applicazione è molto semplice e minimale.
Il logo dell'applicazione occupa la maggior parte dello schermo; oltre a questo possiamo trovare 3 FAB (Floating Action Buttons), che sono stati una scelta ricorrente di design durante l'intero sviluppo, in quanto ho notevolmente apprezzato la loro comoditá e funzionalità.

Il primo FAB muterà la schermata dell'activity presentando un'interfaccia dove gestire i dati registrati durante l'uso dell'app.

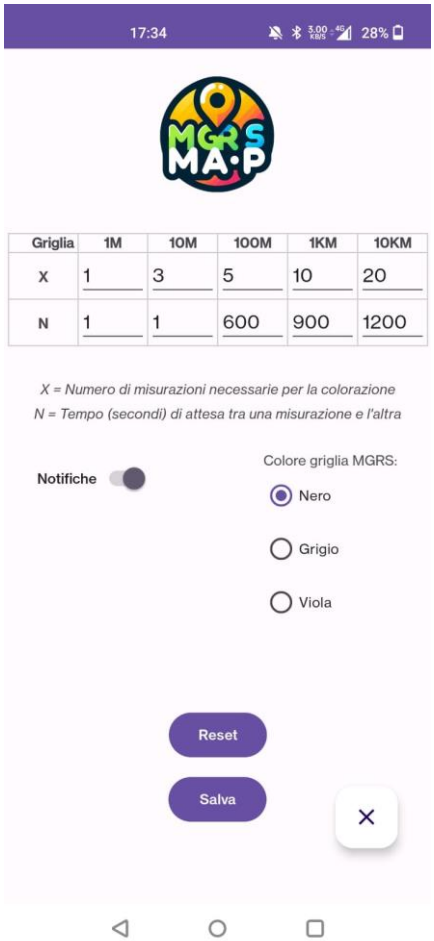
Il secondo ci permetterà di spostarci alla seconda activity che ha come elemento principale la mappa, analogamente all'icona del bottone.

Il terzo ed ultimo FAB, come il primo, permette di cambiare la schermata dell'attuale activity e mostrare un'interfaccia dove poter gestire alcune impostazioni della mappa e l'attivazione o meno delle notifiche.

Questa è la UI che si presenta all'utente dopo aver premuto: 

L'ImageView del logo viene ridimensionata e vengono resi visibili 4 TextView, 3 Button e uno Spinner. Tramite questi elementi, l'utente ha la possibilità di cancellare i dati immagazzinati nel database in 3 diversi modi.
Il primo a partire dall'alto, come indica l'Hint della TextView, permette di cancellare i dati di una singola cella di una qualunque griglia disponibile; basterà inserire la coordinata MGRS (processo che sarà semplificato da una feature nella seconda activity) e premere il tasto "*ELIMINA*" che si trova al suo fianco.

In alternativa, l'utente può decidere di eliminare tutti i dati di un'intera griglia, selezionandola tramite lo Spinner e premendo il tasto "*ELIMINA*" affiancato ad esso.
Non è presente alcun meccanismo di "*difesa*" per queste prime due funzionalità, in quanto l'utente, prima di poter eliminare dei dati, dovrà inserire un'informazione o effettuare una scelta.
Come ultima possibilità, l'utente può decidere di cancellare tutti i dati registrati nel database; per evitare cancellazioni non volute, dopo la pressione di "*CANCELLA DATI*", sarà chiesto di confermare ripremendo nuovamente lo stesso pulsante che cambierà testo al suo interno in questo modo:



Possiamo anche notare come, in entrambi i layout, i rispettivi FAB che sono stati premuti per aprire le schermate avranno cambiato icona con quella di una classica "*X*" di chiusura; infatti, per tornare al layout iniziale, bisognerà premere quel FAB con icona "*X*".

Premendo invece il FAB con l'iconica icona delle impostazioni, potremo modificare alcune impostazioni riguardanti la mappa: 

