UNIVERSITÀ DI BRESCIA
FACOLTÀ DI INGEGNERIA
Dipartimento di Elettronica per l'Automazione

Laboratorio di Robotica Avanzata
Advanced Robotics Laboratory

Corso di Robotica Mobile
(Prof. Riccardo Cassinis)

Parcheggio di Speedy : Movimentazione

Elaborato di esame di: **Stefano Cadei, Manuel Piemonti, Costantino Mauro Sanfilippo, Marco Serina, Alberto Venturini**

Consegnato il: **27 giugno 2006**

Sommario

L'elaborato svolto costituisce una delle due parti del progetto più ampio finalizzato alla guida del robot ActivMedia Pioneer "Speedy" alla sua stazione di ricarica. Si è affrontato il problema della movimentazione del veicolo in base ai dati forniti da un modulo di gestione di una videocamera posta a bordo del robot stesso.

1. Introduzione

Per consentire al robot di riconoscere la stazione di ricarica e di entrarvi correttamente, l'ambiente di lavoro è stato opportunamente strutturato introducendo due elementi ben identificabili: una linea gialla diretta verso la stazione e un rettangolo verde posto all'inizio di tale linea che permette di raggiungere la posizione di partenza per la vera e propria operazione di parcheggio. La stazione inoltre presenta due guide che facilitano la fase finale di parcheggio del robot, anche in caso di un ingresso non perfettamente allineato, e due fermi metallici che permettono a Speedy di arrestarsi nella posizione desiderata.

Figura 1: Dettaglio della stazione di ricarica e dei due marker da riconoscere

L'operazione di parcheggio all'interno della stazione di ricarica è stata suddivisa in due fasi principali:

- ricerca del marker verde attraverso l'esplorazione casuale dell'ambiente
- avvicinamento alla stazione di ricarica seguendo la traiettoria indicata dalla linea gialla

Inizialmente infatti il robot esplora l'ambiente con movimento casuale alla ricerca del marker di colore verde, sufficientemente grande da poter essere rilevato dalla webcam già a una discreta distanza. Una volta individuato e raggiunto tale marker, il robot ruota su se stesso alla ricerca della striscia gialla che lo guida fino alla stazione di ricarica. In questa fase il compito del robot è quello di seguire tale linea correggendo man mano il proprio orientamento. Una volta che il robot è giunto alla stazione e le ruote hanno toccato gli appositi fermi meccanici, mandando in stallo i motori, il programma termina e il robot si ferma.

Figura 2: Speedy sul marker verde e nella posizione finale di ricarica

La soluzione del problema complessivo è composta da due attività complementari, svolte dai due gruppi cui è stato assegnato il progetto: l'elaborazione delle immagini acquisite mediante la videocamera e il controllo dei movimenti del robot sulla base delle informazioni ottenute. In particolare il gruppo che si è occupato della visione ha elaborato le immagini acquisite in modo da riconoscere il marker verde e la striscia gialla, calcolandone la relativa posizione. Il secondo gruppo, che si è occupato del movimento, ha utilizzato queste informazioni sulla posizione del rettangolo e della linea per la navigazione del robot e per prendere decisioni sui comportamenti e sui movimenti da compiere per raggiungere l'obiettivo.

Questa porzione del progetto è dedicata alla gestione dei dati inviati dal modulo dedicato alla visione al fine di regolare la movimentazione del robot.

2. Il problema affrontato

Il problema generale affrontato nella realizzazione del sistema è quello di consentire al robot di localizzare la propria stazione di ricarica e raggiungerla attraverso l'elaborazione delle immagini ricavate dalla videocamera posta sul robot.

Il problema così presentato, che a prima vista potrebbe sembrare relativamente semplice da risolvere, nella realtà nasconde diverse problematiche.

Il robot non ha conoscenza spaziale dell'ambiente in cui si trova, non possedendo una mappa dell'ambiente stesso e quindi non sapendo precisamente dove è situata la stazione di ricarica; per questo motivo sono essenziali le informazioni rilevate dalla videocamera e ancora di più le informazioni che le elaborazioni delle immagini acquisite possono fornire.

La stazione di ricarica è realizzata in modo da permettere l'ingresso e la connessione con i terminali di metallo e la batteria in modo agevole e con una certa tolleranza sulla posizione del robot grazie alle apposite guide che permettono di raggiungere i contatti in modo semplice. La parte più complicata è quella di consentire al robot di arrivare in una posizione ben orientata rispetto alla stazione.

Per raggiungere l'obiettivo prefissato è possibile operare su due fronti, il primo riguardante l'avvicinamento all'obiettivo e il secondo che consiste nel posizionamento fine all'interno della stazione di ricarica.

Il problema dell'avvicinamento è riconducibile al problema di raggiungere un punto identificato da un marker verde che, come già accennato, nel nostro caso è un rettangolo posizionato sul pavimento abbastanza vicino alla stazione di ricarica. Per quanto riguarda l'ingresso vero e proprio in quest'ultima, il robot deve seguire una traiettoria rettilinea identificata dalla già citata linea gialla che congiunge il marker verde con la stazione di ricarica. Mentre il primo movimento può essere effettuato anche con un andamento non troppo preciso che porti il robot nelle vicinanze dell'obiettivo, il secondo è più delicato perché è fondamentale che l'ingresso nel centro di ricarica sia fatto con una orientazione non troppo diversa da quella ideale.

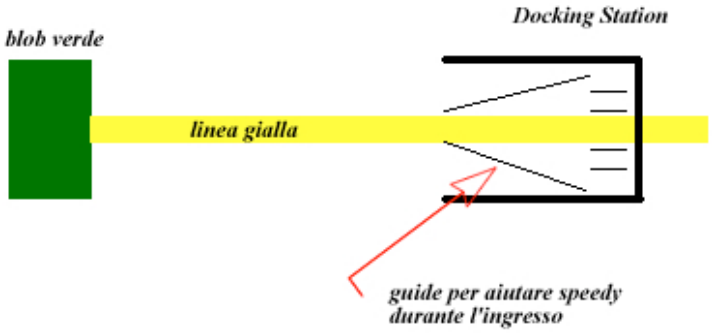

Figura 2.2 Schematizzazione di marker verde, linea gialla e docking station

Per una corretta riuscita del progetto i due gruppi devono creare un sistema di comunicazione tra i loro moduli e fare in modo che l'interazione tra questi due sottosistemi porti il robot generale a comportarsi nel modo corretto. Fondamentale per la parte di visione è la capacità di riconoscere il marker verde e la linea suddetti, distinguendoli al meglio da possibili fonti di errore (ad esempio oggetti con colori simili) e fornendo le posizioni in modo preciso.

