UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Intelligent Systems

and

Robotics Laboratory

Progetto:

Esplorazione e risoluzione del labirinto con il Robot Freenove4WD

Marco Pinori 274319

Stefano Fattore 259869

xx/yy/2023

Indice generale

1 Obiettivo del progetto

L'obiettivo del progetto consiste nel far esplorare al robot a quattro ruote FreeNove 4WD un labirinto la cui struttura non è nota al robot, in modo autonomo e intelligente. L'algoritmo che abbiamo ideato ha qualche caratteristica in comune con l'algoritmo di ricerca in profondità DFS. Il robot inizia la sua esplorazione da un punto iniziale, chiamato radice o stato iniziale e, in base ai valori dei sensori di cui è dotato, l'algoritmo elabora i dati e decide la direzione e verso in cui si deve muovere il robot. Se il robot trova un vicolo cieco dal quale non è più possibile proseguire in avanti, l'algoritmo deciderà di far ripercorrere al robot il percorso intrapreso nel verso opposto fino a quando non trova un nuovo percorso da esplorare. L'esplorazione si conclude quando il robot trova il punto finale del labirinto.

Nella presente documentazione, Agente, Robot e macchina assumono lo stesso significato.

2 Software e linguaggi utilizzati

Il progetto è stato realizzato utilizzando i seguenti software/servizi e linguaggi di programmazione.

Software/servizi:

- IDE: Pycharm e Spyder
- Simulatore di robotica 3D: CoppeliaSim
- Sistema di controllo di versione: GitHub

Linguaggio di programmazione:

- Python

3 Vincoli di progetto

3.1 Labirinto

Il labirinto deve avere una struttura tridimensionale che deve rispettare i seguenti vincoli:

1. Struttura statica e non può mutare nel tempo.

2. Un solo punto iniziale (radice/root).
3. Un solo punto finale (uscita, final).
4. Può essere rappresentato attraverso un albero semplice:
    4.1. Non ci sono cicli.
    4.2. Non è possibile passare per uno stesso punto se si esplora in profondità.
5. Presenza di vicoli ciechi per aumentare la complessità dell'esplorazione.
6. Percorso formato da muri che formano angoli di 90° e 180°.
7. Esiste un solo percorso più breve che permette all'agente di andare dallo stato iniziale allo stato finale.
8. Nella robotica virtuale:
    8.1. In prossimità dell'uscita del labirinto, è presente un "gate" costituito da sensori di prossimità che consentono di rilevare la presenza o meno del robot. Se il sensore rileva il robot vuol dire che quest'ultimo è riuscito a trovare l'uscita e l'algoritmo di esplorazione termina.
    8.2. Distanza tra due muri deve essere almeno di 0.45 m.
    8.3. Spessore deve essere almeno di 0.10 m.

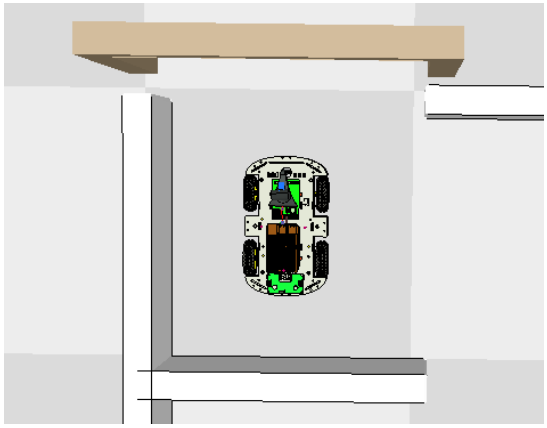


Figura 1: Gate

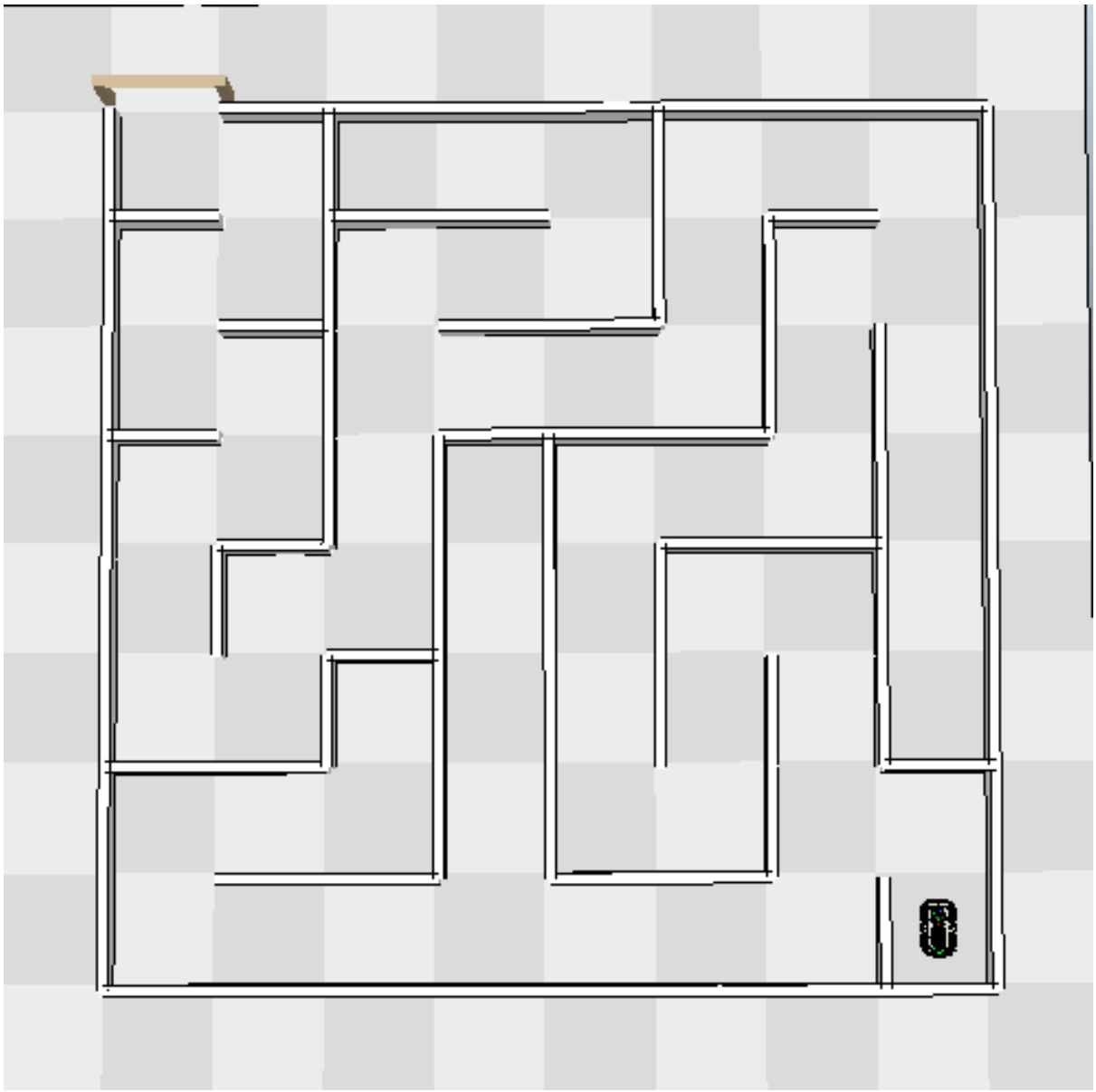


Figura 2: Maze_1

3.2 Robot

Il robot è il modello in 3D del FreeNove 4WD.

I vincoli del robot sono:

1. Labirinto sconosciuto all'agente e quindi non può sapere a priori il percorso più breve dall'ingresso fino all'uscita (detto anche punto finale).
2. Uscita sconosciuta.
3. L'agente tiene traccia delle decisioni che ha eseguito e lo fa senza la nozione di tempo.
4. Al fine di prendere una decisione informata riguardo a quale azione eseguire, il robot deve essere in grado di osservare ciò che è attorno a lui. Ciò viene fatto utilizzando diversi sensori.
5. Altri vincoli del robot sono definiti successivamente nella descrizione dello stato INITIAL.