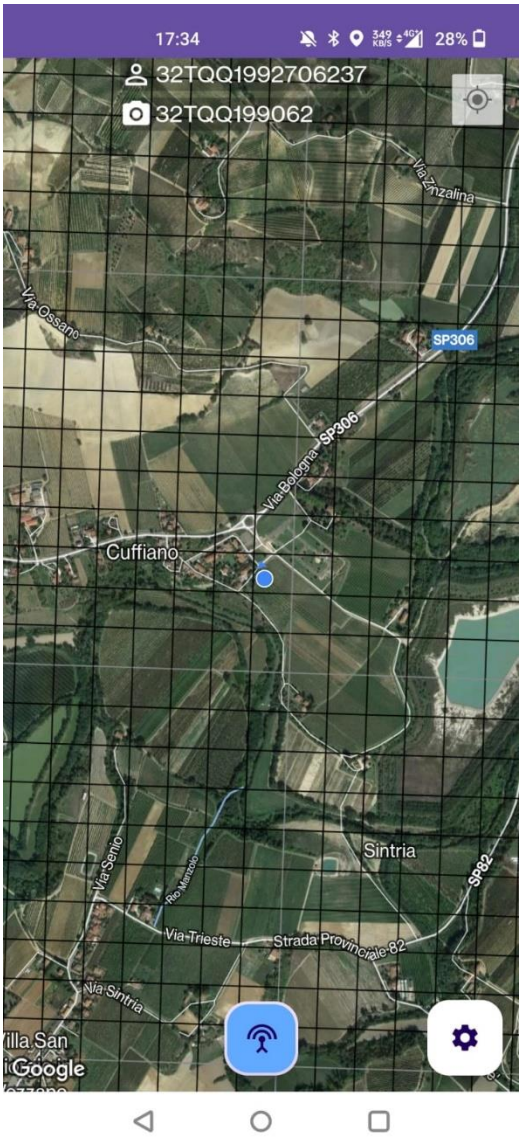
Potremo infatti cambiare il colore della griglia MGRS scegliendo tra varie opzioni, oppure modificare i singoli valori di: numero di misurazioni necessarie per la colorazione di una cella e tempo di attesa (in secondi) tra una misurazione e la successiva.
Questo sarà possibile per ogni tipo di griglia implementata (1 metro, 10 metri, 100 metri, 1 chilometro, 10 chilometri).

Alternativamente, tramite un Switch sarà possibile attivare e disattivare la ricezione di notifiche.

Infine, sono presenti due pulsanti: "*Reset*", che ripristina le impostazioni di default, e "*Salva*", che invece andrà a salvare in modo persistente le scelte di opzioni fatte dall'utente.

Grid and Map Activity

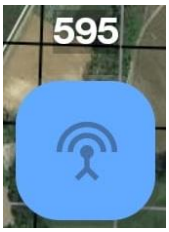
In questa activity risiedono la maggior parte delle funzionalità dell'applicazione; è infatti tramite questa che possiamo: visualizzare la mappa, mostrare la griglia MGRS con grandezza variabile, colorare le celle della griglia dove presenti sufficienti misurazioni e registrare i dati di qualità del segnale LTE, WIFI oppure la quantità di rumore.



L'interfaccia grafica dell'activity si presenta inizialmente in maniera abbastanza spoglia.
Nella parte superiore dell'applicazione abbiamo due Text View con annesse Image View che indicano le coordinate MGRS rispettivamente dell'utente (sempre con precisione massima) e della telecamera, ovvero la coordinata della cella nell'esatto centro dello schermo.
Tramite entrambe queste Text View è possibile copiare i valori testuali con un click prolungato, funzionalità utile per condividere queste informazioni al di fuori dell'applicazione o per inserirle nel campo di input per la cancellazione dei dati di una singola cella nell'activity precedente.

Infine, nella parte nord-est troviamo il pulsante integrato nella mappa di Google per centrare la telecamera sulla nostra posizione.

Nella zona inferiore dell'interfaccia utente troviamo due FAB: uno più vistoso che ho voluto differenziare in quanto è utilizzato per attivare la funzionalità principale dell'applicazione, la rilevazione delle misurazioni.



Dopo la sua pressione, questo pulsante viene disattivato e sopra di esso appare un timer che indica il tempo rimanente fino alla riattivazione.

Tutta la personalizzazione implementata rimanente si cela sotto all'ultimo pulsante visibile, caratterizzato dal noto ingranaggio. Alla pressione di tale pulsante, l'UI muterà mostrando diversi elementi:

Nella parte superiore dello schermo appariranno, nel lato destro, uno Switch, accompagnato da una Image View, che se attivato farà in modo che l'utente si trovi sempre al centro dello schermo sulla mappa e un FAB che ci permetterà di colorare o decolorare le celle della griglia.
Quest'ultimo è stato adeguatamente spaziato per lasciare spazio al pulsante "*integrato*" nella mappa di Google.

Nella parte inferiore invece noteremo un maggiore cambiamento:

Avremo infatti un totale di 5 nuovi FAB e un nuovo pulsante che va a sostituire il FAB di rilevazione delle misure mantenendo il suo stesso canonico colore. Infine, il FAB appena premuto per ottenere questo layout subirà un cambiamento di icona che va ad indicare la sua nuova funzionalità: "*chiudere*" le impostazioni.

I pulsanti apparsi sul lato sinistro dello schermo sono abbastanza intuitivi: servono per fornire all'utente un'alternativa allo zoom manuale con due dita, che spesso può rovinare l'orientamento della mappa e risultare scomodo per alcuni utenti.

Il nuovo bottone centrale, caratterizzato dal colore azzurro, serve per permettere all'utente di selezionare quale tipo di misurazione vuole effettuare e quale tipo di griglia visualizzare in caso di colorazione. Premendolo, infatti, la scritta al suo interno varierà a rotazione tra "*MOBILE*", "*WIFI*", "*AUDIO*"; quella presente a schermo rappresenta la selezione attiva dell'utente.

