UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED
ELETTRICA E MATEMATICA APPLICATA

Mobile Robots For Critical Missions - Project Work

Nome	Matricola	Email
Marco Milone	0622701992	m.milone15@studenti.unisa.it
Fabrizio Sarno	0622702052	f.sarno14@studenti.unisa.it
Beniamino Squitieri	0622702021	b.squitieri@studenti.unisa.it
Francesca Venditti	0622701969	f.venditti1@studenti.unisa.it

L.M. INGEGNERIA INFORMATICA - A.A. 2023/2024

Indice

1 L'obiettivo

Il progetto mira a sviluppare un sistema di navigazione autonoma per il robot TurtleBot4, consentendo a quest'ultimo di muoversi all'interno di una mappa nota e di seguire le indicazioni dei segnali stradali posti ad ogni intersezione identificati tramite dei codici QR ed evitando ostacoli sia statici che dinamici durante la navigazione.
Le varie fasi di progettazione prevedono:

- Definizione dell'idea realizzativa.
- Progettazione dell'architettura del sistema.
- Rilevamento dei segnali stradali.
- Implementazione.

Il robot utilizza una telecamera per rilevare e decodificare i codici posizionati lungo il percorso. Ognuno rappresenta un segnale stradale specifico:

- `STRAIGHTON`: indica di proseguire dritto
- `GOBACK`: indica di eseguire un'inversione di marcia
- `RIGHT`: indica di eseguire una svolta a destra
- `LEFT`: indica di eseguire una svolta a sinistra
- `STOP`: indica di fermare la marcia

Una volta identificato un QR Code, il TurtleBot4 deve seguire le istruzioni associate al segnale.

Dal punto di vista tecnico, il progetto utilizza l'ambiente di sviluppo ROS2 (Robot Operating System) per l'implementazione della logica di navigazione e controllo. Per l'elaborazione delle immagini e la decodifica dei codici QR viene impiegata la libreria `OpenCV` e la classe `BarcodeReader` di `DynamSoft` [1], mentre gli algoritmi di path planning e obstacle avoidance sono integrati con i pacchetti di navigazione di ROS2. L'hardware principalmente utilizzato consiste in quello montato sul robot stesso.

La validazione del sistema avviene in un ambiente di test controllato, dove vengono posizionati diversi segnali stradali e ostacoli. Durante i test, il TurtleBot4 viene osservato per assicurarsi che riconosca correttamente i segnali QR, segua le istruzioni contenute al loro interno ed eviti gli ostacoli in modo efficiente. Questo progetto dimostra le capacità avanzate di navigazione e percezione del TurtleBot4, rendendolo un candidato ideale per applicazioni in ambienti dinamici e complessi.

2 Gli strumenti

Di seguito viene presentata una piccola descrizione degli strumenti utilizzati:

- **ROS2**: è un insieme di software e strumenti open-source per lo sviluppo di applicazioni robotiche. Fornisce una serie di librerie per la comunicazione tra nodi tramite il paradigma publisher-subscriber, in cui i nodi possono pubblicare o sottoscriversi a specifici topic.
- **TurtleBot4**: un robot equipaggiato con una serie di sensori, tra cui un LIDAR per la mappatura e la navigazione, sensori di distanza e un IMU per la stima dell'orientamento e della posizione. Include una telecamera ad alta risoluzione, utile per aumentare le performace di algoritmi di visione artificiale e riconoscimento di oggetti. Infine, la base mobile offre una capacità di carico di 9 kg ed è in grado di viaggiare ininterrottamente a velocità fino a 0,306 m/s quando il sistema di sicurezza è attivato e fino a 0,46 m/s in modalità priorità.
- **Cattura e decodifica dei QR Code**: Per l'operazione di rilevamento e decodifica dei codici QR è stato utilizzata la libreria `BarcodeReader` dell'azienda `Dynamsoft`. Tale strumento permette di decodificare codici a barre lineari (1D) e codici a barre 2D, di cui fanno parte proprio i codici QR. Inoltre, è resistente a situazioni particolari come codici deformati, inclinati, danneggiati o poco visibili a causa di illuminazione o basso contrasto.

