Esplorazione multirobot con agente cieco

B. Lain, L. Lovato, M. Zorzi
Università degli studi di Padova
Dipartimento di Ingegneria dell'Informazione
Padova, Italy, 35100

16 Febbraio 2009

Abstract

In questo lavoro si è implementato sperimentalmente un problema di navigazione autonoma di squadre di veicoli in ambiente sconosciuto. Nello specifico l'obiettivo prefissato è far raggiungere un luogo predeterminato ad un agente cieco. Per svolgere questo compito esso ha bisogno delle informazioni procurate da altri veicoli con cui è in comunicazione. In questo progetto si sono sviluppati algoritmi che ottimizzano l'esplorazione multi-robot attraverso la coordinazione degli agenti e che minimizzano mediamente il tempo necessario all'agente cieco per giungere alla meta. Si è in grado di gestire diversi observer e di operare anche nel caso che alcuni di loro siano impossibilitati ad avanzare. Si è sperimentato il tutto in laboratorio su un gruppo di robot e-Puck comandati da un calcolatore, collegato ad un sistema GPS simulato con una telecamera.

Introduzione

Gli ambiti di applicazione di squadre di robot mobili, tra effettivi e potenziali, comprendono lo svolgimento di compiti di diversa natura:

- missioni in ambienti ostili o diffiilmente raggiungbili da esseri umani (quali possono essere quello sottomarino o quello spaziale)
- compiti di assistenza e servizio.

Nell'esplorazione spaziale, ad esempio, si potrebbe utilizzare una squadra di robot tra i quali uno é equipaggiato di molti strumenti di misura sofisticati e delicati (atti per esempio a prelevare ed analizzare campioni di terreno, fare misurazioni di temperature, etc...). Per la riuscita della missione é indispensabile mantenere l'integritá di questo robot: quindi é essenziale che nei suoi spostamenti non corra rischi. Per tale motivo parallelamente vengono utilizzati altri robot con compiti di supporto al primo, principalmente l'accertamento della sicurezza dei luoghi in cui il primo robot si debba spostare. In tale contesto l'esplorazione degli ambienti al fine di condurre un robot principale, denominato actor, ha un ruolo importante: qui si inserisce questo lavoro. L'actor si suppone che sia un agente cieco, che deve portarsi quanto prima alla posizione denominata goal. Gli altri robot, denominati observer, devono esplorare l'ambiente senza sapere dov'é posizionato questo goal. L'elaborato si propone di sviluppare e mettere a confronto tre algoritmi di esplorazione e di elaborare, per lo specifico problema, un algoritmo di cammino minimo basato sull'algoritmo di Dijkstra. Lo scopo é quello di applicare tali algoritmi su un sistema reale di una squadra di robot presso il laboratorio Navlab.

Stato dell'arte

Una delle maggiori applicazioni della robotica mobile é creare modelli di ambienti utilizzando i sensori che equipaggiano i robot, ad esempio sonar, laser, sensori a infrarossi, telecamera. Questo processo é chiamato mapping. La localizzazione, ovvero la conoscenza della posizione del robot, é necessaria per molte strategie di mapping. Uno dei campi di ricerca attuali consiste nella localizzazione e nel mapping in simultanea (SLAM). Il robot si muove in un ambiente sconosciuto e deve contemporaneamente mapparlo e localizzarsi all'interno della mappa attraverso i sensori che ha a disposizione.

I piú comuni approcci al mapping sono quelli *grid-based* o *metric mapping* e *topological mapping*. Recentemente é stato sviluppato un metodo ibrido che combina questi due approcci [**?**]. Il *metric mapping* si basa sulle misure dello spazio mappato. Il path planning nelle mappe metriche include, solitamente, un certo numero di punti di via in specifiche posizioni, quali possono essere i punti in cui il robot deve fare una svolta. Questi punti di via sono connessi da segmenti in linea retta. Un metodo largamente usato per generare una metric-map é ricoprire l'ambiente con una griglia equispaziata. Ad ogni cella della griglia sono associati uno o piú valori che rappresentano la presenza o meno di un ostacolo. In certi casi il valore associato alla cella indica la probabilitá che questa sia occupata (solitamente indicato con $Pocc_{x,y}$). Un vantaggio di questo approccio é che questa mappa puó essere riutilizzata se cambia l'ambiente. Inoltre la relazione tra la mappa e la posizione del robot é immediata. Lo svantaggio é che per ottenere una ragionevole accuratezza della mappa é necessario usare delle celle di dimensione molto piccola, rendendo il path planning oneroso in termini computazionali e di memorizzazione.