In conclusione, sulla destra troviamo 3 nuovi FAB: il primo, posizionato in alto a sinistra, permette di modificare la tipologia di griglia visualizzata sulla mappa; il secondo, collocato immediatamente accanto, offre la possibilità di alternare la visualizzazione della mappa tra la modalità satellitare (ibrida) e quella stradale (roadmap); infine, il terzo pulsante, situato in basso a sinistra, serve per attivare o disattivare la visualizzazione della griglia.

Quest'ultima immagine illustra come varia nuovamente la sezione sud-est dell'applicazione dopo la pressione del FAB che ci permette di cambiare la grandezza della griglia.

Possiamo anche notare un esempio di colorazione di una cella della griglia, e la variazione dell'icona del FAB che ci ha permesso di effettuare tale azione.

Un'ulteriore differenza si nota nell'icona dell'ultimo pulsante in basso a destra: non compare più la "*X*" di chiusura, ma è stata sostituita da un'icona che simboleggia l'azione di "*tornare indietro*", che consente di uscire dalla selezione della griglia e di ritornare al layout precedentemente descritto.
La scelta della freccia orientata a destra è stata fatta in quanto l'espansione del menú viene fatta verso la direzione opposta, quindi per quanto possa sembrare contro intuitiva a prima vista penso che sia più adatta alla user experience dell'applicazione.

Dettagli di implementazione

Per lo sviluppo di questo progetto ho scelto di utilizzare Kotlin (1.9.0) come linguaggio di programmazione, questa scelta è stata fatta perché il linguaggio ha una certa somiglianza a Python, a livello di sintassi, con il quale ho abbastanza esperienza e perché dal 2019 Kotlin è diventato il linguaggio ufficiale per la programmazione android nativa. Dunque ho preferito allenarmi con qualcosa di più moderno e diverso dato che ho già accumulato sufficiente esperienza con Java in passato.

Durante lo sviluppo del progetto ho deciso di utilizzare:

- le API di Google Maps (18.1.0) per l'implementazione della mappa e delle funzionalità di localizzazione (21.0.1)
- MGRS Android (2.2.2), sviluppata al National Geospatial-Intelligence Agency (NGA) per gestire la griglia MGRS e le conversioni da latitudine-longitudine a MGRS
- Room (2.3.0) per il database

Metodi di misurazione della connettività e del rumore

La funzionalità di misurazione della connettività e del rumore è la più importante dell'applicazione dato che ogni cosa è costruita attorno ad essa. Per lo sviluppo ho utilizzato componenti che ci vengono forniti da Android: TelephonyManger, WifiManager e AudioRecord.
Ho utilizzato un approccio molto simile per tutti e tre le tipologie di misurazioni: ho definito una funzione che restituisse il valore registrato, una seconda funzione gestisce la valutazione del risultato e il riconoscimento della griglia nella quale veniva effettuata la misurazione.
Successivamente i dati vengono inseriti nel database tramite altre procedure di cui parleró nella sezione "Notifiche e salvataggio dati".

LTE/UMTS

Il primo metodo sviluppato compie semplicemente il compito di cattura del valore rssi riguardante la connettività della rete mobile, questa funzione ha firma `recordCellularSignalStrength(context: Context, onSignalStrengthReceived: (Int) -> Unit)` essa richiede come argomenti un contesto che userà per TelephonyManager, successivamente esegue un controllo, tramite un if statement, per verificare se il dispositivo dell'utente ha una versione di Android maggiore o uguale di Android 12. Questo serve perché le modalità di accesso al dato rssi variano.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
```

La funzione utilizza `TelephonyCallback` e il suo listener `SignalStrengthsListener` per ricevere aggiornamenti sulla forza del segnale utilizzando la funzione `onSignalStrengthsChanged(signalStrength: SignalStrength)` sulla quale è stato eseguito un override.
Quest'ultima filtra le istanze di `cellSignalStrengths` per ottenere la "sezione" dell'LTE e se presente passarlo alla callback, in caso di assenza la funzione ottiene il valore filtrando, in modo analogo a come appena spiegato, alla parte W-CDMA e ottiene il valore dal campo "dbm"; la tecnologia UMTS utilizza l'interfaccia di trasmissione W-CDMA, dunque ove possibile restituisce il valore LTE, altrimenti quello UMTS.

La filtrazione viene effettuata in questo modo:
`signalStrength.cellSignalStrengths.filterIsInstance<CellSignalStrengthLte>().firstOrNull()`
effettuata sostituendo `CellSignalStrengthLte` con `CellSignalStrengthWcdma` per il secondo caso.

Dopodiché, sempre all'interno di `onSignalStrengthsChanged` viene deregistrata la callback usando `telephonyManager.unregisterTelephonyCallback(this)` per fare in modo che non vengano ricevuti altri aggiornamenti successivamente al primo.

La callback viene registrata, subito dopo la definizione di `TelephonyCallback.SignalStrengthsListener` in questo modo:

```
telephonyManager.registerTelephonyCallback(context.mainExecutor, telephonyCallback)
```

```
}else {
```

Se il dispositivo sta eseguendo una versione precedente di Android 12, la funzione utilizza `PhoneStateListener()` e, come per il body del ramo if, viene invocato `onSignalStrengthsChanged`. La logica seguente è analoga al caso precedente.
La differenza sostanziale si nota nei metodi di registrazione e deregistrazione della callback, che in questo caso variano rispettivamente in questo modo:

```
telephonyManager.listen(phoneStateListener, PhoneStateListener.LISTEN_SIGNAL_STRENGTHS)

telephonyManager.listen(this, PhoneStateListener.LISTEN_NONE)
```

```
}
```

Il metodo appena mostrato viene richiamato tramite la funzione `recordDataLTE()` che si occupa di valutare se la misura ottenuta sia idonea `if(it > -120 && it < -20)` ponendo un limite superiore ed inferiore nel quale essa deve rientrare, questi valori sono stati ispirati da questo articolo:

https://opinionitech.com/2021/05/07/rssi-rsrp-rsrq-sinr-cosa-sono-e-come-intepretarli/

Sono state valutate altre fonti, ma questo articolo penso sia il più completo in italiano.
Ho scelto di aggiungere un po' di margine ai valori indicati in rete per fare in modo che anche in caso di alcune imprecisioni la misurazione vada a buon fine.
In caso di misurazione accettabile l'utente viene avvisato tramite Snackbar e viene azionato un thread che si occuperá dell'inserimento della misura nel database tramite una funzione che verrá descritta successivamente.

E' interessante osservare come tramite l'utilizzo di un `when` statement viene riconoscita la griglia nella quale è stata effettuata la misurazione:

```
when(activeTilesSize){

    DbTable.ONE_M -> {
        ...
    }
    DbTable.TEN_M -> {
        ...
    }
    DbTable.HUNDRED_M -> {
        ...
        ...
        ...
```

Wi-Fi

La funzione incaricata di rilevare il valore rssi della connessione WiFi è
`recordWifiSignalStrength(context: Context, callback: SignalStrengthCallback)`
questo metodo utilizza `SignalStrengthCallback` un'interfaccia definita subito prima in questo modo:

```
interface SignalStrengthCallback {
    fun onSignalStrengthReceived(strength: Int)
}
```

Il meccanismo logico utilizzato nella funzione `recordWifiSignalStrength` è molti simile a quello di `recordCellularSignalStrength`.
In primis viene eseguito anche qui un controllo sulla versione Android del dispositivo, questa volta però il controllo suddivide il corpo della funzione tra versione minore o uguale ad Android 11 e maggiore di Android 11.

```
if (Build.VERSION.SDK_INT <= Build.VERSION_CODES.R) {
```

Questa parte della funzione è nettamente più semplice rispetto alla successiva: come prima cosa viene creato un oggetto `WifiManager` tramite il quale potremo accedere alle informazioni sulla connessione tramite il suo elemento `wifiManager.connectionInfo`.
Successivamente viene eseguito un controllo per verificare che il WiFi sia attivo:

```
if (wifiManager.isWifiEnabled) {
    callback.onSignalStrengthReceived(wifiInfo.rssi)
} else {
    callback.onSignalStrengthReceived(-1)
}
```

In caso di risultato positivo viene mandato alla callback che ho definito il valore rssi, altrimenti viene inviato -1 come valore fittizio, e viene avvisato l'utente.
La scelta del valore non è stata completamente casuale, infatti è stato scelto appositamente per creare una situazione analoga ai valori di ritorno che possiamo ottenere anche nel caso di versione maggiore di Android 11.
Infatti successivamente `recordDataWIFI()` utilizzerà un unico controllo sul valore restituito senza dover differenziare per entrambe le versioni come invece siamo obbligati a fare in `recordWifiSignalStrength`.

```
}else {
```

Per i dispositivi che utilizzano Android 12, o versioni successive, viene creato l'oggetto `WifiManager` e controllata la disponibilità del wifi `if (!wifiManager.isWifiEnabled)`.
In caso di risultato negativo, esattamente come per il corpo dell primo if della funzione viene inviato -1 alla callback.
Se il WiFi è abilitato invece viene creato un elemento `ConnectivityManager` tramite il quale viene definita una callback `ConnectivityManager.NetworkCallback()` tramite la quale viene ascoltata qualsiasi variazione nella capacitá di rete.

Viene eseguito l'override di `onCapabilitiesChanged(network: Network, networkCapabilities: NetworkCapabilities)` : viene effettuato un controllo per verificare che la rete abbia il trasporto WiFi, in caso positivo viene restituito il valore `networkCapabilities.signalStrength` alla callback, altrimenti restituiamo -1 e come ultimo passaggio viene deregistrata la callback in questo modo: `connectivityManager.unregisterNetworkCallback(this)` e termina l'override.

Tornando al corpo dell'else, viene creata una richiesta di rete, specificando l'interesse unicamente per la rete WiFi, tramite:

```
NetworkRequest.Builder()
.addTransportType(NetworkCapabilities.TRANSPORT_WIFI)
.build()
```

Per concludere registriamo la callback tramite il ConnectivityManager:
`connectivityManager.registerNetworkCallback(networkRequest, networkCallback)`
Questa callback, come abbiamo visto, verrá poi deregistrata all'interno di `onCapabilitiesChanged`.
}

La funzione appena descritta viene poi richiamata in `recordDataWIFI()` dove viene effettuato un controllo, anticipato poco fa, sul risultato ottenuto.
In caso di risultato -1 l'utente viene avvisato tramite Snackbar dell'assenza del WiFi, altrimenti il procedimento sarà analogo a `recordDataLTE()`.

Rumore

Per la registrazione del rumore ho definito un'interfaccia:

```
interface NoiseLevelCallback {
    fun onNoiseLevelReceived(noiseLevel: Double)
}
```

Che sarà l'unico argomento richiesto dalla funzione che effettivamente esegue la rilevazione del valore ovvero: `captureNoiseLevel(callback: NoiseLevelCallback)`
La prima cosa che succede all'interno di questa funzione è un controllo suoi permessi, viene infatti controllato il permesso `RECORD_AUDIO` e in caso di assenza viene richiamata la funzione incaricata di richiedere il permesso a runtime e viene terminata l'esecuzione della funzione.
L'utente, successivamente allo svolgimento del dialogo per la richiesta dei permessi, potrà immediatamente premere di nuovo il pulsante per la registrazione dato che in questo caso esso non verrá disabilitato.

In caso di permessi concessi la funzione può procedere come segue:
Viene definito un bufferSize con `AudioRecord.getMinBufferSize(rate, channel, format)` con valori rate = 8 KHz, channel = `AudioFormat.CHANNEL_IN_MONO` e format = `AudioFormat.ENCODING_PCM_16BIT`.
Creiamo poi un array con `shortArray(bufferSize)` ed è inizializzata un'istanza di AudioRecord:

```
AudioRecord(
    MediaRecorder.AudioSource.MIC,
    rate,
    channel,
    format,
    bufferSize
)
```

Avviamo poi la registrazione e subito dopo un thread per la valutazione dei dati ottenuti:
Legge i dati audio dal microfono nel buffer utilizzando `recording.read` e poi l'algoritmo esegue questi passaggi:

1. Calcola la somma dei quadrati dei campioni audio per determinare l'energia del segnale.
2. Calcola l'ampiezza come radice quadrata della media dei quadrati dei campioni.
3. Determina l'ampiezza massima possibile per un audio a 16 bit (amplitudeMax), che è $2^{15} - 1$.
4. Calcola il livello di rumore in decibel (dB) utilizzando un logaritmo in base 10 dell'ampiezza normalizzata rispetto all'ampiezza massima.

Questa è stata la formula migliore che sono riuscito a trovare e che non richiedesse calibrazione del microfono.
Il risultato infine viene passato alla callback, all'interno di `recordDataNoise()` viene poi controllato, come per le altre due misurazioni, se il valore ottenuto ricade all'interno di un range prestabilito. In questo caso:

`if(noiseLevel <= 0 && noiseLevel >= -160)` il resto del procedimento è identico alle quasi omonime funzioni mostrate negli scorsi paragrafi.

SISTEMA DI SUDDIVISIONE IN AREE NON INCIDENTI

Griglia

Per la gestione delle aree non incidenti ho deciso di utilizzare, come suggerito nel testo della consegna, la griglia MGRS.
Per l'implementazione ho utilizzato una libreria trovata su GitHub : [MGRS Android].
Le feature utilizzate maggiormente sono state quelle per la creazione della griglia che risiede sopra alla mappa di Google Maps e le funzioni di conversione da latitudine-longitudine a quadrante della griglia.

In primis ho creato un oggetto globale `MGRSTileProvider` nominato activeTiles che sarà l'oggetto che per tutto lo sviluppo ho utilizzato per mostrare la griglia a schermo in questo modo:

```
mMap.addTileOverlay(TileOverlayOptions().tileProvider(activeTiles))
```

Oltre a questo ho creato altri 5 oggetti dello stesso tipo, uno per ogni tipo di griglia che l'utente può utilizzare: 1 M, 10M, 100M, 1KM, 10KM.
In questo modo ho ridotto l'ambiguità del codice avendo sempre a disposizione l'informazione del tipo di griglia attiva per poter sviluppare le funzioni che la utilizzano in maniera più ordinata.
Ho anche creato un oggetto DbTable, una struttura enum definita per riflettere le tabelle del database, che ho utilizzato per lo stesso scopo nelle funzioni di gestione del database.

Conversioni

Per quanto riguarda le conversioni da `LatLng` a `MGRS` ho utilizzato la funzione getMGRS in questo modo: `activeTiles.getMGRS(userLocation)`, richiamando la funzione dal nostro oggetto activeTiles e passando come parametro l'oggetto userLocation creato con il costruttore di LatLng.

Tramite l'oggetto MGRS possiamo dunque usufruire del suo metodo `coordinate()` che è stato definito secondo più firme: senza alcun argomento, passando un intero (accuracy) oppure fornendo un oggetto GridType.
Un oggetto GridType non è altro che un enum definito nella libreria che indica le varie griglie disponibili.
Utilizzare il metodo coordinate senza passare alcun argomento semplicemente restituisce la stringa MGRS con precisione 1 metro, ma considerando che la funzione `getMGRS` con la quale otteniamo l'oggetto MGRS lo crea con precisione massima (1 metro), per la conversione delle coordinate dell'utente (oggetto MGRS) a Stringa ho utilizzato la funzione .toString() di Kotlin.

Quando però ho avuto la necessità di convertire le coordinate dell'utente a stringa ma con diversa precisione, quindi ogni volta che l'utente non utilizzava la griglia da 1M, la scelta della conversione è inevitabilmente ricaduta su `coordinate(GridType)`, in questo modo mi è stato possibile convertire la stringa direttamente nel formato necessario.
Per le coordinate della "camera", cosí chiamata nelle API di Google Maps, ho utilizzato il metodo

```
activeTiles.getCoordinate(cameraPosition.target, cameraPosition.zoom.toInt())
```

che mi permette di ottenere le coordinate relative alla griglia visibile a schermo senza dover eseguire una doppia conversione, prima da LatLng a MGRS e poi da MGRS a String.
Lo zoom indica il livello di precisione della griglia dato che tutte le griglie hanno uno zoom massimo e minimo entro il quale sono mostrate a schermo.

`MGRS.parse(String)` è l'ultimo metodo utilizzato per la conversione, tramite questo possiamo convertire una stringa in un oggetto MGRS, l'ho utilizzato durante la colorazione delle griglie in quanto nel mio database viene salvata una stringa per identificare una cella e non un oggetto MGRS.

Colorazione griglia

Riguardo la colorazione della griglia ho definito due funzioni: una prima che colora una singola cella della mappa e la seconda che invece richiama la prima per ogni cella con misurazioni nel database.
Il funzionamento della prima è abbastanza semplice: `colorSquare(location: String, color: Int)`

- Tramite la lunghezza della stringa riconosce il tipo di griglia da colorare.
- Tramite la funzione `mgrsToLatLngSquare` delimita i 4 angoli della cella.
- Crea il poligono che andrà poi a colorare utilizzando. `runOnUiThread {mMap.addPolygon(polygonOptions)},` questo è necessario perché questa funzione viene richiamata da `colorMap` che viene eseguita su thread diverso dal principale.

La seconda funzione, appena citata, è `colorMap(tileSize: DbTable)` che concettualmente funziona in questo modo:

- Inizializza un thread nel quale in base al tileSize che gli è stato passato come parametro ottiene tutti gli elementi della tabella corrispondente dal database.
- Per ogni elemento, in base al tipo di misurazione "attiva", esegue un controllo inserendo i valori ottenuti in 5 diverse categorie caratterizzate da 5 colori.
- Infine richiama `colorSquare` per ogni elemento passando come argomento la stringa ed il colore.