3 L'idea

Per adempiere al task richiesto l'idea è stata quella di modellare una struttura che permettesse di approcciare ogni incrocio nella maniera desiderata. In particolare, facendo riferimento all'immagine sottostante, che è una schematizzazione della mappa dell'ambiente di navigazione, per ogni incrocio sono stati definiti alcuni punti chiave. Essi rappresentano il punto di stop del TurtleBot4 dopo aver eseguito una navigazione da un incrocio ad un altro. Ogni incrocio può presentare diversi punti chiave, questo è legato alla possibilità del robot di poter navigare in direzioni differenti. Considerando quindi i quattro punti cardinali, (NORD - SUD - EST - OVEST), ogni incrocio può avere diversi punti di approccio. Questi sono stati poi nominati con la nomenclatura (UP - DOWN - DX - SX) in modo da evitare confusioni durante l'implementazione e l'analisi dei risultati.
Ad esempio, Figura 1, si può osservare la presenza di tre punti chiave legati alla possibilità del TurtleBot4 di approcciare l'incrocio navigando verso NORD a cui corrisponde il punto DOWN come quello d'arresto. L'assenza del punto SX è legata al fatto che il robot non può arrivare nell'incrocio navigando verso OVEST perchè la mappa non lo consente.

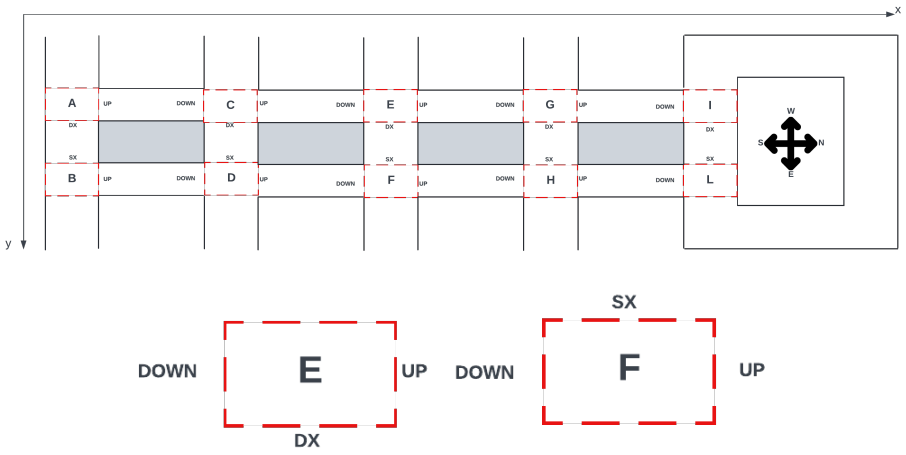

Figure 1: Incroci e punti chiave

La totalità degli incroci è stata modellata come un grafo orientato che collega ogni punto con i suoi vicini con archi in entrambe le direzioni. Siccome ogni possibile indicazione letta dal segnale porta ad un incrocio differente a seconda dell'orientamento cardinale verso cui è orientato il robot durante la navigazione, tale grafo deve avere quattro versioni, una per ogni direzione cardinale.

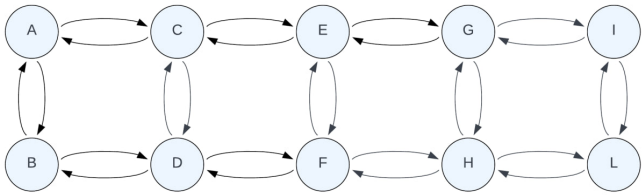

Figure 2: Grafo degli incroci

I grafi sono rappresentati qui di seguito tramite forma matriciale in cui è possibile osservare per ogni orientamento cardinale del robot, e per ogni punto del grafo, quale indicazione rappresenta un arco disponibile e verso quale punto ci si può dirigere. L'ultima riga della tabella rappresenta l'orientamento che il robot deve assumere una volta raggiunta la destinazione.
Le possibili combinazioni per il robot orientato verso NORD sono riportate nella prima tabella.

NORTH	**Left**	**Right**	**GoBack**	**StraightOn**
A	None	B	None	C
B	A	None	None	D
C	None	D	A	E
D	C	None	B	F
E	None	F	C	G
F	E	None	D	H
G	None	H	E	I
H	G	None	F	L
I	None	L	G	None
L	I	None	H	None
Next Orientation	WEST	EAST	SOUTH	NORTH

Table 1: Direzioni possibili con orientamento NORTH

Quando invece l'orientameno del robot è OVEST, si fa affidamento alle seguenti possibili direzioni di movimento.