Il *topological mapping* non si basa su precise misure, ma piuttosto su dei punti di riferimento(*landmark*) presenti nell'ambiente da esplorare e di rilevanza per la mappa (es: porte, intersezioni di corridoi...). Un esempio di istruzione con questa mappa é: "muoviti lungo il corridoio finchè non trovi un'intersezione di corridoi, gira a destra e poi a sinistra attraverso la prima porta". In una mappa topologica é essenziale conoscere la locazione dei landmark ma non la distanza tra di loro. Tipicamente la topological map é rappresentata da un grafo in cui ogni nodo é un landmark. Gli archi del grafo indicano se c'é una via che collega i landmark. Queste mappe sono molto compatte rispetto alle corrispondenti grid-based map. In [**?**] fornisce un esempio della trasformazione di una grid-based map in una topological map: la mappa originale ha 27000 celle, rispetto a 67 nodi della seconda. Un altro vantaggio é che il robot puó utilizzare l'informazione del sensore riguardo all'ambiente in maniera diretta, mentre il grid-based deve convertire l'informazione del sensore in maniera rappresentabile nella griglia. Un aspetto negativo é che i sensori possono perdere un landmark o confondere due landmark (ad esempio confondere due porte identiche).

In questo progetto si é deciso di mappare l'ambiente secondo l'approccio basato sulla grid-map: si é stabilito di dividere l'ambiente in quadrati di 10 cm di lato. Di conseguenza, avendo a disposizione una pedana di dimensione $2m \times 2.7m$ si sono ottenute 540 celle.

La navigazione é un processo nel quale si determina un percorso o una traiettoria per portare un veicolo dalla sua posizione iniziale ad un goal. La ricerca del miglior percorso dallo start al goal é un problema di ottimizzazione e richiede la selezione di un criterio di ottimizzazione, che puó essere basato sulla lunghezza del percorso o sul tempo di arrivo al goal. In questo progetto il path planning é stato implementato secondo i criteri dell'algoritmo di Dijkstra.

Si consideri un robot situato in un punto dell'ambiente nel quale si puó muovere e tre possibili tipi di scenario:

- Il goal é conosciuto. In queste condizioni il robot puó muoversi in direzione del goal sfruttando uno strumento di visione; tuttavia non sa la sua posizione. Quando incontra un ostacolo decide in maniera casuale se svoltare a destra o a sinistra. Aggira quindi l'ostacolo mantenendosi a distanza fissa da esso. Quando la direzione verso il goal

torna libera, il robot cambia il suo task e torna a muoversi in direzione del goal.

- Il goal e gli ostacoli non sono visibili. In questo caso si conosce la posizione del robot disponibile tramite dispositivo GPS, grazie al quale puó percorrere delle distanze predefinite. Se il GPS non é disponibile il robot puó usare gli encoder delle ruote. Questo metodo é utile solo per piccole distanze e inoltre richiede che il robot conosca la sua posizione e orientazione iniziale nella mappa.

- Il goal non é visibile dallo start, ma vi sono nell'ambiente dei punti di riferimento (visibili dal robot) dai quali il robot puó vedere il goal. In questo tipo di situazione il robot puó navigare interamente usando la visione, muovendosi dal punto iniziale a quello finale passando dai punti di riferimento.

Il problema principale dell'esplorazione é di capire come muovere i robot in modo da minimizzare il tempo necessario per esplorare completamente un'ambiente. Sfortunatamente, se si considera l'ambiente come un grafo questo problema diventa di tipo NP-difficile.