Esso è costituito dai seguenti sensori di prossimità:

- Front: posizionato nella parte frontale del Robot (fps in Coppelia).
- Left: posizionato nella parte di sinistra del robot e forma un angolo di 90° con il sensore Front (lps in Coppelia).
- Right: posizionato nella parte di destra del robot e forma un angolo di 90° con il sensore Front (rps in Coppelia).

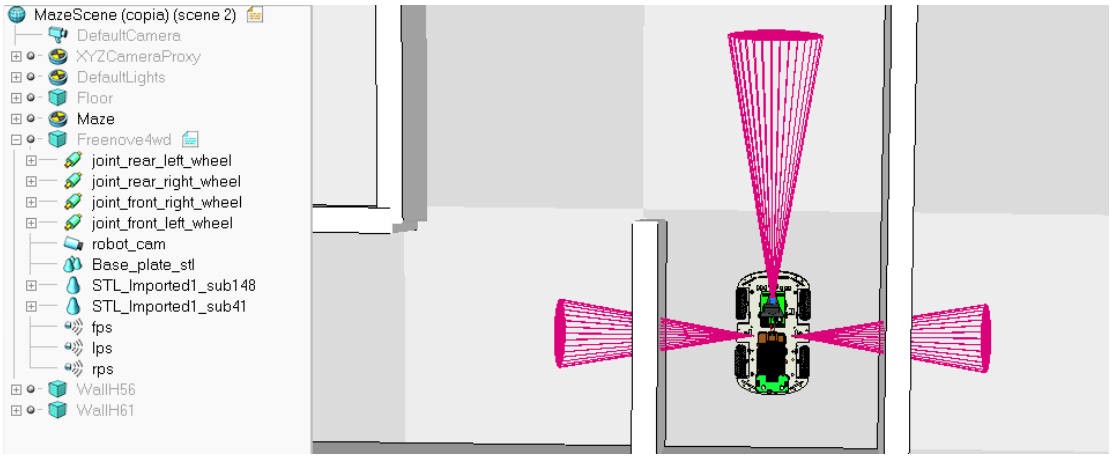


Figura 3: Modello del robot e sensori

Il robot, ottiene i dati di ciascun sensore e controlla se si trova ad una distanza minima di sicurezza per poter continuare l'esplorazione, evitando urti contro il muro. In caso contrario, se i sensori rilevano che il robot è troppo vicino al muro, il robot effettua il bilanciamento se l'opzione è attivata (come spiegato avanti), per allontanarsi dai muri in tutta sicurezza e posizionarsi in una distanza ragionevole.

4 Algoritmo

Il Controller contiene l'algoritmo principale per prendere decisioni e dare comandi al robot. Tutto ciò avviene solo dopo che sono stati analizzati ed elaborati i valori dei sensori di prossimità, lo stato (State), la posizione della macchina (Position), la modalità di esplorazione (Mode) e l'albero (Tree) del labirinto. Le decisioni si traducono in comandi che devono essere eseguiti dal physical_body. L'algoritmo si basa sull'architettura SENSE, THINK, ACT che verrà analizzata in modo più dettagliato successivamente.

Adesso andremo a spiegare quali valori possono assumere State, Position, Mode capendo l'importanza che hanno per il corretto funzionamento dell'algoritmo.

State, Position, Mode sono classi di tipo Enum.

4.1 State, Position e Mode

4.1.1 State

Lo stato *State* del robot può assumere diversi valori come:

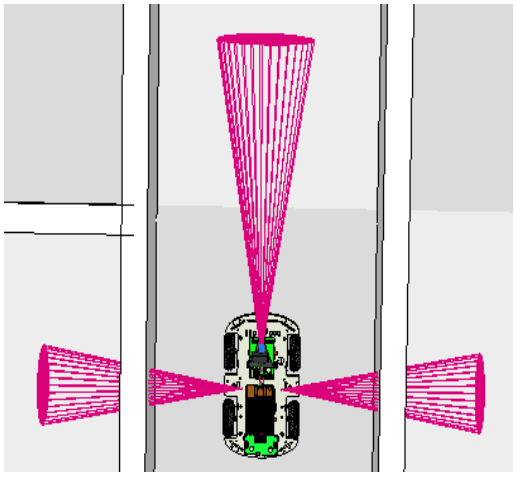
- STARTING = -1
- STOPPED = 0
- RUNNING = 1
- ROTATING = 2
- SENSING = 3

STARTING: Primo stato che assume il robot, nella fase di inizializzazione. Non può tornare più in questo stato durante l'esplorazione.

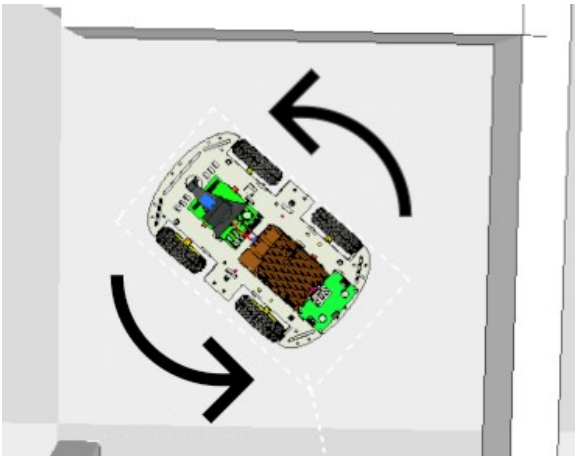
STOPPED: Il robot ha i motori spenti ed è fermo.

RUNNING: Robot in corsa durante il quale analizza continuamente, tramite i sensori, lo spazio circostante.

ROTATING: Stato in cui il Robot sta ruotando su sé stesso (asse z).



RUNNING



ROTATING

SENSING: Stato in cui il Robot è fermo e analizza, tramite i sensori, lo spazio circostante. Stato cruciale in cui avvengono importanti operazioni tra le quali è presente l'aggiornamento dell'albero del labirinto.

4.1.2 Position

La posizione *Position* della macchina non è la posizione data dalle coordinate, ma assume un altro significato. Può avere valori come:

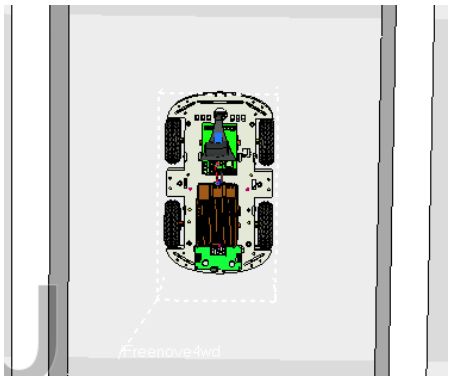
- UNKNWON = 0
- INITIAL = 1
- CORRIDOR = 2
- JUNCTION = 3

UNKNWON: Posizione sconosciuta al robot. Il robot deve analizzare lo spazio circostante tramite i sensori in modo da potersi posizionare nella posizione corretta. Dato che i sensori del robot sono Front, Left e Right è necessario

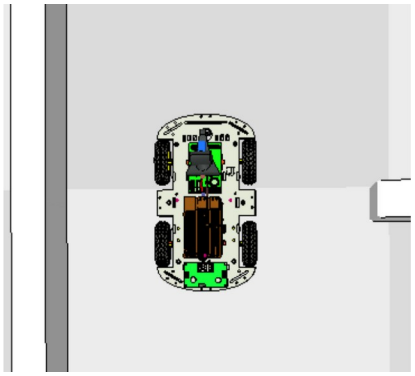
ruotare il robot per capire se dietro di esso c'è un muro o meno. Una volta fatte le dovute rotazioni, la parte posteriore deve essere contro il muro e il valore di Position diventa INITIAL.