WEST	**Left**	**Right**	**GoBack**	**StraightOn**
A	None	C	B	None
B	None	D	None	A
C	A	E	D	None
D	B	F	None	C
E	C	G	F	None
F	D	H	None	E
G	E	I	H	None
H	F	L	None	G
I	G	None	L	None
L	H	None	None	I
Next Orientation	SOUTH	NORTH	EAST	WEST

Table 2: Direzioni possibili con orientamento WEST

Per il TurtleBot 4 orientato verso SUD, si hanno le seguenti combinazioni.

SOUTH	**Left**	**Right**	**GoBack**	**StraightOn**
A	B	None	C	None
B	None	A	D	None
C	D	None	E	A
D	None	C	F	B
E	F	None	G	C
F	None	E	H	D
G	H	None	I	E
H	None	G	L	F
I	L	None	None	G
L	None	I	None	H
Next Orientation	EAST	WEST	NORTH	SOUTH

Table 3: Direzioni possibili con orientamento SOUTH

Infine, nel caso in cui il robot sia rivolto a EST, le direzioni in cui questo può muoversi sono contenute nella tabella sottostante.

EAST	**Left**	**Right**	**GoBack**	**StraightOn**
A	C	None	None	B
B	D	None	A	None
C	E	A	None	D
D	F	B	C	None
E	G	C	None	F
F	H	D	E	None
G	I	E	None	H
H	L	F	G	None
I	None	G	None	L
L	None	H	I	None
Next Orientation	NORTH	SOUTH	WEST	EAST

Table 4: Direzioni possibili con orientamento EAST

Questa modellazione permette di pianificare facilmente ogni azione da eseguire una volta raggiunto um nodo della mappa. Ad esempio, partendo dal punto A con orientamento NORTH e ricevendo il comando STRAIGHTON, ci si dirigerà verso il nodo C con orientamento finale NORTH. Allo stesso modo, partendo dal punto C con orientamento NORTH e ricevendo il comando RIGHT, ci si dirigerà al nodo D con prossimo orientamento EAST.

4 La discovery

Terminata l'operazione di navigazione, resta da gestire l'individuazione del codice QR all'interno dell'area dell'incrocio. A tal proposito, è stato posto il seguente problema: una volta che il TurtleBot4 ha raggiunto l'incrocio designato, la camera potrebbe non avere un campo visivo sufficiente per individuare il segnale poiché potrebbe essere posizionato in punti estremi dell'area (estrema destra o estrema sinistra). Per ovviare a ciò, è stata ideata un'operazione che prende il nome di *discovery*: questa consiste nel far ruotare il robot di quarantacinque gradi prima a sinistra e poi a destra in modo da essere certi che il codice QR possa essere correttamente letto.
Per evitare situazioni anomale in cui il QR non rientri completamente nel campo visivo della telecamera, la rotazione non avviene in una singola operazione ma è divisa in due intervalli. Alla fine di questa operazione, il robot ritorna nella posizione nella quale era prima di cominciare la discovery.

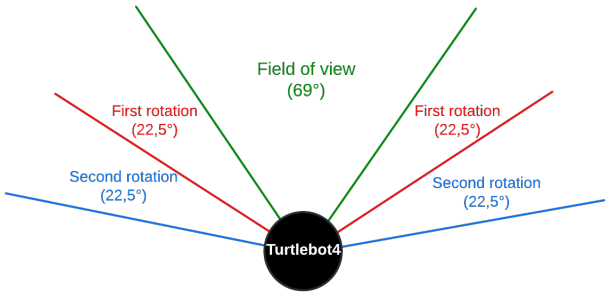

Figure 3: Operazione discovery del TurtleBot4

5 L'architettura

Questa architettura illustra un tipico approccio modulare nei sistemi ROS2, dove le diverse funzionalità sono suddivise in nodi distinti, ognuno responsabile di un compito specifico. La comunicazione tra questi nodi è gestita tramite topic, garantendo una chiara separazione dei compiti e facilitando il debug e la manutenibilità. Il `camera_node` gestisce gli input visivi, mentre il `navigation_node` elabora questi input per prendere decisioni di movimento. Infine, il `discovery_node` esegue l'esplorazione basandosi sui comandi di navigazione.

1. **camera_node**: Questo nodo cattura ed elabora le immagini provenienti della fotocamera. In particolare riceve le immagini dal topic `image_raw`, le analizza alla ricerca di un codice QR e pubblica il risultato sul topic `command`.