In sintesi, quindi, l'obiettivo è quello di muovere il robot inizialmente verso il marker verde e successivamente fargli seguire la linea gialla fino a portarlo nella stazione di ricarica, permettendogli di accorgersi di essere arrivato a destinazione.

3. La soluzione adottata

Il problema descritto al paragrafo precedente non è di immediata soluzione e si è perciò optato per una sua suddivisione in sottoproblemi più semplici, cercando di seguire il paradigma "divide-et-impera".

I sottoproblemi individuati sono i seguenti:

1) Comunicazione con il modulo dedicato alla visione
2) Raggiungimento del marker verde
3) Individuazione della linea che conduce alla stazione di ricarica
4) Inseguimento della linea che conduce alla stazione di ricarica
5) Riconoscimento dell'arrivo nella stazione di ricarica

Si noti come il primo sottoproblema non sia legato direttamente alla movimentazione del robot, quanto piuttosto all'interfacciamento con il modulo gestito dal secondo gruppo di lavoro di questo progetto.

Gli altri quattro sottoproblemi sono invece legati tra di loro, sia perché riguardano tutti il problema della navigazione, sia perché è necessario un coordinamento tra ciascuno dei moduli che li gestisce. In particolare, come verrà meglio illustrato nel paragrafo 3.3, si è resa necessaria l'implementazione di una macchina a stati, ciascuno dei quali corrisponde ad un sottoproblema del movimento.

Si segnala inoltre che, poiché la programmazione del robot avviene attraverso le librerie Aria [1] progettate per il linguaggio C++, si è deciso di utilizzare tale linguaggio per la gestione di tutte le parti del progetto, sfruttando quando necessario anche la sua compatibilità con il linguaggio C.

3.1. Organizzazione del progetto

Il progetto è stato realizzato cercando di seguire l'approccio modulare descritto in precedenza. In particolare, l'implementazione dell'applicativo è stata effettuata in vari file sorgenti, ciascuno destinato a risolvere un compito specifico, e ciascuno dotato di un proprio header file. Un makefile appositamente creato ha poi permesso di gestire facilmente tutti i riferimenti fra di essi in fase di compilazione.

Il file più importante è `main.cpp` che, come dice il nome stesso è destinato a contenere il programma principale. I compiti di quest'ultimo sono sostanzialmente i seguenti:

- Aprire una connessione con il robot
- Comunicare con l'applicativo di gestione della telecamera
- Impostare il comportamento da attivare in ogni stato in cui si trova il robot
- Gestire i cambiamenti di stato
- Chiudere la connessione con il robot

La parte principale è ovviamente quella che gestisce gli stati del robot. Pur non entrando nei dettagli dell'automa, che verranno illustrati nel paragrafo 3.3, è utile fornire fin da subito una breve descrizione di come il `main` gestisca questa parte, visto che rappresenta il cuore di tutto l'applicativo.

Lo stato in cui si trova il robot è rappresentato da una variabile globale di tipo intero, il cui valore viene controllato all'interno di un ciclo da cui si esce solamente in caso di arrivo alla docking station. Tale ciclo consta sostanzialmente di un reperimento delle coordinate del marker verde o della linea e nel controllo di una variabile globale (`global_state`) che indica lo stato corrente. Un'ulteriore variabile globale booleana (`state_changed`) permette di sapere se lo stato deve essere cambiato e, in caso affermativo, il main gestisce il passaggio allo stato successivo attraverso una struttura di controllo switch-case che provvede ad aggiungere/rimuovere i comportamenti necessari. Più precisamente, ciascuno stato è caratterizzato da un insieme di azioni (istanze della classe Aria `ArAction`), alcune predefinite ed altre realizzate appositamente, che tramite l'utilizzo di opportune priorità permettono al robot di esibire una serie di comportamenti relativamente complessi. In altre parole, quindi, ciascuno stato viene gestito in logica comportamentale.

Gli altri file che compongono il progetto sono quelli dedicati alle implementazioni delle azioni che caratterizzano ciascuno stato. Ciascuno di questi, infatti, pur essendo costituito da un insieme di azioni, ne possiede una che lo contraddistingue nettamente dagli altri: ad esempio, lo stato di inseguimento del marker verde possiede un'azione denominata `ActionFollowBlob` facilmente distinguibile dalle altre, comunque presenti, destinate ad evitare ostacoli o risolvere casi di stallo.

I paragrafi seguenti presentano in maggior dettaglio le soluzioni dei problemi di comunicazione tra gli applicativi, la logica della macchina a stati e i dettagli sulle azioni coinvolte.

3.2. Comunicazione con l'applicativo di gestione della videocamera

Per la gestione della comunicazione tra il modulo dedicato al movimento e quello dedicato alla videocamera sono state proposte inizialmente varie soluzioni, ciascuna con i propri pregi e difetti. In questa analisi preliminare sono stati coinvolti rappresentanti di entrambi i gruppi di lavoro, perché entrambi i moduli devono utilizzare il medesimo canale di comunicazione. L'esito di questa fase di analisi è stato quello di optare per l'utilizzo di due processi distinti che utilizzano una comunicazione basata su socket, usando quindi una comunicazione esplicita. Le ragioni che hanno spinto verso tale scelta sono sostanzialmente quattro:

- Il meccanismo dei socket è ben noto a tutti i componenti dei due gruppi di lavoro
- L'utilizzo dei socket permette di utilizzare un approccio client server tra i due applicativi, anch'esso ben noto a tutti gli studenti
- L'utilizzo dei socket permette di inviare diversi tipi di messaggi fra i due applicativi
- Non c'è necessità di coordinare l'accesso ad un'area di memoria condivisa

Si è quindi deciso di fare in modo che il processo dedicato alla visione fosse un server (messo in ascolto sulla porta 20000) destinato a rispondere ai comandi inviati dal processo dedicato alla gestione del movimento, che quindi corrisponde al client. Le richieste previste sono solamente due:

- Richiesta delle coordinate del baricentro del marker verde
- Richiesta dell'ascissa della linea gialla che conduce alla stazione di ricarica

Al fine di semplificare al massimo la fase di comunicazione i due messaggi sono stati implementati attraverso il carattere "0" ed il carattere "1", rispettivamente (tutti i dati scambiati via socket sono in formato ASCII).