INITIAL: Posizione iniziale, fase di inizializzazione. Il robot deve assumere una posizione che deve rispettare i seguenti vincoli:

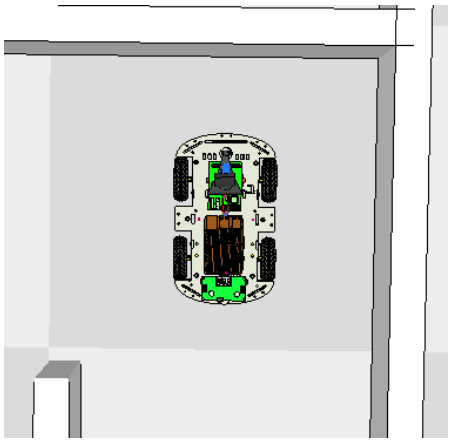
- La parte posteriore del robot deve essere contro il muro.
- Il robot non può essere posizionato in una giunzione/intersezione con 4 direzioni/strade disponibili.
- Il robot non può essere posizionato in mezzo al corridoio dove è possibile andare in due sensi (avanti e indietro) .



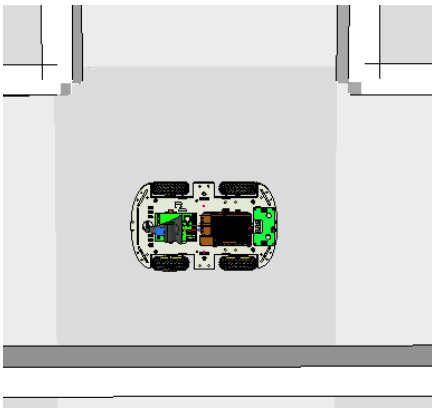
CORRIDOR: solo in avanti e indietro



CORRIDOR: muro a destra come corridoio



JUNCTION: 2 direzioni disponibili



JUNCTION: 3 direzioni disponibili

CORRIDOR: Il Robot si trova nel corridoio. Definizione: *spazio ristretto dove la macchina può andare avanti e indietro ma non può svoltare né a sinistra né a destra*.

JUNCTION: Il Robot si trova in una giunzione. Definizione: *spazio dove la macchina può andare a destra e/o a sinistra oltre alla possibilità di andare avanti e/o indietro. La condizione necessaria e sufficiente è che può andare a sinistra e/o a destra.*

4.1.3 Mode

La modalità *Mode* di esplorazione:

- EXPLORING = 0
- ESCAPING = 1

EXPLORING: Modalità di default. La macchina si trova in questa modalità quando deve esplorare spazi non conosciuti del labirinto. Durante questa fase, l'algoritmo crea in modo progressivo dei nodi che appenderà all'albero in modo sistematico. Tale albero mappa il labirinto in base a determinate condizioni che saranno spiegate dettagliatamente più avanti.

ESCAPING: Modalità che viene attivata quando la macchina trova un vicolo cieco. L'algoritmo deciderà di far ripercorrere al robot il percorso fatto fino ad ora fino a quando non trova un nuovo percorso da esplorare e, in tal caso, verrà riattivata la modalità EXPLORING.

4.2 I comandi

Una volta che il Controller ha analizzato lo stato effettivo in cui si trova il robot, lo spazio circostante ed elaborato dati, prende decisioni su quali movimenti la macchina deve svolgere. Il Controller quindi invia al Physical_body i comandi e azioni che poi si traducono in movimenti fisici della macchina. Una nota importante è che ogni comando Command è affiancato da un'azione di tipo Compass e il solo comando non è sufficiente a far svolgere un movimento al robot. Infatti il Controller invia al physical_body la seguente coppia:

```
[command, action]
```

Es:

```
[<Command.ROTATE: 2>, <Compass.WEST: 180.0>]
```

La classe *Command* è costituita dai seguenti comandi:

- START = - 1
- STOP = 0
- RUN = 1
- ROTATE = 2
- GO_TO_JUNCTION = 3
- BALANCE = 4

START: Primo comando inviato all'accensione della macchina e quindi alla prima esecuzione dello script in python.

STOP: Utilizzato nei seguenti casi per fermare la corsa della macchina:

- Vicinanza di un ostacolo nella direzione frontale del robot.
- Robot posizionato in una giunzione.

RUN: Comando per far muovere il robot in avanti.

ROTATE: Usato per far ruotare il robot su sé stesso.

GO_TO_JUNCTION: Usato per far posizionare il robot al centro di una giunzione.

BALANCE: Quando una parte laterale (destra o sinistra) del robot si trova troppo vicino ad un muro, il Controller invia questo comando per far posizionare il robot ad una distanza minima di sicurezza dal muro per evitare scontri non voluti. È possibile attivare e disattivare la modalità di bilanciamento semplicemente modificando il file *config.conf.*

Abbiamo realizzato la classe Compass per la definizione dei 4 punti cardinali e per alcune funzioni utili per la conversione di stringhe con il tipo Compass e viceversa.

Compass:

- NORTH = 90.0
- SOUTH = - 90.0
- EAST = 0.0
- WEST = 180.0

Tutti i comandi e le azioni eseguite vengono memorizzati in una specifica lista di liste chiamata **`performed_com_actions.`** Inoltre, viene memorizzata anche la traiettoria del robot nella lista **`trajectory`** costituita dalle azioni di tipo Compass eseguite dalla macchina.

4.3 Sense - Think - Act

L'algoritmo per l'esplorazione del labirinto si basa sull'architettura Sense – Think – Act.

```
def algorithm(self):
    """ Algorithm used to explore and solve the maze """

    if self.goal_reached():
        return True

    # SENSE
    self.read_sensors()
    self.left_values.append(self.left_value)
    self.front_values.append(self.front_value)
    self.right_values.append(self.right_value)

    # THINK
    actions, com_actions = self.control_policy()
    com_action = self.decision_making_policy(com_actions)
    command, action = com_action

    # Update of the tree
    self.update_tree(actions, action)

    # ACT
    performed = self.do_action(com_action)

    return False
```

Nel SENSE avviene la lettura dei valori dei sensori di prossimità: *Front*, *Left*, *Right*.

Nel THINK ci sono principalmente due funzioni:

- **`control_policy()`**: in base ai valori dei sensori, lo stato effettivo del robot e l'albero del labirinto restituisce due liste:
  - **`actions`**: lista di azioni di tipo **`Compass`** (ovvero bussola) ed è usata solo per aggiornare l'albero del labirinto.
  - **`com_actions`**: abbreviazione di commands_actions. È una lista di liste di due elementi: uno di tipo **`Command`** e l'altro di tipo **`Compass`**. In altre parole, contiene una lista di coppie `[command, action]` e solo una di queste viene realmente eseguita dal robot.
- **`decision_making_policy(com_actions)`**: data la lista **`com_actions`** precedentemente descritta, decide la coppia `[command, action]` che deve essere eseguita dal robot. La scelta viene effettuata considerando la lista di priorità **`priority_list`** che contiene in ordine di priorità i punti cardinali North, South, East, West. L'ordine della lista può essere modificato aggiornando il file *config.conf.*

Nell'ACT viene semplicemente dato l'ordine al Physical_body di eseguire l'azione precedentemente scelta dalla Decision Making Policy (DMP) chiamando il metodo **`do_action(com_action)`**.

4.4 Imparare dalle esperienze passate: l'albero

Il Controller, prima di prendere decisioni sulla prossima azione da fare, deve analizzare le informazioni su ciò che è accaduto nel passato, quelle del presente e lo stato attuale della macchina. I dati e le informazioni del passato sono memorizzate in un albero che contiene nodi. Ogni nodo viene generato quando il robot si trova in parti specifiche del labirinto e, in particolare, il nodo radice *root* viene creato all'avvio della macchina che si trova nella posizione iniziale.