2. **navigation_node**: Questo nodo gestisce la logica di navigazione del robot, decidendo dove esso debba muoversi in relazione ai comandi ricevuti, per far ciò si sottoscrive al topic `command`. Inoltre si sottoscrive al topic `kidnap_status` per rendere possibile il riposizionamento del robot nel punto precedente. Infine pubblica sul topic `discovery` per avviare l'operazione di ricerca.

3. **discovery_node**: Questo nodo gestisce la rotazione del robot quando non viene letto alcun segnale. Si sottoscrive al topic `discovery` per avviare l'operazione di discovery quando il `navigation_node` lo richiede. Avviata la discovery pubblica i comandi di velocità sul topic `cmd_vel` per effettuare le rotazioni.

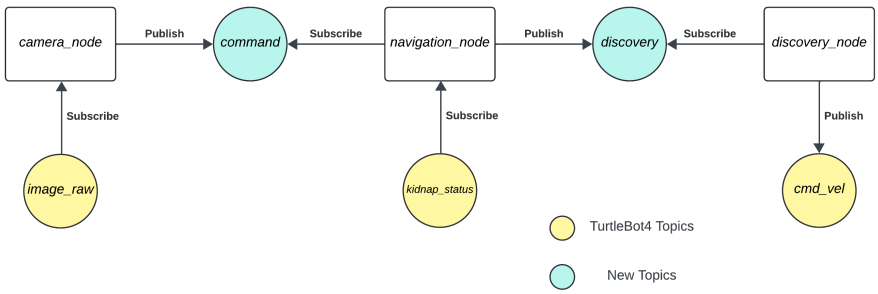

Figure 4: Architettura del sistema

6 L'implementazione

6.1 Cattura ed elaborazione delle immagini

Di seguito viene presentato il codice implementativo per il rilevamento e la decodifica dei codici QR presenti lungo il percorso seguito dal robot.
Le operazioni vengono gestite dal nodo `QRCodeReader` che sfrutta diversi attributi per gestire la logica della decodifica dei comandi e i tempi di rilevamento di questi ultimi.
Alla creazione del nodo avviene l'inizializzazione e le successive configurazioni delle istanze di `CVBridge` e `BarCodeReader`. In particolar modo viene impostata la licenza per il QR detector e vengono create le sottoscrizioni e i publisher per i topic necessari.

6.1.1 Decodifica del QR Code

La funzione `decodeframe` esamina un codice QR fornito. A seconda dell'esito della decodifica restituisce la stringa di testo associata al codice, tale stringa è `None` se non è presente nessun QR oppure uno dei possibili segnali elencati in precedenza. La funzione utilizza il metodo `decode_buffer`, generalmente utilizzato per la decodifica di flussi video.

```
1 def decodeframe(self, frame):
2     try:
3         text_results = self.reader.decode_buffer(frame)
4         if text_results:
5             return text_results[0].barcode_text
6     except BarcodeReaderError as bre:
7         print(bre)
```

6.1.2 Pubblicazione del segnale

La funzione `publish_command` si occupa di pubblicare i comandi decodificati dal codice QR. Vengono indicate come "comandi" le stringhe testuali decodificate dai QR code, le quali rappresentano effettivamente un comando da impartire al TurtleBot4 per aggiornare il percorso da seguire durante la navigazione.
In questa funzione viene, dunque, effettuato un controllo affinché lo stesso comando non venga pubblicato ripetutamente in un intervallo di tempo relativamente breve. Ciò viene garantito tramite l'utilizzo di un'opportuna coda, la quale viene aggiornata nel caso in cui venga rilevato un segnale diverso. L'aggiornamento di questa struttura avviene anche quando viene rilevato un segnale uguale, tuttavia a patto che siano trascorsi almeno quindici secondi dal rilevamento del precedente. Per fare ciò, ogni qual volta viene inserito un comando all'interno della coda, viene salvato anche l'istante temporale in cui l'operazione è stata effettuata, implementando così una sorta di meccanismo di *debounce*. Questa strategia permette di evitare la pubblicazione di comandi duplicati sul topic `/command` i quali potrebbero causare navigazioni errate del TurtleBot4.

```
1  def publish_command(self, command):
2      msg = String()
3      msg.data = command
4      self.detection_time = time.time()
5
6      if not self.command_queue: # queue is empty, save the detected
           command
7          self.command_queue.append(command)
8          self.saving_time = self.detection_time
9          self.get_logger().info("Publishing signal for the first
               time...")
10         self.publisher.publish(msg)
11     else:
12         if command != self.command_queue[-1]: # new signal is
               different from the previous one
13             self.command_queue.clear() # replace the previous
                   signal with the current one
14             self.command_queue.append(command)
15             self.saving_time = self.detection_time # update time of
                    signal save
16             self.get_logger().info("Publishing new signal...")
17             self.publisher.publish(msg)
18         else:
19             if (self.detection_time - self.saving_time) >= self.
                   expiration_time: # new signal is equal to the
                   previous one
20                 self.saving_time = self.detection_time # just
                       update time of signal save if the new signal
                       has been detected after the time threshold
21                 self.get_logger().info("Publishing same signal...")
22                 self.publisher.publish(msg)
```