In aggiunta alle suddette richieste il protocollo prevede un ulteriore messaggio che il client invia al server per segnalare l'avvenuto parcheggio del robot, e permettere quindi al server di chiudere la connessione. Anche questo messaggio è stato implementato tramite l'invio di un solo carattere, e precisamente il carattere '2'.

Le risposte attese sono stringhe di caratteri nel formato "X Y" (la stringa è formata dalle due sequenze di cifre che rappresentano due valori numerici separate da uno spazio). Nel caso di risposta ad una richiesta sulla posizione del marker verde i due valori rappresentano le coordinate del suo baricentro, mentre nel caso di richiesta relativa alla linea ciò che il client considera è solamente il primo dei due valori inviati, mentre il secondo viene completamente ignorato.

Nel caso in cui l'oggetto di cui si richiedono le coordinate non sia presente nella visuale della telecamera è previsto l'invio da parte del server della stringa "999 999".

3.3. La macchina a stati

Come già accennato in precedenza si è deciso di affrontare il problema realizzando una macchina a stati, in cui il robot fosse in grado di passare da una suite di comportamenti ad un'altra, a seconda degli eventi che attivano le transizioni di stato. L'approccio è un ibrido tra il paradigma comportamentale e il paradigma imperativo: infatti l'utilizzo di una macchina a stati che guidi il comportamento del robot in risposta agli stimoli ambientali è sicuramente una soluzione classica ai problemi della robotica mobile; tuttavia all'interno di ogni stato, il comportamento del robot è guidato da comportamenti di uso generale, la cui combinazione, con relative priorità e regole di inibizione, realizza un'architettura "subsumption" [2], quindi un approccio sicuramente più moderno, e reso particolarmente efficace in quanto pienamente supportato dalle librerie Aria. Infine è necessario notare come in realtà gli stati utilizzati siano solamente tre, e le transizioni siano particolarmente semplici: una scelta che dimostra la predilezione per l'uso di comportamento.

3.3.1. Specifica della macchina a stati

Il seguente statechart (Figura 3.1) rappresenta la macchina a stati realizzata dall'intero programma di docking. Nei commenti a ogni stato sono elencate le azioni (`ArAction`) attivate, e tra parentesi le relative priorità.