4.4.1 Il nodo

Ogni nodo ha diverse proprietà:

- Name: nome del nodo ottenuto dalla concatenazione del primo carattere della stringa memorizzata in Type e un indice numerico generato in modo sequenziale.
- Action: azione di tipo Compass associata al nodo.
- Type: tipo di nodo.
  - OBSERVED: Nodo aggiunto che è stato osservato ma che non è stato ancora esplorato. In pratica il robot trova una direzione libera, genera il nodo in questa direzione e assegna tale attributo al nodo poiché ha deciso di andare in un altra direzione.
  - EXPLORED:
    - Nodo visitato per la prima volta.
    - Può avere figli di tipo DEAD_END ma almeno uno di essi deve essere EXPLORED.
  - DEAD_END:
    - Nodo foglia che non ha figli, oppure
    - Nodo che ha tutti figli DEAD_END.
  - FINAL
    - Ultimo nodo alla fine del labirinto. Indica l'uscita.
- Parent: nodo genitore
- Left: figlio sinistro
- Mid: figlio centrale
- Right: figlio destro

I nodi vengono generati nelle seguenti posizioni:

- Posizione iniziale del robot: nodo *root*

- In una giunzione: nodo intermedio
- Punto finale di un vicolo cieco: nodo *dead end*
- Punto finale del labirinto: nodo *final*

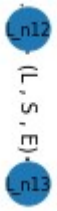
e ciascuno è associato ad una terna di caratteri:

- Il primo può essere: L (Left), M (Mid), R (Right).
- Il secondo: N (North), S (South), E (East), W (West).
- Il terzo: O (Observed), E (Explored), D (Dead end), F (Final).

Nella rappresentazione grafica dell'albero il nodo figlio è collegato al padre da una freccia unidirezionale che parte da quest'ultimo e finisce sul figlio. La freccia è etichettata dalla suddetta terna che mostra i 3 attributi del nodo figlio.

Esempio:

- Nodo padre: L_n12
- Nodo figlio: L_n13
- Terna: (L,S,E)→ (**L**eft, **S**outh, **E**xplored)



4.4.2 Aggiornamento dell'albero

Ad ogni ciclo dell'algoritmo principale viene chiamato il metodo:

```
update_tree(actions, action_chosen)
```

ed aggiorna l'albero soltanto quando il robot è nella modalità SENSING (*Mode.SENSING*) attenendosi alle azioni disponibili ottenute dalla Control Policy e all'azione scelta dalla DMP. Nelle altre modalità (*Mode*) del robot i nodi dell'albero non vengono aggiunti o modificati.

L'albero viene esportato in un dizionario utilizzando il metodo

```
build_tree_dict()
```

che chiama la ricerca DFS ricorsiva opportunamente modificata per memorizzare correttamente le proprietà di ciascun nodo. Una volta ottenuto il dizionario dell'albero, questo viene memorizzato nel file *data_analysis.conf* e può essere rappresentato graficamente in un secondo momento eseguendo lo script DrawTree.py che, automaticamente, genera l'immagine con estensione png dell'albero nella directory *MetalPhoenix/other/*. Questo script utilizza opportune librerie Python per il plotting di grafi come: networkx e matplotlib.pyplot.

Nell'esempio che segue, il robot ha risolto il labirinto Maze_1 trovando il percorso per raggiungere l'uscita ma non prima di trovare vicoli ciechi da cui uscire. Il robot si è basato sui seguenti input dati dall'utente modificando opportunamente il file *config.conf*:

```
priority_list = SOUTH, WEST, EAST, NORTH
#  Intelligence. Available options: [low, mid, high (not developed yet)]
intelligence = low
```

Dalla lista di priorità si può dedurre che il robot preferisce andare a SUD (SOUTH) dato che ha priorità più alta e, se non fosse possibile, proverà ad andare a OVEST (WEST) e poi preferirà andare a EST (EAST) prima di NORD (NORTH).

Una volta esplorato il labirinto e dopo aver raggiunto l'uscita l'albero che si ottiene eseguendo lo script DrawTree.py è il seguente:

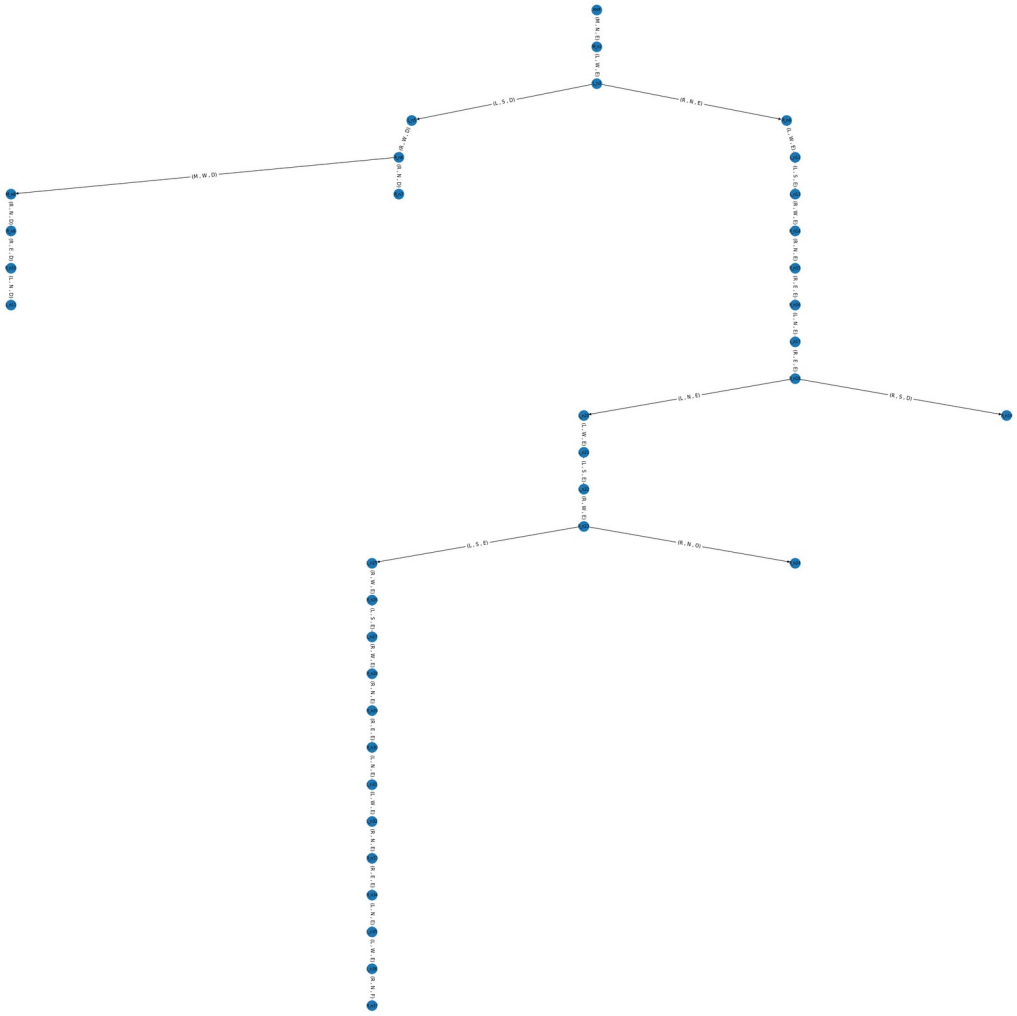


Figura 4: Albero del labirinto Maze_1

Da notare che l'albero di un labirinto non è unico e dipende da come il robot esplora il labirinto; più precisamente, dipende dalla lista di priorità che gli viene assegnata e dall'intelligenza. Per una maggiore chiarezza, abbiamo realizzato la seguente immagine che contiene sia l'albero che il labirinto. Prestare particolare attenzione su cosa rappresentano i nodi dell'albero, le loro caratteristiche e le posizioni in cui vengono generati.