6.1.3 Funzione di callback

La funzione `camera_callback` viene attivata ogni volta che viene rilevata una nuova immagine dalla telecamera. Questa viene convertita in un'immagine compatibile con `OpenCV`, dopodiché la funzione prova a decodificare il codice QR presente al suo interno per infine pubblicare la stringa di comando ottenuta. Se non è presente alcun codice viene pubblicato il valore `NOSIGNAL` utilizzato per l'avvio dell'operazione di discovery.

```
def camera_callback(self, data):
    try:
        # Convert the ROS Image message to an OpenCV image
        cv_image = self.bridge.imgmsg_to_cv2(data, desired_encoding
            ='bgr8')
    except CvBridgeError as e:
        self.get_logger().error(f"Could not convert image: {e}")
        return

    # Decode the QR code from the image
    command = self.decodeframe(cv_image)
    if command is None:
        command = "NOSIGNAL"
        self.publish_command(command)
    else:
        self.publish_command(command)
```

6.2 Operazione di discovery

A seguire, sono riportati le funzioni e gli accorgimenti principali per la corretta implementazione del comportamento di discovery.

6.2.1 Nodo `DiscoveryNode`

Il primo passo è stato quello di inizializzare un nuovo nodo chiamato `DiscoveryNode` con i rispettivi log e timer. A questo si aggiunge anche la creazione di un publisher sul topic `/cmd_vel` e la sottoscrizione a `/discovery` da cui leggere il comando per far partire o meno la rotazione del robot.

6.2.2 Funzione di callback per la discovery

La funzione `discovery_callback` viene attivata quando viene ricevuto un messaggio sul topic `/discovery` citato prima. Se il messaggio contiene la stringa `START_DISCOVERY`, e il processo non è già in esecuzione, avvia in un nuovo thread l'operazione di ricerca del segnale. Ciò è necessario per gestire la discovery contemporaniamente al timer utilizzato per gli input di velocità. Se non si utilizzasse un altro thread, il nodo resterebbe bloccato nell'aggiornamento del timer.

```
def discovery_callback(self, msg):
    if msg.data == "START_DISCOVERY" and self.discovery_flag ==
        False:
        threading.Thread(target=self.discovery).start() # Start
            discovery in a new thread
```

6.2.3 Funzione di callback per il movimento

La parte responsabile del movimento del robot è gestita dalla callback sottostante. Questa viene richiamata periodicamente dal timer e pubblica il comando di movimento corrente se il sistema è in uno stato operativo.

```
def move_callback(self):
    if self.ok:
        self.cmd_pub.publish(self.msg)
```

6.2.4 Funzioni di movimento e stop

La funzione `rotate` è responsabile della rotazione del robot. Il verso di rotazione e l'angolo possono essere specificati durante la chiamata nel codice e sono espressi rispettivamente come un valore booleano (`True` se orario, `False` se antiorario) e in gradi. Vengono poi pubblicati comandi di velocità angolare sul topic specifico del robot affinché questo possa attuare la rotazione.
La funzione `stop` ferma invece il movimento del TurtleBot4 impostando la sua velocità lineare e angolare a zero.

```
def rotate(self, angle_degrees, clockwise=True):
    angle_radians = math.radians(angle_degrees)
    velocity = 1.57 # rad/s
    factor = -1 if clockwise else 1
    msg = Twist()
    msg.angular.z = velocity * factor
    self.msg = msg
    sleep_time = abs(angle_radians / velocity)

    self.ok = True

    self.sleep(sleep_time)

    self.ok = False

def stop(self):
    msg = Twist()
```

```
18      msg.linear.x = 0.0 # m/s
19      msg.angular.z = 0.0
20      self.msg = msg
21      self.ok = True
22      self.sleep(0.5)
23      self.ok = False
```