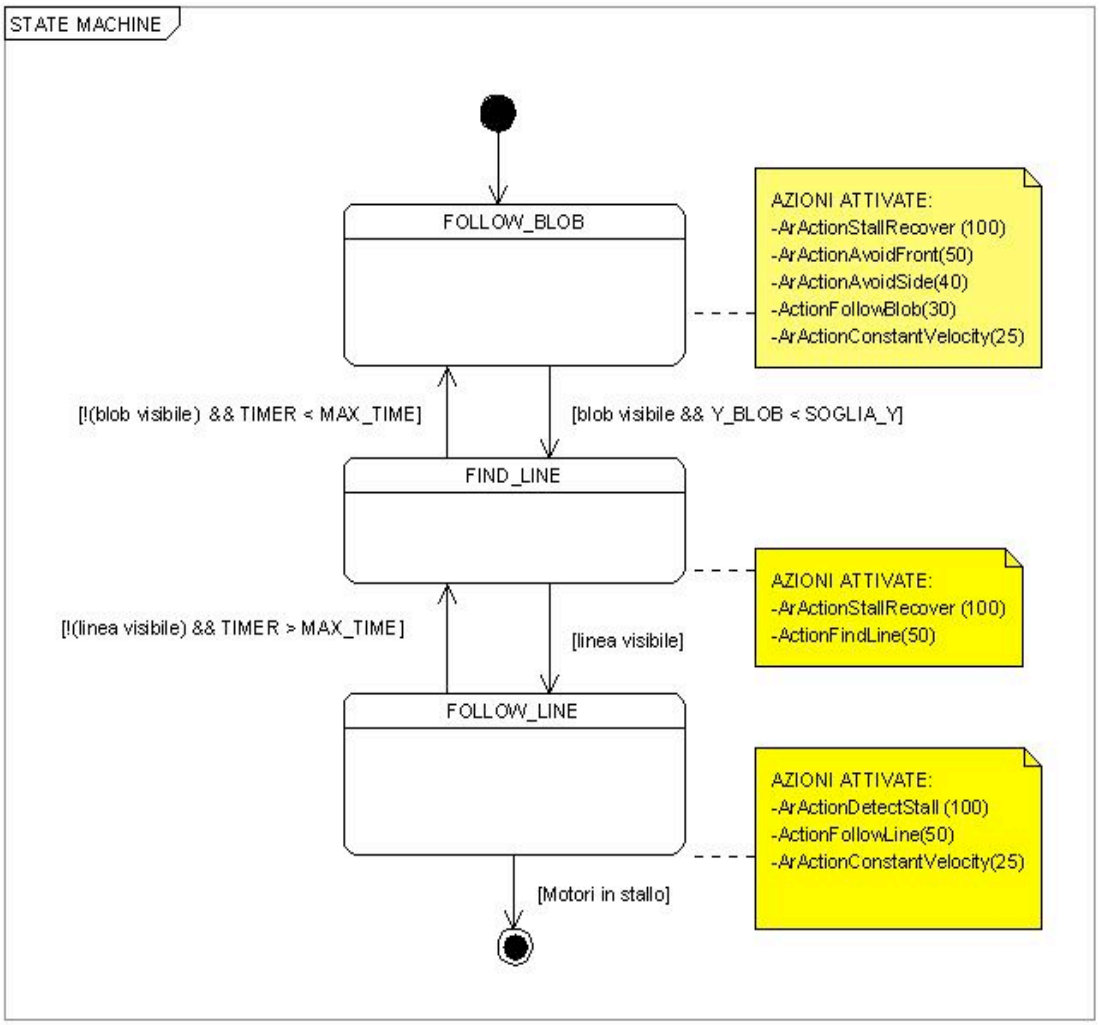

Figura 3.1 Macchina a stati

3.3.2. Descrizione degli stati

3.3.2.1 Stato FOLLOW_BLOB:

I comportamenti semplici attivati nello stato `FOLLOW_BLOB`, combinati dal resolver tenendo conto delle le loro priorità relative, realizzano un comportamento piuttosto complesso che permette al robot di cercare il marker verde vagando nella stanza evitando ostacoli frontali e laterali. In questo stato vengono richieste alla videocamera le coordinate del baricentro del marker verde. Se la videocamera fornisce delle coordinate ammissibili, il robot modifica il suo orientamento, cercando di mantenere la coordinata orizzontale vicina al centro dell'immagine, ma continuando ad evitare ostacoli. L'unica transizione di stato avviene quando la coordinata verticale del baricentro del marker, fornita dalla telecamera, è molto vicina alla parte bassa dell'immagine. Da tale informazione si deduce che, a meno di disturbi nell'immagine, il robot è molto vicino al marker , in un raggio di circa 20 cm, e quindi che si trova in una posizione tale da poter rilevare la linea gialla. Lo stato successivo è perciò `FOLLOW_LINE`.

Action coinvolte:

- `ArActionStallRecover` (priorità 100):

  Action fornita dalla suite Aria, che permette al robot di uscire da situazioni di movimento impossibilitato, rilevato grazie allo stallo di uno dei due motori, o di entrambi.
- `ArActionAvoidFront` (priorità 50):

  Action fornita dalla suite Aria, che rileva ostacoli frontali grazie all'uso dei sonar, e che impone al robot di evitare tali ostacoli modificando il proprio orientamento in maniera casuale.
- `ArActionAvoidSide` (priorità 40):

  Action fornita dalla suite Aria, che rileva ostacoli laterali grazie all'uso dei sonar, e che impone al robot di evitare tali ostacoli modificando il proprio orientamento nella direzione opposta al lato in cui è stato rilevato l'ostacolo.
- `ActionFollowBlob` (priorità 30):

  Action creata appositamente per seguire il marker verde, che modifica l'orientamento del robot in modo da mantenerlo orizzontalmente al centro dell'immagine. Questa Action sarà descritta nei dettagli nel prossimo paragrafo.
- `ArActionCostantVelocity` (priorità 20):

  Action fornita dalla suite Aria che impone al robot una velocità traslazionale costante, senza imporre alcun orientamento, che viene scelto dal resolver in base alle altre azioni.

3.3.2.2 Stato FIND_LINE:

Lo stato `FIND_LINE` è lo stato di transizione tra la ricerca del marker verde e la ricerca della linea. Questa funzione di intermediazione è particolarmente critica: la videocamera ha percepito il marker verde in prossimità della parte bassa dell'immagine, quindi probabilmente il robot si trova nelle vicinanze del marker stesso, ma non è possibile stabilire a priori la sua posizione relativamente alla linea, la quale potrebbe risultare totalmente fuori dal campo visivo. Per questo motivo è necessaria un'azione che permetta di stabilire in maniera robusta se il robot è realmente nella postazione desiderata. In questo stato, quindi, il robot mantiene la propria posizione, e cerca la linea gialla girando su se stesso, finché la telecamera non gli fornisce la coordinata della linea stessa, permettendogli di passare nello stato `FOLLOW_LINE`, oppure fino a quando, dopo aver compiuto circa un giro completo, scade un timer, che impone il ritorno allo stato `FOLLOW_BLOB`, nell'ipotesi che si siano verificati disturbi nella visione.

Action coinvolte:

- `ArActionStallRecover` (priorità 100):

  Action fornita dalla suite Aria che permette al robot di uscire da situazioni di movimento impossibilitato, rilevato grazie allo stallo di uno dei due motori, o di entrambi.
- `ActionFindLine` (priorità 50):

  Action implementata dal gruppo di lavoro che imposta una velocità di avanzamento nulla, ed una velocità di rotazione costante, in modo da far ruotare il robot su se stesso. La rotazione si interrompe quando viene vista la linea gialla.

3.3.2.3 Stato FOLLOW_LINE:

Lo stato `FOLLOW_LINE` è, similmente allo stato `FOLLOW_BLOB`, una composizione di comportamenti elementari, che nel loro insieme impongono al robot di seguire la linea gialla, mantenendola al centro della visuale, e di uscire dal programma non appena si è verificato uno stallo su entrambi i motori. Si tratta di uno stato molto critico dal punto di vista della sicurezza del robot, in quanto sono disattivati i sensori di prossimità, e non vi è alcuna azione di svincolo da situazioni di stallo. Inoltre, come descritto nel primo capitolo, mentre segue la linea, il robot si trova in una zona piuttosto angusta, con ostacoli vicini e di piccole dimensioni, difficilmente rilevabili. Se la telecamera non fornisce le coordinate della linea, il robot torna nello stato `FIND_LINE`, e inizia a girare su se stesso. Questo caso è piuttosto pericoloso, poiché il robot potrebbe non avere spazio a sufficienza. Per questo motivo sono stati aggiunti

degli accorgimenti, che verranno dettagliati in seguito, che permettono di sopperire in maniera più sicura agli errori del sistema di visione. Le condizioni di uscita da questo stato sono due: se la linea viene persa per un tempo superiore a un intervallo prestabilito il robot torna allo stato `FIND_LINE`; se invece viene rilevato lo stallo di entrambi i motori, mentre la linea è ancora visibile e centrata, si assume che il robot sia riuscito ad ancorarsi alla stazione di ricarica, e quindi il programma termina.

Action coinvolte:

- `ActionDetectStall` (priorità 100):

  Action realizzata dal gruppo di lavoro che rileva lo stallo di entrambi i motori e impone una transizione verso lo stato finale, e quindi il termine del programma.
- `ActionFollowLine` (priorità 50):

  Action realizzata dal gruppo di lavoro, che imposta l'orientamento del robot in modo da mantenere la coordinata orizzontale della linea al centro dell'immagine
- `ArActionCostantVelocity` (priorità 25):

  Action fornita dalla suite Aria che impone al robot una velocità traslazionale costante, senza imporre alcun orientamento, che viene scelto dal resolver in base alle altre azioni.