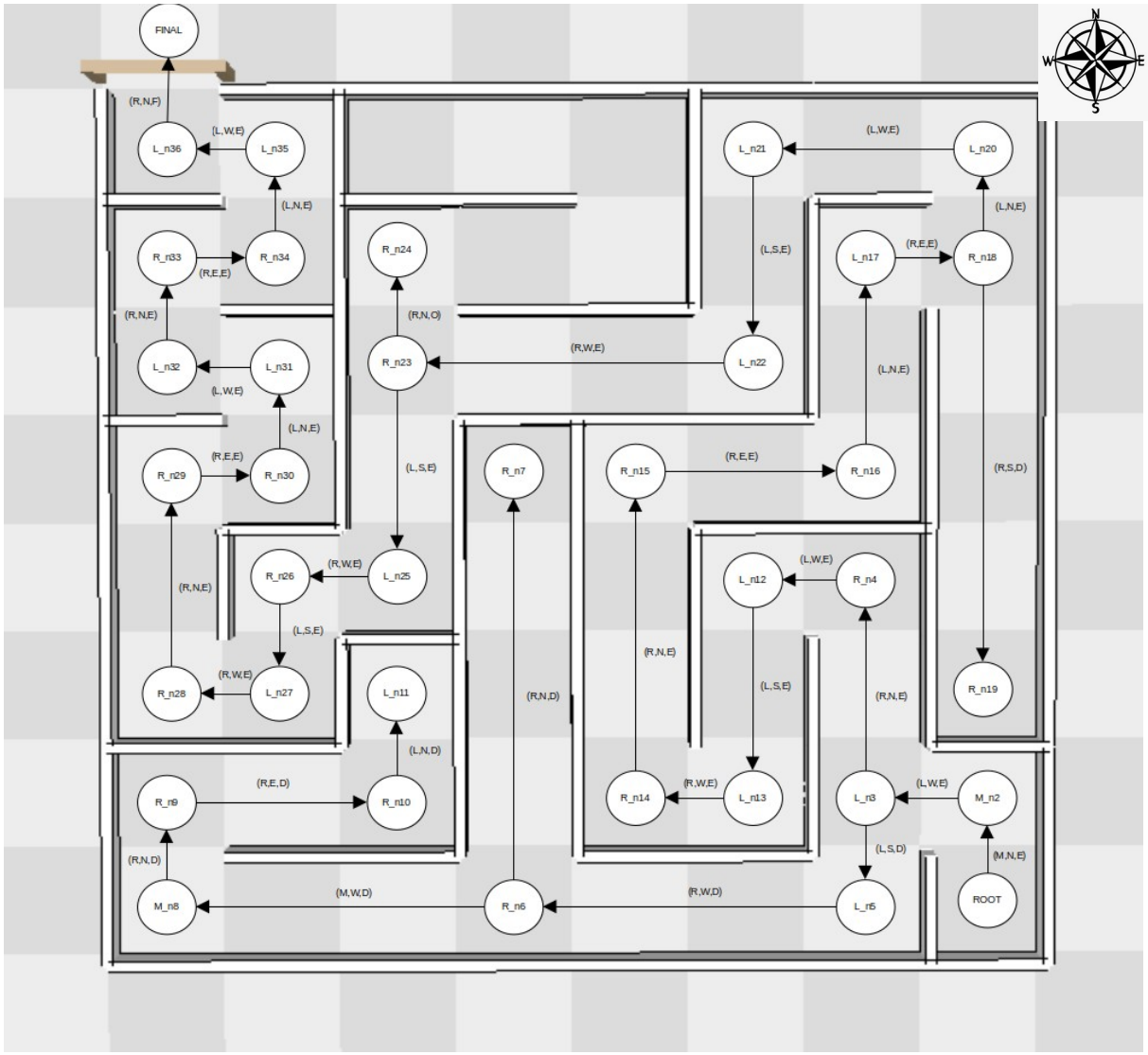


Figura 5: Labirinto (Maze_1) + Albero dei nodi

Dal nodo ROOT il robot, in modalità EXPLORING e stato SENSING, rileva che può andare solo nella direzione frontale; genera e appende il nodo M_n2 come figlio Mid al nodo ROOT. Sceglie di proseguire avanti nella direzione NORD (NORTH) fino ad incontrare il muro, fermarsi e analizzare lo spazio circostante (SENSING). Il robot apprende che c'è solo una strada alla sua sinistra e genera il nodo L_n3 come figlio sinistro (Left) di M_n2. Sceglie quindi di andare a sinistra, nella direzione OVEST (WEST), e ruota di 90° in senso antiorario. Una volta che ha ruotato, prosegue a OVEST (WEST) raggiungendo la giunzione rappresentata dal nodo L_n3. Adesso genera due nodi: R_n4 e L_n5 come figlio destro (Right) e

sinistro (Left) di L_n3 rispettivamente. Dovrà prendere una decisione tra due opzioni:

- Andare a NORD (NORTH), nodo R_n4.
- Andare a SUD (SOUTH), nodo L_n5.

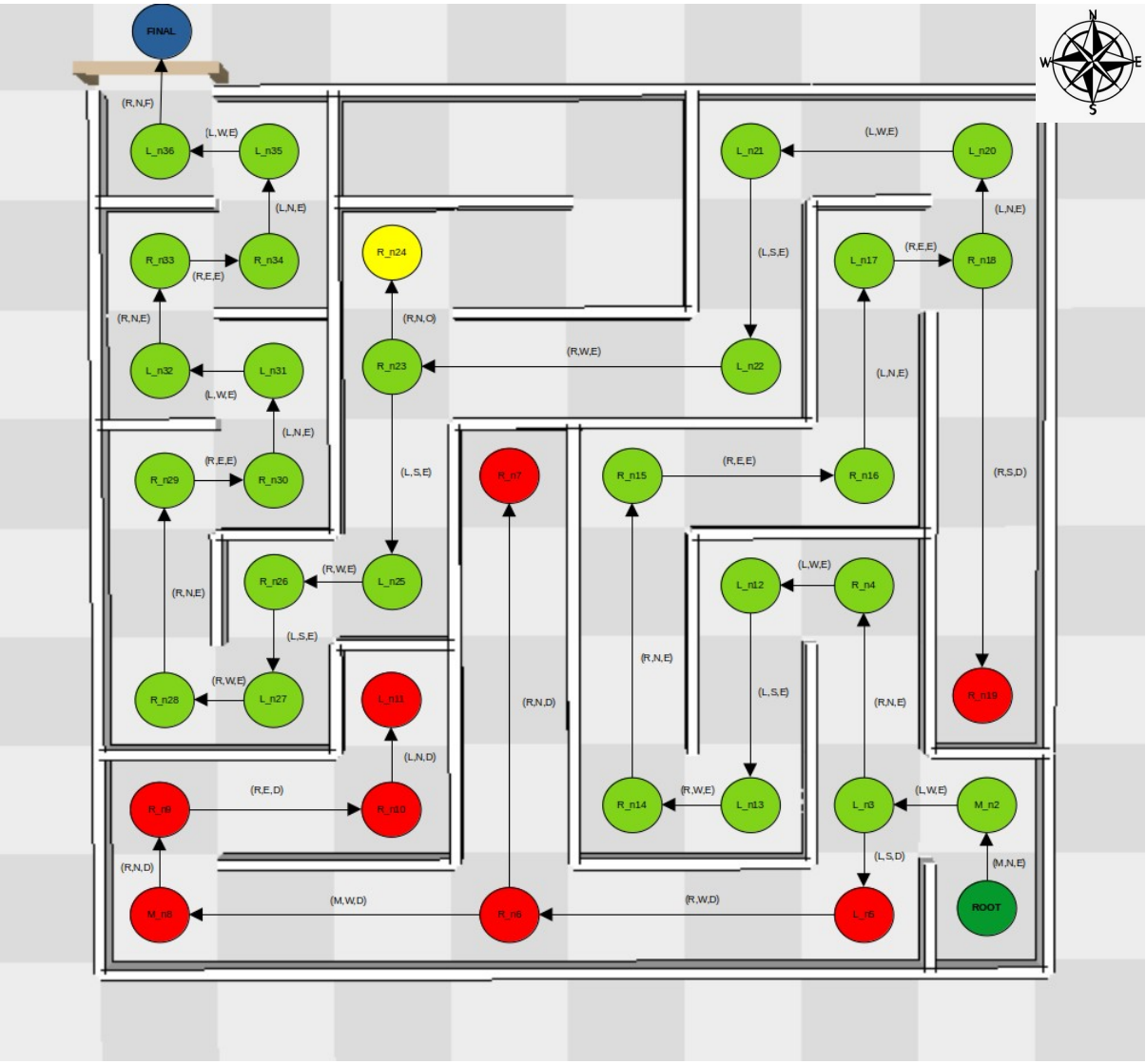


Figura 6: Il percorso che permette di arrivare all'uscita del labirinto è dato dai nodi in verde