6.2.5 Funzione di discovery

La funzione `discovery` implementa una routine che vede il robot ruotare intorno all'asse z a sinistra e destra. Nello specifico, la rotazione verso sinistra - così come quella a destra - viene eseguita in due intervalli con angoli uguali in modo da inquadrare il segnale nel caso in cui questo fosse in una posizione tale che non possa essere rilevato se la rotazione fosse eseguita in un solo intervallo. Tra una rotazione e un'altra, il robot resta in attesa in modo da stabilizzare la telecamera e permettere così il corretto rilevamento del codice QR. Alla fine del processo, l'intera routine viene resettata.

```
1  def discovery(self):
2      self.discovery_flag = True
3      self.get_logger().info("Discovery starting...")
4      self.sleep(1)
5
6      self.rotate(22.5, clockwise = False )
7      self.stop()
8      self.sleep(1.5)
9      self.rotate(22.5, clockwise = False )
10     self.stop()
11     self.sleep(1.5)
12
13     self.rotate(67.5, clockwise = True)
14     self.stop()
15     self.sleep(1.5)
16     self.rotate(22.5, clockwise = True)
17     self.stop()
18     self.sleep(1.5)
19
20     self.rotate(45, clockwise = False )
21     self.stop()
22     self.sleep(0.5)
23     self.stop()
24     self.get_logger().info("Discovery ended.")
25     self.discovery_flag = False
```

6.3 Gestione del grafo

Questa sezione contiene il codice per la creazione e gestione del grafo utilizzato dal robot per navigare nel suo ambiente. Viene definita la classe `Graph` che rappresenta la struttura di un grafo con nodi e connessioni, inizializzata da un file YAML. Questa classe è fondamentale per la navigazione, poiché gestisce la struttura del grafo e le coordinate dei punti necessari per dirigere il robot nel suo ambiente.
Nello script seguente viene caricata la struttura del grafo con i rispettivi punti, le cui coordinate sono descritte nei rispettivi file YAML.

```
    def __init__(self, graph_file_path, points_file_path):
        self.graph = self.load_yaml(graph_file_path)
        self.points = self.load_yaml(points_file_path)

    def load_yaml(self, file_path):
        with open(file_path, 'r') as file:
            return yaml.safe_load(file)
```

6.3.1 Ricerca del punto più vicino

La funzione `find_nearest_point` calcola la distanza euclidea tra il punto in esame e tutti gli altri presenti nel grafo in modo da trovare quello più vicino. Questa funzione consente quindi di individuare il nodo del grafo in cui il robot si trova attualmente.

```
    def find_nearest_point(self, input_point):
        min_distance = float('inf')
        nearest_point = None

        for main_key, sub_points in self.points.items():
            for sub_key, value in sub_points.items():
                distance = math.sqrt((input_point['x'] - value['x']
                    ]) ** 2 + (input_point['y'] - value['y']) ** 2)
                if distance < min_distance:
                    min_distance = distance
                    nearest_point = main_key

        return nearest_point
```

6.3.2 Calcolo del prossimo nodo e orientamento

Questa funzione recupera il prossimo nodo e l'orientamento basandosi sui valori del nodo corrente. Utilizza la struttura del grafo per determinare il prossimo nodo e l'orientamento successivo. Se non trova il percorso, restituisce la coppia di valori `(None, None)`.

```
def get_next_node_and_orientation(self, current_orientation,
    direction, current_node):
    try:
        next_node = self.graph[current_orientation][direction][
            current_node]
        next_orientation = self.graph[current_orientation][
            direction]['NextOrientation']
        landmark = self.orientation_landmark_conversion(
            next_orientation)
        return self.points[next_node][landmark], next_orientation
    except KeyError as e:
        return None, None
```

6.3.3 Recupero delle coordinate

La funzione `get_coordinates` recupera le coordinate di un nodo specificato. Se il nodo non viene trovato, restituisce la coppia di valori `(None, None)`.

```
def get_coordinates(self, node):
    try:
        return node['x'], node['y']
    except KeyError:
        return None, None
```