3.4. Le Action realizzate

Come si è visto, alcune delle azioni usate erano già implementate nelle librerie Aria; altre, invece, sono state implementate "ex novo". Tali azioni consistono in classi derivate dalla classe `ArAction`; ogni classe è definita in un file header ed implementata nel relativo file di sorgente C++. Di seguito vengono descritte in modo più dettagliato le azioni implementate "ex novo" per ciascuno stato del programma.

3.4.1. Azione ActionFollowBlob

Lo stato `FOLLOW_BLOB` distingue, al suo interno, tre sottostati: il sottostato in cui il robot non vede il marker verde, quello in cui lo vede e quello in cui il marker verde non è visibile, ma lo era fino a poco tempo prima. In figura 3.2 viene proposto lo statechart che rappresenta le transizioni fra i suddetti sottostati:

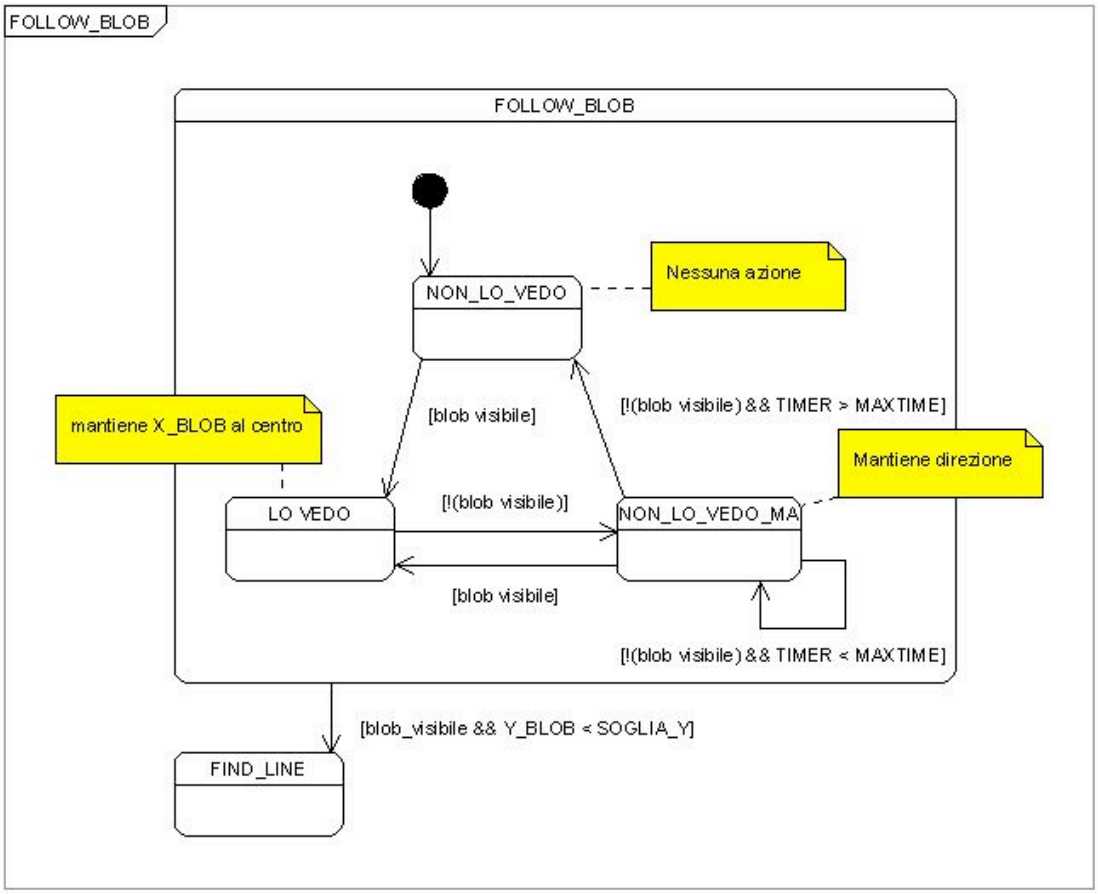

Figura 3.2 Dettaglio sullo stato FOLLOW_BLOB

Inizialmente, quando il marker verde non compare nell'immagine ripresa dalla videocamera, ci si trova nel sottostato `NON_LO_VEDO`; in questo caso l'azione `ActionFollowBlob` non imposta alcun comando e il comportamento complessivo del robot è determinato dalle altre azioni `ArActionStallRecover`, `ArActionAvoidSide`, `ArActionAvoidFront` e `ArActionConstantVelocity`, già descritte nel paragrafo precedente. L'effetto complessivo è molto simile a quello del programma `wander` presente tra gli esempi delle librerie Aria: il robot va sempre dritto fin quando non incontra un ostacolo, non c'è uno stallo o non vede il marker .

Se il marker compare nel campo visivo della telecamera, si passa allo stato interno `LO_VEDO` e l'azione `ActionFollowBlob` imposta dei comandi `setDeltaHeading` in modo tale da portare la coordinata x del baricentro del blob verde al centro dell'immagine vista: l'obiettivo è quello di portare il valore di x il più possibile vicino a 0, dal momento che la coordinata x vale zero nel caso in cui sia esattamente centrale.

In questo caso, dal metodo `fire()` dell'azione `ActionFollowBlob` viene richiamato il metodo `lo_vedo();` all'interno di questo metodo si imposta il comando `setDeltaHeading` nel modo seguente:

- se la coordinata x compare in una posizione abbastanza centrale, ovvero se il valore assoluto di x è minore della costante `SOGLIA_MOVIMENTO_X`, si imposta un angolo di rotazione pari a zero: in questo caso, infatti, il blob verde è centrato nel campo visivo del robot, quindi non è necessario alcun movimento rotazionale;
- se la coordinata x ha valore assoluto maggiore di `SOGLIA_MOVIMENTO_X`, ma minore di `SOGLIA_SATURAZIONE_X`, si imposta un angolo di rotazione proporzionale ad x: più precisamente, `deltaHeading=-x/FATTORE_SCALA`;

- infine, se la coordinata x ha valore assoluto maggiore di `SOGLIA_SATURAZIONE_X`, si imposta un angolo di rotazione costante, moltiplicato per il segno di x. Questo terzo caso è necessario ad evitare di compiere azioni troppo energiche nel caso in cui il baricentro compaia molto laterale.