Sceglierà di andare a SOUTH poiché nella lista di priorità ha la priorità più alta. Ruoterà di 90° in senso antiorario per andare verso SOUTH. Proseguirà ad esplorare il labirinto con la stessa metodologia fino a trovare l'uscita, aggiornando opportunamente gli attributi dei nodi e di passare in modalità ESCAPING per uscire dai vicoli ciechi.

Nell'immagine i nodi sono colorati secondo le loro proprietà:

- Verde scuro: ROOT
- Verde: EXPLORED
- Giallo: OBSERVED
- Rosso: DEAD_END
- Blu: FINAL

Il codice di *update_tree(actions, action_chosen)* è il seguente:

```
def update_tree(self, actions, action_chosen):
    """
    This method is used to update the tree of the maze accordingly to the actions and the
current state
    of the robot.
    The update is done if only the robot is in SENSING mode and the actions and
action_chosen type must
    be a Compass or a list of Compass elements respectively.
    """

    global LOG_SEVERITY

    if not self._state == State.SENSING:
        return

    if self._mode == Mode.EXPLORING:
        if Logger.is_loggable(LOG_SEVERITY, "mid"):
            self.__class_logger.log("*** UPDATING TREE (MODE: EXPLORING) ***", "gray",
newline=True)

        """
        Different actions returned by Control Policy.
        In this section are added the nodes of the tree based on the available actions,
        updating also the current node as EXPLORED
        """
        for action in actions:
```

```
            dict_ = f_r_l_b_to_compass(self.orientation)
            if dict_["FRONT"] == action:
                node = Node("M_" + self.tree.generate_node_id(), action)
                self.tree.append(node, DIRECTION.MID)
                self.tree.regress()
                self.number_of_nodes += 1
                if Logger.is_loggable(LOG_SEVERITY, "mid"):
                    self.__class_logger.log("ADDED MID", "dkgreen")
            if dict_["LEFT"] == action:
                node = Node("L_" + self.tree.generate_node_id(), action)
                self.tree.append(node, DIRECTION.LEFT)
                self.tree.regress()
                self.number_of_nodes += 1
                if Logger.is_loggable(LOG_SEVERITY, "mid"):
                    self.__class_logger.log("ADDED LEFT", "dkgreen")
            if dict_["RIGHT"] == action:
                node = Node("R_" + self.tree.generate_node_id(), action)
                self.tree.append(node, DIRECTION.RIGHT)
                self.tree.regress()
                self.number_of_nodes += 1
                if Logger.is_loggable(LOG_SEVERITY, "mid"):
                    self.__class_logger.log("ADDED RIGHT", "dkgreen")

        self.tree.current.set_type(Type.EXPLORED)

        """
        Only one action that has been decided by DMP.
        In this section it is updated the current node of the tree based on the action chosen
        """
        dict_ = f_r_l_b_to_compass(self.orientation)
        if dict_["FRONT"] == action_chosen:
            self.tree.set_current(self.tree.current.mid)
        elif dict_["LEFT"] == action_chosen:
            self.tree.set_current(self.tree.current.left)
        elif dict_["RIGHT"] == action_chosen:
            self.tree.set_current(self.tree.current.right)

    elif self._mode == Mode.ESCAPING:
    if Logger.is_loggable(LOG_SEVERITY, "mid"):
        self.__class_logger.log("*** UPDATING TREE (MODE: ESCAPING) ***", "gray",
newline=True)

    """
    In this section it is updated the type property of the current node accordingly if
    the current node is a leaf or has children that are all dead end, otherwise it is updated
the current node
    """

    cur = None

    # The node is a leaf
    if self.tree.current.is_leaf:
```

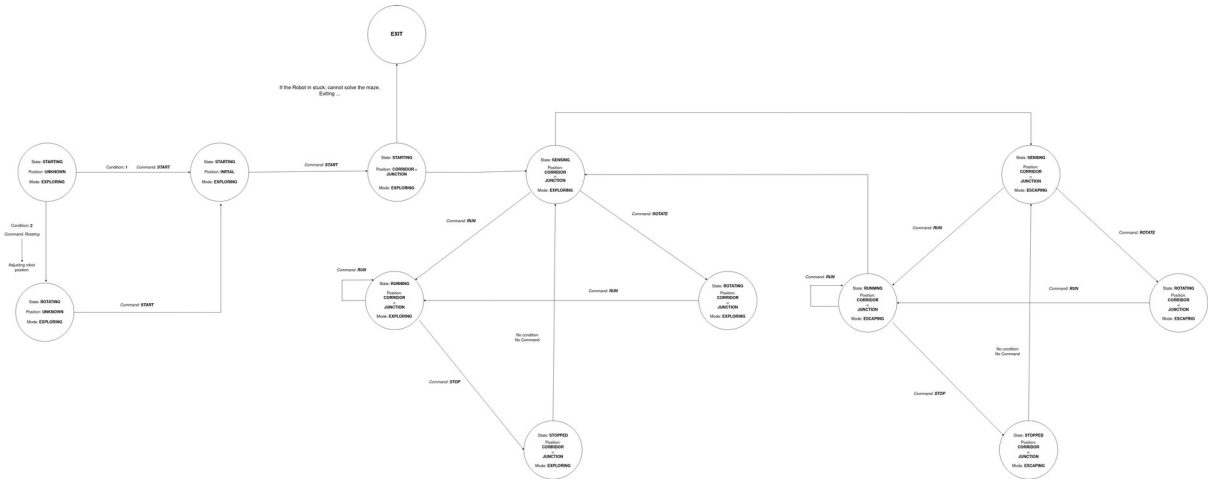
```
        self.tree.current.set_type(Type.DEAD_END)
        self.number_of_dead_end += 1
        if Logger.is_loggable(LOG_SEVERITY, "low"):
            self.__class_logger.log("*** DEAD END NODE DETECTED ***", "green")
            self.__class_logger.log(" >>>> REGRESSION <<<< ", "yellow", newline=True)
            self.__class_logger.log(f" --CURRENT NODE: {self.tree.current}", "yellow")
            self.__class_logger.log(f" --PARENT NODE: {self.tree.current.parent}", "yellow")

        cur = self.tree.current.parent

    # The children are all DEAD END
    elif ((self.tree.current.has_left and self.tree.current.left.type == Type.DEAD_END) or
          self.tree.current.left is None) and \
            ((self.tree.current.has_right and self.tree.current.right.type == Type.DEAD_END)
             or self.tree.current.right is None) and \
            ((self.tree.current.has_mid and self.tree.current.mid.type == Type.DEAD_END)
             or self.tree.current.mid is None):
        self.tree.current.set_type(Type.DEAD_END)
        self.number_of_dead_end += 1
        if Logger.is_loggable(LOG_SEVERITY, "low"):
            self.__class_logger.log("*** ALL CHILDREN ARE DEAD END NODES ***", "green")
            self.__class_logger.log(" >>>> REGRESSION <<<< ", "yellow", newline=True)
            self.__class_logger.log(f" --CURRENT NODE: {self.tree.current}", "yellow")
            self.__class_logger.log(f" --PARENT NODE: {self.tree.current.parent}", "yellow")

        cur = self.tree.current.parent

    else:
        """
        This is the case when the action chosen by DMP is an action that brings the robot
        to an OBSERVED node and this node becomes the current node.
        """
        if Logger.is_loggable(LOG_SEVERITY, "low"):
            self.__class_logger.log("No leaf or DEAD END children", "yellow+", italic=True)

        if self.tree.current.has_left and self.tree.current.left.action == action_chosen:
            cur = self.tree.current.left
        elif self.tree.current.has_mid and self.tree.current.mid.action == action_chosen:
            cur = self.tree.current.mid
        elif self.tree.current.has_right and self.tree.current.right.action == action_chosen:
            cur = self.tree.current.right
        else:
            if Logger.is_loggable(LOG_SEVERITY, "low"):
                self.__class_logger.log("!!! ESCAPING ERROR UPDATING CURRENT !!!", "dkred",
True, True)
                self.__class_logger.log(" >>>>  EXITING  <<<< ", "red", italic=True)
            self.virtual_destructor()
            exit(-1)

    self.tree.set_current(cur)
```