6.3.4 Conversione delle stringhe di orientamento

La funzione converte le stringhe di orientamento in stringhe rappresentanti i punti chiave di ogni incrocio del grafo. Restituisce la stringa del punto di riferimento corrispondente all'orientamento fornito.

```
def orientation_landmark_conversion(self, orientation):
    if orientation == "NORTH":
        return "DOWN"
    elif orientation == "EAST":
        return "SX"
    elif orientation == "SOUTH":
        return "UP"
    elif orientation == "WEST":
        return "DX"
```

6.4 Navigazione all'interno della mappa

La navigazione del TurtleBot4 è gestita dagli script seguenti, dove viene configurato un nodo in grado di ricevere i comandi correlati alla direzione da seguire durante la navigazione e gestire lo scenario di "robot rapito", ovvero quando quest'ultimo viene sollevato dal suolo per compiere operazioni di riposizionamento all'intereno del percorso. Per rendere la gestione del robot più semplice ed intuitiva, al lancio del file `python`, verrà chiesto all'utente di inserire, da terminale, le coordinate iniziali del robot insieme all'orientamento iniziale.

6.4.1 Inizializzazione del nodo

Alla base del funzionamento del nodo c'è la classe `NavigationNode`, che estende la classe `Node` di ROS2. Questa classe gestisce tutta la logica di navigazione del robot.
Nella fase di inizializzazione vengono effettuate le seguenti azioni:

- Definizione del `Quality of Service` per il topic `/kidnap_status` per rendere compatibili tra i diversi nodi i messaggi inviati e ricevuti, impostando le relative policy di affidabilità e mantenimento della cronologia dei messaggi.
- Creazione di un'istanza del grafo per gestire la navigazione tramite i punti caratterizzanti quest'ultimo.
- Impostazione dello stato iniziale (punto di partenza, orientamento e direzione iniziale del robot). In aggiunta vengono memorizzati i valori precedenti in caso di rapimento.
- Sottoscrizione ai topic `/command` e `/kidnap_status`, rispettivamente per ricevere i comandi di direzione e quelli relativi allo stato di rapimento del robot.
- Creazione di un publisher per il topic `/discovery` per inviare i rispettivi messaggi di rotazione.
- Chiamata alla funzione di `perform_navigation` per avviare la navigazione.

```
    def __init__(self, graph_file_path, points_file_path,
        starting_point, orientation=None):
        super().__init__('navigation_node')

        # Kidnap topic QoS definition
        qos_profile = QoSProfile(
            reliability=ReliabilityPolicy.BEST_EFFORT,
            history=HistoryPolicy.KEEP_LAST,
            depth=1
        )

        self.graph = Graph(graph_file_path, points_file_path)
        self.navigator = TurtleBot4Navigator()

        self.current_node = starting_point
        self.current_orientation = orientation
        self.current_direction = "STRAIGHTON"

        self.previous_node = None
        self.previous_orientation = None
        self.previous_direction = None
        self.kidnap_status = False

        # Set initial position
        self.set_initial_pose(self.current_node, self.
            current_orientation)

        # Subscribe to the command topic
        self.command_subscription = self.create_subscription(
            String,
            'command',
            self.direction_callback,
            1
        )

        # Subscribe to the kidnap status topic
        self.kidnap_subscription = self.create_subscription(
            KidnapStatus,
            'kidnap_status',
            self.kidnap_callback,
            qos_profile
        )

        # Discovery publisher
        self.publisher = self.create_publisher(String, 'discovery',
             1)

        # Perform navigation given a starting position, orientation
             and direction
        self.perform_navigation(self.current_node, self.
            current_orientation, self.current_direction)
```

6.4.2 Funzione di callback per la direzione

Questa funzione viene chiamata quando viene ricevuto un messaggio sul topic `/command`. Se questo è `NOSIGNAL`, pubblica un messaggio per avviare la discovery. Se il messaggio ricevuto è `STOP`, arresta il nodo. Se altrimenti viene ricevuta una direzione valida (`LEFT, RIGHT, STRAIGHTON, GOBACK`), aggiorna quella corrente e chiama la funzione `perform_navigation`.

```
    def direction_callback(self, msg):
        discovery_msg = String()
        discovery_msg.data = "START_DISCOVERY"

        if msg.data == "NOSIGNAL":
            self.publisher.publish(discovery_msg)
        else:
            if msg.data == "STOP":
                self.get_logger().info("Stop command received.
                    Shutting down node.")
                rclpy.shutdown()
            elif msg.data == "LEFT" or msg.data == "RIGHT" or msg.
                data == "STRAIGHTON" or msg.data == "GOBACK":
                self.current_direction = msg.data
                self.perform_navigation(self.current_node, self.
                    current_orientation, self.current_direction)
```