Il tipo di misurazione "attiva" è determinato semplicemente controllando il testo nel Button che serve per cambiare tale valore per le registrazioni, è sicuramente un dettaglio che ha margine di miglioramento ma con i dovuti controlli non mi ha dato problemi e sono riuscito a risparmiare, anche se poco, lo spazio in memoria di una variabile.

Notifiche e salvataggio dati

Per sviluppare il salvataggio dei dati ho utilizzato Room per quanto riguarda il database, SharedPreferences per le variabili, tra cui anche le impostazioni, per le quali volevo mantenere il loro valore dopo la chiusura dell'app.

Le notifiche invece sono state più complesse da sviluppare, volevo creare un sistema di notifiche che non fosse oppressivo in alcun modo, ho dunque utilizzato due BroadcastReceiver, un JobScheduler, un AlarmManager e anche qui alcuni valori salvati nelle SharedPreferences.

Salvataggio dati

Possiamo suddividere in due categorie ciò che riguarda il salvataggio dei dati all'interno dell'applicazione proposta: database, sviluppato utilizzando Room, e impostazioni, implementate con SharedPreferences.

Room database:

Il database che ho creato è strutturato in una tabella per ogni tipo di griglia, quindi per un totale di 5 tabelle, e ogni tabella al suo interno contiene un elemento per ogni cella identificata da una stringa
`@PrimaryKey val mgrsLocation: String`
gli altri valori salvati nel database sono 3 `Float` che contengono la media delle misure effettuate dentro quella cella (LTE - WiFi - Rumore) e 3 `Int` che indicano invece il numero di misurazioni effettuate.
Per ogni tabella ho poi definito un metodo `INSERT`, due `SELECT` che restituiscono un singolo elemento o

tutti e una `UPDATE`. In aggiunta ho successivamente definito una `DELETE` per un singolo elemento ed una per ogni elemento.

Utilizzando un'interfaccia definita dall'insieme delle Query soprannominate mi è stato possibile leggere e scrivere dati a runtime; un altro elemento peculiare dell'implementazione è stato l'uso del design pattern del *Singleton* che ho utilizzato per la mia classe astratta tramite la quale ho potuto compiere le azioni su database.

SharedPreferences:

Durante lo sviluppo del layout delle impostazioni volevo un sistema in grado di "ricordare" le modifiche applicate ad alcune variabili, ho quindi scelto di utilizzare le SharedPreferences per ottenere questo risultato.
Ho utilizzato questo metodo in quanto mi è sembrato molto comodo e intuitivo, oltre che thread-safe tanto per aggiungere.
Ogni elemento della facciata di impostazioni, nella MainActivity, è stato reso "dinamico" grazie all'uso delle SharedPreferences, infatti nella funzione `onCreate` ogni elemento in quella schermata è inizializzato all'ultimo valore salvato, in caso di assenza del valore richiesto sono stati definiti dei valori di default.
Durante il processo di costruzione dell'Intent che ci permette di spostarci all'Activity della mappa, tramite la funzione `intent.putExtra` ho potuto utilizzare le SharedPreferences per ottenere le impostazioni modificate dall'utente e passarle all'activity successiva, cosí creando un sistema di impostazioni persistenti, anche se piuttosto limitato al momento, che garantisce che l'utente ottenga sempre la sua Activity con le impostazioni che ha scelto.

Notifiche

Le notifiche vengono inviate all'utente quando questo si trova in una zona nella quale non ha effettuato alcuna misurazione, per controllare tale informazione viene richiesta la posizione, qualora fosse concesso il permesso di accedervici in background, con un
`LocationRequest.Builder(5*60*1000).setPriority(Priority.PRIORITY_HIGH_ACCURACY).build()`
nel quale si è impostato un intervallo in millisecondi equivalente a 5 minuti per renderlo poco incisivo sull'uso della batteria del dispositivo.

Il sistema di notifiche è stato sviluppato utilizzando svariati elemente che cooperano tra di loro:

```
BroadcastReceiver() {
```

La scelta di utilizzare due diversi receiver non è strettamente necessaria per il funzionamento del sistema, infatti l'unica differenza tra le due classi che sono state definite risiede nella condizione di azionamento:
`class BootReceiver : BroadcastReceiver()` è utilizzato come rilevatore di boot del telefono, il suo compito è quello di azionare un JobScheduler, di cui parleró a breve, nel momento in cui alcune condizioni sono rispettate. Le condizioni di avviamento dipendono da valori salvati nelle SharedPreferences, semplicemente qualora l'utente avesse abilitato le notifiche dal menú di impostazioni, viene attivato il JobScheduler incaricato poi di inviare la notifica al momento opportuno.
In questo Receiver ho voluto inserire una riga di codice potenzialmente fastidiosa per l'esperienza dell'utente ovvero:
`sharedPref.edit().putBoolean("notiSentAlready", false).apply()`
questo commando è potenzialmente "pericoloso" in quanto va a modificare l'impostazione che impedisce all'applicazione di inviare una notifica all'utente in caso questo sia già stato fatto nella giornata attuale.
Questa scelta è stata fatta tenendo in considerazione che in media gli utenti non riavviano il proprio

dispositivo numerose volte al giorno, oltre a ciò bisogna considerare che nel peggiore dei casi l'utente riceverebbe solamente 1 notifica per riavvio.
Senza contare che l'utente si deve trovare all'interno di una zona nella quale non ha effettuato misurazioni, quindi si suppone non sia una zona abituale dell'utente, ciò ci permette di abbassare ulteriormente il rischio che l'utente venga infastidito da questa scelta implementativa.
Per quanto riguarda l'altro Receiver invece: `class NotificationReset : BroadcastReceiver()`
Questo viene utilizzato per resettare, una volta al giorno, i valori salvati nelle SharedPreferences che memorizzano se la notifica sia stata già inviata nella giornata attuale e se l'AlarmManager sia stato impostato.
L'unica differenza di attivazione tra i due BroadcastReceiver si trova nell'if statement nel quale è contenuto tutto il loro codice, il BootReceiver si aziona solamente se l'intent che lo ha richiamato è del tipo `Intent.ACTION_BOOT_COMPLETED`
mentre l'altro Receiver non ha questo tipo di controllo.

`}`

`AlarmManager {`

Questo elemento è utilizzato per azionare NotificationReset esattamente 1 volta per giorno, per ottenere questo risultato vengono effettuati numerosi controlli sullo stato di alcuni valori delle SharedPreferences:

- `lastAlarmTime`: Viene usato per memorizzare l'orario, in millisecondi, dell'ultimo allarme impostato. Utilizzato poi per verificare che questo valore non abbia superato l'ora attuale, in caso fosse cosí imposterà il successivo elemento dell'elenco a `false`
- `alarmSet`: Non è altro che un valore booleano che indica se l'allarme è impostato o no, in caso questo valore sia `true` la funzione che arma l'allarme termina.
- `timeLastNotification`: Analogo al primo elemento dell'elenco, ma indica il tempo di invio dell'ultima notifica e viene utilizzato per impostare l'allarme al giorno successivo o attuale in base al risultato di `(System.currentTimeMillis() - timeLastNotification < oneDayInMillis)`

Qualora tutti controlli fossero superati viene impostato l'allarme utilizzando la funzione `setInexactRepeating`, scelta al posto di `setExact` per lasciare maggiore libertá al sistema operativo di prioritizzare funzionalità di maggiore importanza del dispositivo.

`}`

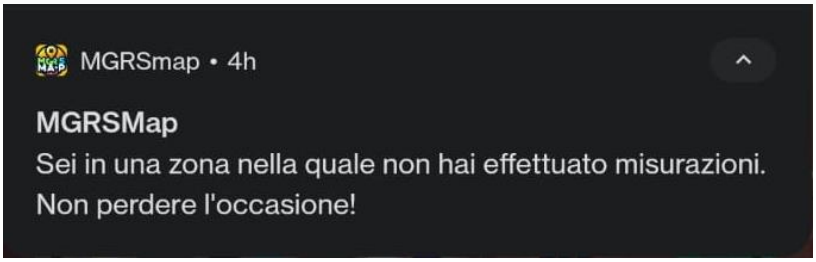
`class LocationCheckJob : JobService() {`

Questo è il componente che viene eseguito dal JobScheduler, viene eseguito in background; azionato nella onDestroy() della MainActivity, oppure dal BootReceiver in entrambi solamente se l'utente ha attivato lo switch delle notifiche.
Il carico di lavoro maggiore è affidato alla funzione `checkLocation(params: JobParameters?)` che controlla periodicamente la posizione dell'utente tramite un oggetto `LocationRequest.Builder(5*60*1000).setPriority(Priority.PRIORITY_HIGH_ACCURACY).build()` quando viene ricevuta una posizione non nulla l'algoritmo valuta se l'utente si trova in una zona nella quale non ha effettuato misurazioni richiamando `isUserInZone(location: Location): Boolean` , questa funzione si limita a converitre l'oggetto Location in stringa MGRS con precisione (tipo di griglia) 1KM e controllare se presente nel database.
In caso l'utente si trovi dunque in una zona assente di dati viene composta una notifica tramite la funzione `triggerNotification()` che, come illustrato a lezione, crea un oggetto `NotificationManager` tramite il quale sarà effettivamente inviata la notifica; esegue un controllo per verificare l'esistenza del NotificationChannel e in caso non esista lo crea e compone l'intent che associerá alla notifica per fare in modo che se cliccata l'applicazione venga aperta.

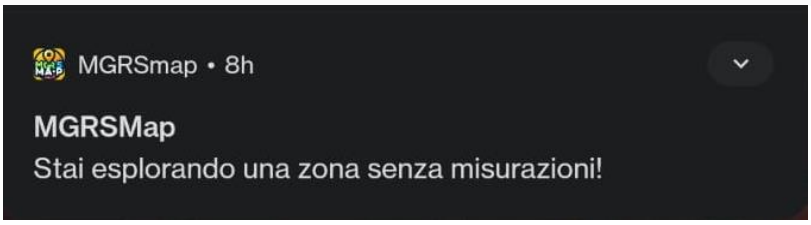
In questo ultimo passaggio è possibile migliorare l'user experience facendo in modo che l'intent sia creato per l'Activity della mappa, mentre al momento è creato in modo che l'applicazione si apra alla MainActivity in quanto volevo fare in modo che un utente che preme la notifica con l'intenzione di "spegnere" lo switch delle notifiche avesse questa possibilitá a "distanza" di un solo click esattamente come l'utente che invece vuole aprire l'Activity con la mappa.

```
}
```

La notifica si presenta in questo modo:



Notifica estesa



Notifica non estesa

Conclusione

In questa relazione, ho mirato a fornire un quadro chiaro e conciso delle tecniche implementate nel progetto, ho organizzato i contenuti seguendo la struttura della consegna, assicurando una corrispondenza diretta tra i requisiti richiesti e le soluzioni adottate.
In conclusione, benchè alcuni dettagli implementativi siano stati omessi per brevità, spero che la trattazione offra una visione d'insieme che non solo documenti il lavoro svolto ma che possa anche servire, al sottoscritto, come solida base per futuri sviluppi e miglioramenti.