4.5 Diagramma degli stati *[bozza]*

lo stato effettivo e reale del robot è ottenuto dalla combinazione della terna:

- State
- Position
- Mode

Nella seguente immagine sono presenti tutti gli stati che il robot può assumere e come interagiscono tra loro. Il robot passa da uno stato ad un altro in seguito ad un evento, più precisamente dal comando generato dal Controller.



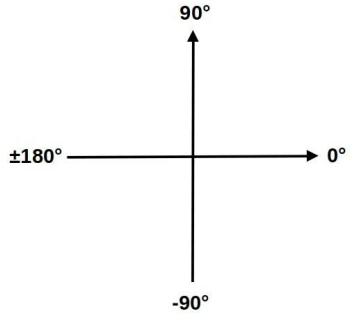
4.6 Junction Time *[bozza]*

Il robot, per potersi posizionare correttamente al centro di una giunzione (JUNCTION) fa uso di un timer chiamato `junction_sim_time` che è stato calcolato effettuando diversi test modificando la velocità della macchina `speed` e misurando il tempo necessario a raggiungere il centro della giunzione. Il tempo di giunzione si ottiene dal rapporto di 0.25 e la velocità in m/s.

4.7 Rotazione

Quando il robot non può proseguire avanti perché ha di fronte un muro oppure la Decision Making Policy ha scelto una via differente, è necessario che il robot faccia una rotazione su sé stesso di un numero di gradi scelti da quest'ultima per poter prendere la giusta direzione e procedere con l'esplorazione del labirinto.

Prima di effettuare la rotazione, il Controller decide l'angolo finale: 0°, 90°, -90° o +-180° che il robot deve raggiungere.



Ottiene poi l'orientazione corrente rispetto all'asse z del robot nel labirinto chiedendo al simulatore Coppelia il valore gamma *g* poi opportunamente convertito in gradi centigradi. Il metodo del Controller `rotate_to_final_g(vel, final_g)` si occupa di chiamare altri metodi per effettuare la rotazione ed esegue le seguenti operazioni:

- ottiene il valore *g* iniziale del robot, convertito in gradi centigradi: `init_g`
- calcola il più piccolo angolo tra i due possibili (orario e antiorario) in cui ruotare prendendo come angoli di riferimento quello iniziale in cui il robot si trova e quello finale da raggiungere: `degrees`
- viene determinato il senso di rotazione: orario o antiorario secondo l'angolo di rotazione scelto precedentemente: `c (clockwise)`

```
def rotate_to_final_g(self, vel, final_g):
    """ Rotation function that rotates the robot until it reaches final_g """
    self._body.stop()
    init_g = self._body.get_orientation_deg()
```

```
    degrees, c = self.best_angle_and_rotation_way(init_g, final_g)
    self.__do_rotation(vel=vel, c=c, degrees=degrees, final_g=final_g)
    self._body.stop()
```

- Una volta scelto l'angolo più piccolo e il senso di rotazione, si effettua la rotazione:

  `__do_rotation(vel=vel, c=c, degrees=degrees, final_g=final_g)`

Dato che nel simulatore, come nella realtà, è quasi impossibile raggiungere un angolo preciso senza approssimazioni, sono stati implementati metodi per controllare se il robot ha raggiunto la posizione finale entro un range di gradi e di aggiustare la posizione se è al di fuori di questo range effettuando vari tentativi. Più precisamente, l'orientazione del robot deve raggiungere l'intervallo:

`[degrees - delta, degrees + delta]`

dove:

- `degrees`: sono i gradi finali da raggiungere
- `delta`: definisce la grandezza dell'intervallo. Il valore scelto in modo arbitrario è 2 e implica che l'intervallo in cui il robot si deve posizionare è di: 2 * *delta* = 2 * 2 = 4 gradi.

```
def __do_rotation(self, vel, c: Clockwise, degrees, final_g):
    """
    This method calls __rotate to perform the rotation.
    It also checks the outcome of check_orientation:
    i) If the orientation is correct nothing happens
    ii) If the orientation is not correct it calls adjust_orientation to
        rotate the robot to the correct orientation
    It stops the program when it is not possible to orient correctly the robot when it
exceeds the number
    of attempts (Critical case)
    """

    global LOG_SEVERITY
```

```
degrees = abs(degrees)
it = 0

self._body.stop()
self.__rotate(vel, c, degrees)

ok, curr_g, limit_range = self.check_orientation(final_g)

if not ok:
    ok, it = self.adjust_orientation(final_g)

# CRITICAL CASE
if it == MAX_ROT_ATTEMPTS:
    if Logger.is_loggable(LOG_SEVERITY, "low"):
        self.__class_logger.log(" ** MAX ATTEMPTS REACHED ** ", "dkred", True, True)
        self.__class_logger.log(" >>>>  EXITING  <<<< ", "dkred", italic=True)
    self.virtual_destructor()
    exit(-1)
```

In sostanza, i metodi necessari per effettuare correttamente la rotazione sono:

`rotate_to_final_g(vel, final_g)` che chiama `do_rotation`

`do_rotation` *chiama:*

- `__rotate:` si occupa di far ruotare il robot.
- `check_orientation:` verifica se l'orientazione corrente del robot è corretta.
- `adjust_orientation:` aggiusta l'orientazione corrente se non è corretta facendo vari tentativi. Il numero di tentativi massimo si può modificare aggiornando il file *config.conf.*

4.8 Bilanciamento

Il robot, durante la sua corsa, potrebbe trovarsi troppo vicino ad un muro rischiando di urtare e quindi creando situazioni non ottimali che bisogna assolutamente evitare. Per risolvere questo problema, è possibile attivare l'auto-

bilanciamento semplicemente modificando il file *config.conf.* Il bilanciamento che abbiamo implementato è triviale:

- Una parte laterale (destra o sinistra) del robot si trova troppo vicino ad un muro e supera la distanza di sicurezza.
- Il robot si ferma e comincia a ruotare in modo tale da avere la parte frontale contro il muro. In questo modo, il sensore Front viene utilizzato per tenere sotto controllo la distanza tra il robot e il muro.
- Il robot fa retromarcia fino a quando non raggiunge una posizione centrale tra i muri del labirinto ad un'adeguata distanza di sicurezza (facendo uso del sensore Front).
- Una volta che si è posizionato al centro, comincia a ruotare nel senso opposto rispetto al senso della prima rotazione. Si ferma quando si trova nella stessa direzione che aveva prima del bilanciamento e, da qui, prosegue l'esplorazione del labirinto.