6.4.3 Funzione di callback per il rapimento

La `kidnap_callback` viene chiamata quando viene ricevuto un messaggio sul topic `kidnap_status`. Se il robot viene rapito, la funzione imposta un flag booleano a `True`, ciò è necessario per individuare il momento in cui il robot viene riposizionato a terra dopo essere stato sollevato. Quando il robot viene rilasciato, la funzione ripristina la posizione e l'orientamento precedenti e reimposta la posizione iniziale del robot.

```

    def kidnap_callback(self, msg):
        # Access the is_kidnapped field from the KidnapStatus
            message
        is_kidnapped = msg.is_kidnapped

        if is_kidnapped:
            self.get_logger().info("robot has been kidnapped.")
            self.kidnap_status = True

        if not is_kidnapped and self.kidnap_status:
            self.kidnap_status = False
            self.get_logger().info("robot has been released.")
            # Reset the current position to the previous one
            self.current_node = self.previous_node
            self.current_orientation = self.previous_orientation
            self.current_direction = self.previous_direction

            self.previous_node = None
            self.previous_orientation = None
```

```
            self.previous_direction = None

            # Set the initial position after the robot has been kidnapped
            self.set_initial_pose(self.current_node, self.current_orientation)
            self.perform_navigation(self.current_node, self.current_orientation, self.current_direction)
```

6.4.4 Esecuzione della navigazione

La funzione esegue la navigazione verso il prossimo nodo. Trova il nodo più vicino al punto corrente, determina il prossimo nodo e l'orientamento in base alla direzione corrente, infine avvia la navigazione verso quel punto. Durante questo processo, viene aggiornato anche lo stato corrente del robot.

```
    def perform_navigation(self, current_point, orientation, direction):
        # Find the nearest node to the starting point
        nearest_node = self.graph.find_nearest_point(current_point)
        self.get_logger().info(f"Starting from node: {nearest_node}")

        # Find the next node and the next orientation
        next_node, next_orientation = self.graph.get_next_node_and_orientation(
        orientation, direction, nearest_node
        )
        self.get_logger().info(f"Navigating to: {next_node}")
        self.get_logger().info(f"Whit direction: {direction}")

        # Navigate to the next node
        self.navigate_to_node(next_node, self.orientation_conversion(next_orientation))

        # Update the current state
        self.current_node = next_node
        self.current_orientation = next_orientation
        self.previous_node = current_point
        self.previous_orientation = orientation
        self.previous_direction = direction
```

6.4.5 Set della posizione iniziale

Questa funzione imposta la posizione iniziale del robot utilizzando il punto di partenza e l'orientamento forniti.

```
    def set_initial_pose(self, start_point, orientation):
        orientation = self.orientation_conversion(orientation)
        initial_pose = self.navigator.getPoseStamped([start_point['
            x'], start_point['y']], orientation)
        self.navigator.setInitialPose(initial_pose)
```

6.4.6 Navigazione verso un nodo di destinazione

Qui si dirige il robot verso il nodo specificato, ottenendo le coordinate di quest'ultimo dal grafo e utilizzando il modulo di navigazione per avviare il movimento.

```

    def navigate_to_node(self, node, orientation):
        x, y = self.graph.get_coordinates(node)
        if x is not None and y is not None:
            goal_pose = self.navigator.getPoseStamped([x, y],
                orientation)
            self.navigator.startToPose(goal_pose)
            time.sleep(1.5)
        else:
            self.get_logger().error("Invalid node coordinates")
```

6.4.7 Conversione dell'orientamento

La funzione `orientation_conversion` converte un orientamento espresso come stringa (`NORTH, EAST, SOUTH, WEST`) nel valore corrispondente al tipo `TurtleBot4Directions`.

```
    def orientation_conversion(self, orientation):
        if orientation == "NORTH":
            return TurtleBot4Directions.NORTH
        elif orientation == "EAST":
            return TurtleBot4Directions.EAST
        elif orientation == "SOUTH":
            return TurtleBot4Directions.SOUTH
        elif orientation == "WEST":
            return TurtleBot4Directions.WEST
```

Riferimenti

[1] Dynamsoft™, *Robust Barcode Scanner SDK with Flexible APIs*, 2015 (First Release).

Elenco delle figure

Elenco delle tabelle