In ognuno di questi casi, inoltre, viene impostata la variabile `myPosition`: questa variabile può assumere uno dei tre valori `CENTER`, `LEFT` e `RIGHT`, e serve a tener traccia della posizione del robot rispetto al marker . Nel primo caso `myPosition` assume un valore pari alla costante `CENTER`, mentre negli altri due casi assume un valore che dipende dalla posizione della coordinata x: se la x è a sinistra del centro visivo, si imposta `myPosition=RIGHT`; se la x è a destra del centro visivo, si imposta `myPosition=LEFT`.

La variabile `myPosition` viene letta nel caso in cui il marker non è visibile, ma lo era stato fino a poco tempo prima. In questo caso, dal metodo `fire()` viene richiamato il metodo `non_lo_vedo_ma()`, passando di fatto al sottostato omonimo `NON_LO_VEDO_MA`. All'interno del metodo viene impostato un comando `setDeltaHeading` che dipende unicamente dal valore di `myPosition`: se `myPosition` vale `CENTER`, si imposta un angolo di rotazione pari a 0; se `myPosition` vale `LEFT`, si imposta un angolo di rotazione negativo costante; se `myPosition` vale `RIGHT`, invece, si imposta un angolo di rotazione positivo, anch'esso costante. L'obiettivo del metodo è quello di "filtrare" eventuali errori temporanei nel sistema di visione del robot: se, per un breve periodo di tempo, la telecamera fornisce dati sbagliati (per esempio a causa di cambiamenti di luminosità, che fanno perdere l'immagine per qualche momento), il robot continua a muoversi nella stessa direzione che stava seguendo prima. Se però il marker non viene visto per troppo tempo, si ritorna al sottostato iniziale `NON_LO_VEDO`, ovvero quello in cui l'azione `ActionFollowBlob` non imposta alcun comando. L'intervallo di tempo in cui si rimane nello stato `NON_LO_VEDO_MA` senza rivedere il marker verde è pari a un secondo.

Come è già stato accennato precedentemente, la transizione dallo stato `FOLLOW_BLOB` allo stato `FIND_LINE` avviene quando la coordinata y del baricentro del blob verde si trova nella parte bassa del campo visivo, ovvero quando y è maggiore o uguale a 215 (ricordiamo che la y può assumere valori compresi tra 0 e 239). In questo caso, le velocità di avanzamento e di rotazione del robot vengono impostate a zero, e si imposta la variabile `global_state` al valore `STATE_FIND_LINE`; è necessario anche impostare la variabile booleana `state_changed` a `true`, in modo che il ciclo del metodo `main()` esegua correttamente la transizione di stato.

3.4.2. Azione ActionFindLine

Lo stato `FIND_LINE` è più semplice di `FOLLOW_BLOB`: in questo caso si hanno due sole azioni attive, la `ActionFindLine e la ArActionStallRecover`. Lo scopo della prima azione consiste nel far ruotare il robot su se stesso, finché non viene vista la linea gialla o finché non passa troppo tempo.

Nel metodo `fire()`, viene eseguito un controllo sulla coordinata x: se il valore di x è compreso entro una certa soglia (`SOGLIA_LINEA_TROVATA`), si passa allo stato `FOLLOW_LINE`; se il valore di x, invece, è fuori soglia, viene impostata una velocità di rotazione in senso orario pari a una costante moltiplicata per la variabile `myHeading`.

Infine, se è trascorso troppo tempo dall'inizio della rotazione del robot, si imposta la variabile `global_state` a `STATE_FOLLOW_BLOB`. Il tempo in cui il robot può rimanere nello stato `FIND_LINE` è sufficiente per permettergli di compiere un intero giro su se stesso, e quindi di trovare la linea gialla, oppure di concludere che non si trova nei pressi della linea.

3.4.3. Azione ActionFollowLine

Lo stato `FOLLOW_LINE` è abbastanza simile a `FOLLOW_BLOB`. Qui si distinguono due sottostati: lo stato in cui il robot vede la linea e quello in cui la linea non è visibile ma lo era stata fino a poco tempo prima. Nel caso in cui la linea non sia vista da tanto tempo è previsto il ritorno allo stato `FIND_LINE`. In figura 3.3 viene proposto lo statechart che rappresenta le transizioni fra i suddetti sottostati:

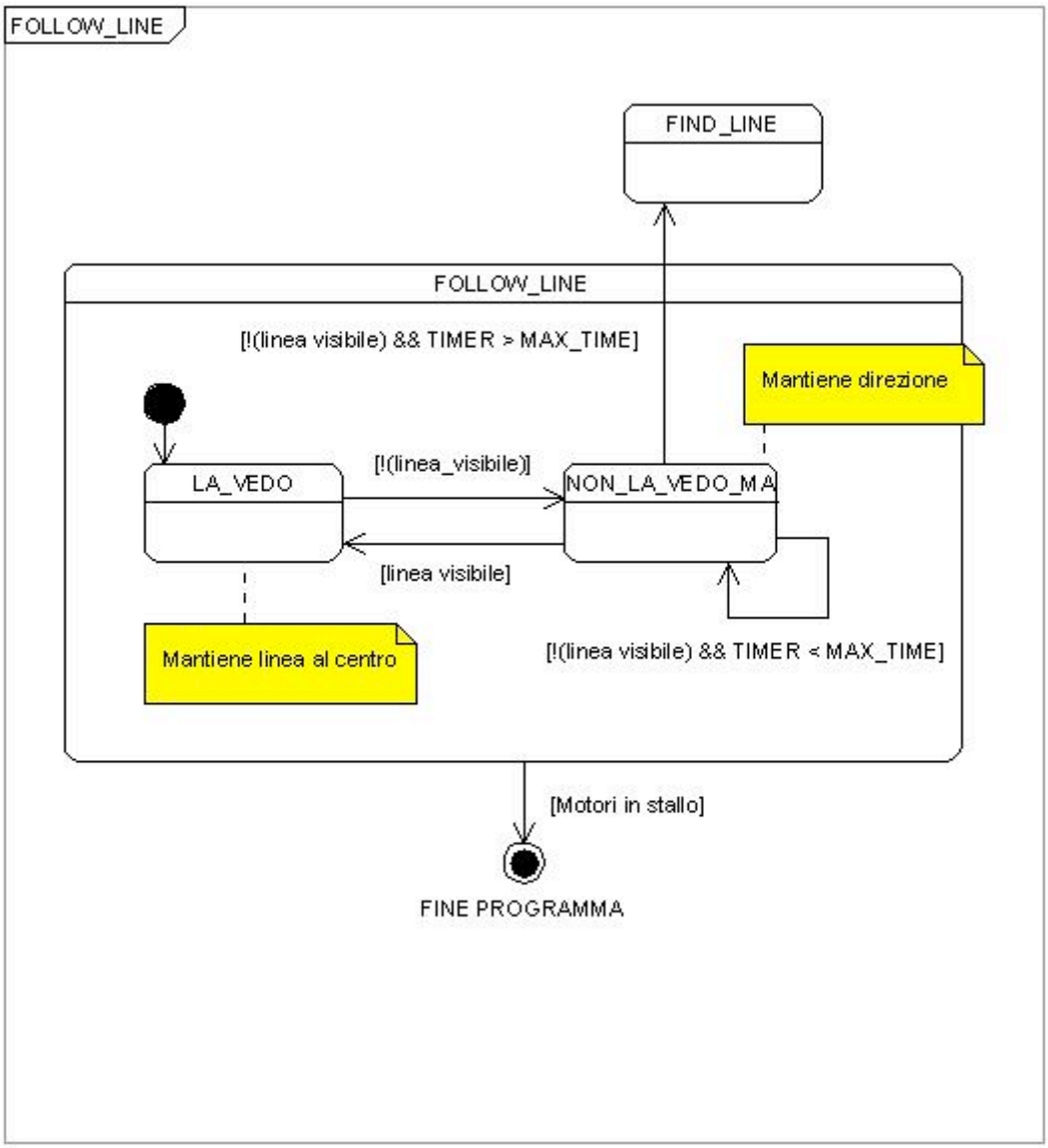

Figura 3.3 Dettaglio sullo stato FollowLine

Il comportamento complessivo è determinato dalla combinazione delle azioni `ActionFollowLine` (il cui obiettivo è di mantenere la linea al centro del campo visivo), `ArActionConstantVelocity` (che fa andare il robot in direzione rettilinea a velocità costante) e `ActionDetectStall` (che serve a capire quando il robot è arrivato alla stazione).

Nel caso in cui la linea sia all'interno del campo visivo (sottostato `LA_VEDO`), il metodo `fire()` dell'azione `ActionFollowLine` richiama il metodo `la_vedo()`. In questo metodo si compie un'azione che dipende dal valore della coordinata x: se il valore di x è, in valore assoluto, minore del valore di soglia `SOGLIA_MOVIMENTO_X`, viene impostato un `deltaHeading` pari a 0; se la x è superiore a `SOGLIA_MOVIMENTO_X`, ma minore di `SOGLIA_SATURAZIONE_X`, il valore di `deltaHeading` è proporzionale al valore di x; infine, se la x è superiore a `SOGLIA_SATURAZIONE_X`, il valore di deltaHeading è una costante moltiplicata per il segno di x. Anche qui, come in `ActionFollowBlob`, viene impostata la variabile `myPosition`, che serve nel metodo `non_la_vedo_ma()`.

Il metodo `non_la_vedo_ma()` viene richiamato nel caso in cui la linea non è più visibile, ma lo era stata fino a qualche istante prima, permettendo quindi di passare al sottostato `NON_LA_VEDO_MA`. Più precisamente, tale metodo viene invocato se la linea non è visibile, ma lo era stata fino a un tempo inferiore a due secondi prima. All'interno di `non_la_vedo_ma()` viene impostato un `deltaHeading` che dipende dal valore della variabile `myPosition`: se `myPosition` vale `CENTER`, si imposta un angolo di rotazione pari a 0; se `myPosition` vale `LEFT`, si imposta un angolo di rotazione negativo costante; analogamente, se `myPosition` vale `RIGHT`, si imposta un angolo di rotazione costante ma positivo. L'obiettivo del metodo è, analogamente a quello di ActionFollowBlob, fornire robustezza nei confronti di eventuali errori temporanei nel sistema di visione, permettendo al robot di continuare a muoversi nella direzione che stava seguendo prima. Tuttavia, in questo caso, il periodo di attesa di un'eventuale nuova percezione della linea è stato raddoppiato rispetto al caso del rettangolo verde, vista la criticità del compito e le maggiori difficoltà di riconoscimento della linea (spesso a causa di cambiamenti di luminosità).

Allo scadere dei due secondi, il metodo `fire()` imposta la variabile `global_state` al valore `STATE_FIND_LINE`, causando quindi il cambiamento di stato verso `FIND_LINE`.

3.4.4. Azione ActionDetectStall

Nel metodo `fire()` dell'azione `ActionDetectStall` si controlla se i motori sono in stallo o meno: se è stato rilevato uno stallo in entrambi i motori, il robot viene fermato (si impostano le velocità di avanzamento e di rotazione pari a 0) e la variabile `global_state` viene impostata a `QUARTO_STATO`: in questo modo si determina l'uscita dal ciclo `while` del `main()` e la terminazione del programma.

4. Modalità operative

L'applicativo che permette il parcheggio nella stazione di ricarica di Speedy può essere fornito in due modi differenti. Si segnala immediatamente che in entrambi i casi è necessario che sul calcolatore che si utilizza siano installate le librerie Aria e le librerie VisLib.

I due modi in cui l'applicazione può essere fornita sono:

- Tramite due file eseguibili (per sistema operativo Linux)
- Tramite l'insieme dei sorgenti dell'applicativo

Il caso più semplice è sicuramente il primo, in cui l'utente non deve di fatto fare alcuna operazione di installazione. I due file eseguibili possono essere lanciati direttamente dalla console di comando e sono:

- `main` (all'interno della sottodirectory `client`)
- `visione` (all'interno della sottodirectory `visione`)

Essi corrispondono ai due processi client e server che comunicano tramite socket per la soluzione dell'intero problema. Il file `visione` è l'eseguibile del processo server che appena è lanciato rimane in attesa di richieste di connessione (ne accetta al massimo una), mentre il file `main` è l'eseguibile del processo client che si connette prima al robot e poi al processo server.