5 File di configurazione e log dei dati

Alcuni parametri del robot possono essere modificati direttamente cambiando i valori assegnati nel file *config.conf* che si trova nella directory *MetalPhoenix/robot/resources/data.* Di seguito è presente un esempio del file di configurazione.

Ad esempio si può modificare il parametro speed (velocità massima dei motori del robot) da un minimo di 5 a un massimo di 10. Se si scelgono valori al di fuori degli intervalli proposti, il robot potrebbe comportarsi in modo imprevedibile.

```
[ROBOT]
# Speed: [Min, Max] = [5, 10]
speed = 8
# Rotation speed: [Min, Max] = [2, 4]
rot_speed = 2
# Safe  distance: [Min, Max] = [0.18, 0.20]
safe_dist = 0.20
```

```
# Safe side distance: [Min, Max] = [0.12, 0.15]
safe_side_dist = 0.12
# Max rotation attempts
max_rot_attempts = 20
# Priority list
priority_list = NORTH, WEST, EAST, SOUTH
# Intelligence. Available options: [low, mid, high (not developed yet)]
intelligence = low
# Auto_balancing. Available options: [on, off]
auto_balancing = on

[COPPELIA]
ip = 127.0.0.1
port = 19997

[MAZE]
# Number/Id of the maze. Change this value if the maze changes.
maze_number = 1

[UTILITY]
controllerlog = ../resources/data/logs/controller/logfile
bodylog = ../resources/data/logs/body/logfile
agentlog = ../resources/data/logs/agent/logfile
ext = log
# Log severity (how much information must be logged).
# Available options: [none, low, mid, high]
severity = high


# # # # # # # Compass # # # # #
#              NORTH: 90             #
#  WEST: 180            EAST: 0     #
#             SOUTH: -90             #
# # # # # # # # # # # # # # # # # #
```

Per tener traccia di tutto ciò che accade durante la simulazione, i dati generati dall'Agent, Body e Controller vengono memorizzati in differenti file con estensione *.log* che si trovano nella directory *MetalPhoenix/robot/resources/data/logs*.

Di seguito è riportato un esempio di una parte di file di log del Controller:

```
[13:15:8] [Controller][INFO] -> LOG SEVERITY: HIGH

[13:15:8] [Controller][INFO] -> CONTROLLER LAUNCHED
```

```
[13:15:9] [Controller][INFO] ->  >>>>>> NEW ALGORITHM CYCLE <<<<<<
[13:15:10] [Controller][INFO] -> --MODE: Mode.EXPLORING
[13:15:10] [Controller][INFO] -> --ACTIONS: [[<Command.START: -1>, None]]
[13:15:10] [Controller][INFO] -> --ACTION: [<Command.START: -1>, None]
[13:15:10] [Controller][INFO] -> --CURRENT NODE: [ Name: root, Type: Type.OBSERVED,
Action: None, Parent: None, Left: None, Mid: None, Right: None ]
[13:15:10] [Controller][DEBUG] -> --CURRENT TREE:
[13:15:10][NOHEADER] -> {'root': {}}

[13:15:10] [Controller][DEBUG] -> --CURRENT NODE: [ Name: root, Type:
Type.OBSERVED, Action: None, Parent: None, Left: None, Mid: None, Right: None ]
[13:15:10] [Controller][INFO] -> --Available actions: [[<Command.START: -1>, None]]
[13:15:10] [Controller][DEBUG] -> --(STATE, POSITION): (State.STARTING,
Position.INITIAL)
[13:15:10] [Controller][DEBUG] -> --Performing action: [<Command.START: -1>, None]

[13:15:10] [Controller][DEBUG] ->  ~~~ [ACTION TIME] ~~~
[13:15:10] [Controller][DEBUG] ->  ** COMMAND START **
[13:15:10] [Controller][DEBUG] -> --(STATE, POSITION): (State.STARTING,
Position.INITIAL)

[13:15:10] [Controller][INFO] ->  >>>>>> NEW ALGORITHM CYCLE <<<<<<
[13:15:10] [Controller][INFO] -> --MODE: Mode.EXPLORING
[13:15:10] [Controller][INFO] -> --ACTIONS: [[<Command.START: -1>, None]]
[13:15:10] [Controller][INFO] -> --ACTION: [<Command.START: -1>, None]
[13:15:10] [Controller][INFO] -> --CURRENT NODE: [ Name: root, Type: Type.OBSERVED,
Action: None, Parent: None, Left: None, Mid: None, Right: None ]

[13:15:10] [Controller][DEBUG] -> *** UPDATING TREE (MODE: EXPLORING) ***
[13:15:10] [Controller][DEBUG] -> --CURRENT TREE:
[13:15:10][NOHEADER] -> {'root': {}}

[13:15:10] [Controller][DEBUG] -> --CURRENT NODE: [ Name: root, Type:
Type.EXPLORED, Action: None, Parent: None, Left: None, Mid: None, Right: None ]
[13:15:10] [Controller][INFO] -> --Available actions: [[<Command.START: -1>, None]]
[13:15:10] [Controller][DEBUG] -> --(STATE, POSITION): (State.SENSING,
Position.CORRIDOR)
[13:15:10] [Controller][DEBUG] -> --Performing action: [<Command.START: -1>, None]

[13:15:10] [Controller][DEBUG] ->  ~~~ [ACTION TIME] ~~~
[13:15:10] [Controller][DEBUG] ->  ** COMMAND START **
[13:15:10] [Controller][DEBUG] -> --(STATE, POSITION): (State.SENSING,
Position.CORRIDOR)

[13:15:10] [Controller][INFO] ->  >>>>>> NEW ALGORITHM CYCLE <<<<<<
[13:15:10] [Controller][DEBUG] -> Control policy EXPLORING
[13:15:10] [Controller][INFO] -> --MODE: Mode.EXPLORING
[13:15:10] [Controller][INFO] -> --ACTIONS: [[<Command.RUN: 1>, <Compass.NORTH:
90.0>]]
[13:15:10] [Controller][INFO] -> --ACTION: [<Command.RUN: 1>, <Compass.NORTH:
90.0>]
[13:15:10] [Controller][INFO] -> --CURRENT NODE: [ Name: root, Type: Type.EXPLORED,
```

```
Action: None, Parent: None, Left: None, Mid: None, Right: None ]

[13:15:10] [Controller][DEBUG] -> *** UPDATING TREE (MODE: EXPLORING) ***
[13:15:10] [Controller][INFO] -> ADDED MID
[13:15:10] [Controller][DEBUG] -> --CURRENT TREE:
[13:15:10][NOHEADER] -> {'root': {'M_n2': '(M , N , O)'}, 'M_n2': {}}

[13:15:10] [Controller][DEBUG] -> --CURRENT NODE: [ Name: M_n2, Type:
Type.OBSERVED, Action: 90.0, Parent: root, Left: None, Mid: None, Right: None ]
[13:15:10] [Controller][INFO] -> --Available actions: [[<Command.RUN: 1>,
<Compass.NORTH: 90.0>]]
[13:15:10] [Controller][DEBUG] -> --(STATE, POSITION): (State.SENSING,
Position.CORRIDOR)
[13:15:10] [Controller][DEBUG] -> --Performing action: [<Command.RUN: 1>,
<Compass.NORTH: 90.0>]

[13:15:10] [Controller][DEBUG] ->  ~~~ [ACTION TIME] ~~~
[13:15:10] [Controller][DEBUG] ->  ** COMMAND RUN **
[13:15:10] [Controller][DEBUG] -> --(STATE, POSITION): (State.RUNNING,
Position.CORRIDOR)
```

Come si può notare, ogni riga è strutturata nel seguente modo:

1. timestamp
2. nome della classe che ha generato il log
3. tipologia di informazione (INFO, DEBUG, ecc)
4. informazione

6 Diagramma delle classi