Per come è costruito, il client cerca di connettersi al server non appena ha stabilito la connessione con il robot: in caso di fallimento in questa fase, il client termina l'esecuzione. È quindi necessario avviare prima il processo server attraverso il comando "`./visione/visione`", ed immediatamente dopo il processo client attraverso il comando "`./client/main`.

Nel caso in cui vengano forniti i sorgenti del progetto i due file eseguibili non sono immediatamente disponibili. In questo caso è necessaria una compilazione preliminare, per i cui dettagli si rimanda al paragrafo "Modalità di installazione".

4.1. Componenti necessari

Il progetto è stato realizzato e testato completamente su un sistema operativo di tipo Linux, e quindi è richiesta una distribuzione di tale sistema per eseguire l'applicativo.

Come già accennato in precedenza è inoltre necessario avere installate anche:

- la suite Aria
- la libreria VisLib (versione 1.8)

Il motivo di questa necessità risiede nel fatto che i file eseguibili richiedono di poter accedere ad alcuni shared object (file con estensione ".so") che vengono linkati dinamicamente e che compongono le librerie suddette.

Si assume che le librerie Aria e VisLib siano installate rispettivamente all'interno della directory "/usr/local/Aria/" e della directory "/usr/local/vislib-1.8/".

4.2. Modalità di installazione

L'applicativo che permette il parcheggio del robot Speedy viene fornito anche tramite l'insieme dei suoi file sorgenti, scritti nei linguaggi C e C++, assumendo che l'utilizzatore abbia a disposizione un sistema operativo di tipo Linux, dove la suite Aria funziona in modo migliore rispetto ad altri sistemi operativi. Questo modo di distribuire applicativi è abbastanza comune in ambiente Linux e più in generale nel mondo dell' "open source", dove la condivisione dei sorgenti rappresenta quasi un dogma. In questo caso l'intento è quello di permettere ad un utilizzatore interessato agli aspetti implementativi di poter analizzare il codice di questo elaborato, al fine di poterlo migliorare risolvendo magari problemi non ancora emersi o ottimizzando alcune porzioni di codice. Tuttavia, questo significa che, prima di iniziare l'esecuzione del progetto, è necessario compilare i file sorgenti. Questa operazione, apparentemente complicata, è in realtà estremamente semplice per l'utente, che non deve far altro che digitare il comando "make" all'interno della directory in cui sono memorizzati i sorgenti. Più precisamente, i file sorgenti sono raggruppati in due distinte directory: la prima, denominata `client`, contiene la parte relativa alla gestione del movimento (il processo client), mentre la seconda, denominata `visione`, contiene i sorgenti relativi al modulo omonimo. L'utente deve invocare due volte il comando make: la prima all'interno della directory `client` per compilare la parte relativa al movimento, la seconda all'interno della directory `visione`, per compilare la parte relativa alla gestione della videocamera. In ciascuna delle suddette directory è presente un apposito makefile costruito per risolvere tutti i riferimenti fra i vari moduli che devono essere compilati e linkati assieme. Nel caso in cui l'utente diventasse più esperto e conoscesse la struttura del progetto e dei makefile, il comando make permette di specificare anche quali singoli file compilare.

Dopo questa operazione, all'interno delle due directory vengono creati i due file eseguibili `main` e `visione` che possono essere lanciati dalla console di comando.

4.3. Avvertenze

Eventuali condizioni di luce sfavorevoli (luce spenta o forte illuminazione) potrebbero rendere difficile il corretto riconoscimento di rettangolo verde e linea gialla. Per ottenere i migliori risultati si consiglia di eseguire le dimostrazioni all'interno dell'ARL con le luci appese al soffitto accese ed eventualmente accendendo anche il faretto a incandescenza posto sull'armadio sopra la docking station.

Si consiglia infine di non variare l'illuminazione durante l'esecuzione del programma, poiché durante il transitorio della regolazione del guadagno della videocamera potrebbero verificarsi problemi di riconoscimento (soprattutto della linea gialla).

5. Conclusioni e sviluppi futuri

L'obiettivo dell'elaborato è stato raggiunto: il modulo di visione sviluppato opera bene in condizioni di luce normale e il modulo di movimento gestisce il veicolo secondo quanto specificato. In altre parole, Speedy arriva correttamente alla sua stazione di ricarica. Tuttavia, nonostante i numerosi accorgimenti adottati sia nella parte dedicata alla visione che in quella dedicata al movimento, in presenza di illuminazioni eccessive o particolari (ad esempio luci colorate) il programma può dare dei problemi. La mancanza di un controllo sofisticato sulla morfologia può talvolta far sì che oggetti lontani vengano scambiati per il marker verde. Tali effetti hanno comunque un impatto marginale poiché quando il robot si avvicina a marker fasulli spesso tali marker non vengono più riconosciuti come tali dato che, avvicinandosi, cambiano il loro colore (ad esempio mattonelle illuminate con riflessi che da lontano sembravano verdi).

Gli sviluppi futuri per la parte di movimentazione sono collocabili principalmente nella dotazione di una conoscenza dell'ambiente per il robot, mettendogli a disposizione una mappa che contenga la posizione di alcuni landmark all'interno del laboratorio. In questo modo si possono sopperire eventuali errori di visione identificando rettangolo e linea all'interno della mappa e si può eliminare l'attuale casualità della traiettoria seguita durante la fase iniziale di ricerca del marker: avendo la capacità di capire dove si trova all'interno del laboratorio, infatti, il robot potrebbe immediatamente portarsi nelle vicinanze della stazione di ricarica ed evitare di vagare in attesa di riconoscere il marker.

Bibliografia

La bibliografia deve essere indicata seguendo uno standard ben preciso. Lo stile da usare è lo stile "Biblio", che numera automaticamente i riferimenti.

[1] ActivMedia Robotics.: "CMPE 300 ARIA LAB MANUAL", 2002.

[2] Gini, G., Caglioti, V.: "Robotica", Zanichelli, 2003.

Indice