Molto spesso l'utilizzo di piú robot porta molti vantaggi rispetto ad un solo robot:

- un task viene eseguito in maniera piú veloce (se i robot cooperano) rispetto ad un singolo robot

- essi possono localizzare la loro posizione in maniera molto efficiente se si scambiano informazioni riguardo la loro posizione

- usando piú robot si introduce della ridondanza e quindi l'intero sistema puó diventare piú efficiente in termini di affidabilitá rispetto ad un'unico robot.

In letteratura il problema dell'esplorazione di un singolo robot é giá stato trattato con dettaglio, tuttavia nel caso multi-robot gli approcci di esplorazione sono ancora pochi. In [**?**],[**?**] viene affrontato il problema di ridurre l'errore dell'odometro durante l'esplorazione nel seguente modo: quando un robot si muove gli altri rimangono fermi e osservano lo spostamernto del primo. Questa tecnica tuttavia non permette di distribuire i robot sull'ambiente in quanto devono rimanere vicini affinché ciascun robot abbia nel suo raggio di visibilitá tutti gli altri. Di conseguenza non si ha una significativa riduzione del tempo di esplorazione rispetto al caso di un singolo robot. In [**?**] viene presentato un'approccio decentralizzato per robot eterogenei; quando un robot trova un'apertura che porta ad una zona inesplorata che non puó raggiungere (a causa delle sue dimensione fisiche) si seleziona un'altro robot che puó terminare questo task. Yamauchi in [**?**] presenta un'esplorazione che consiste nel far muovere i robot in regioni libere che confinano con aree inesplorate; tuttavia in questo approccio non é presente coordinazione per la scelta di queste regioni per i singoli robot. In [**?**] viene affrontato il problema dell'esporazione collaborativa di un'ambiente sconosciuto utilizzando un sistema multi-robot. In questo caso i robot sono muniti di sensori che hanno un raggio di visibilitá che dipende dagli ostacoli (ovvero oltre un'ostacolo non riescono a vedere). Anche in questo caso l'esplorazione consiste nel far muovere i robot verso regioni libere che confinano con aree inesplorate, tuttavia nel far questo si tiene conto del raggio di visibilitá di essi in modo da evitare che piú robot esplorino le stesse aree. Per quanto rigurda l'esplorazione, in questo progetto si sono proposti tre algoritmi: uno di 'esplorazione bruta', uno 'collaborativo' (sviluppato a partire dalle idee di quest'ultimo articolo) e uno 'non collaborativo'. Per questi algoritmi si é ipotizzato che:

- ogni robot tenga traccia delle aree dell'ambiente che ha giá esplorato comunincando ció all'unitá centrale

- l'unitá centrale conosce la posizione di ogni robot durante l'esplorazione

- ogni observer é munito di radar e rileva in maniera deterministica l'attributo delle celle vicinanti[1]

[1] per maggiori dettagli sulle celle adiacenti e vicinanti vedere la sezione dedicata all'esplorazione multi-robot

- observer e actor possono solo muoversi nelle celle adiacenti
- gli observer non conoscono la posizione del goal, mentre l'actor sí
- inizialmente le celle vicinanti alla cella occupata dall'actor siano conosciute.

Esplorazione multi-robot

Formulazione del problema e definizioni

Ogni cella della grid-map é individuata da due indici $\langle k, j\rangle$ proporzionali alle coordinate $\langle x, y\rangle$ in metri del baricentro del corrispondente quadrato secondo la seguente relazione

$$x = (k - 0.5)lato_{cella}, \quad y = (j - 0.5)lato_{cella}.$$

Ognuna di queste celle é caratterizzata da un'attributo che ne descrive la propria natura:

- $cella\langle x, y\rangle = 1$ significa che in $\langle x, y\rangle$ c'e un'ostacolo
- $cella\langle x, y\rangle = 0$ significa che in $\langle x, y\rangle$ non c'e un'ostacolo
- $cella\langle x, y\rangle = -1$ significa che in $\langle x, y\rangle$ non si sa se c'e un'ostacolo o meno (ovvero la cella é inesplorata).

In seguito saranno utili i seguenti insiemi:

- insieme delle celle adiacendti di $\langle k, j\rangle$

$$S_A(\langle k, j\rangle) = \{\langle k_*, j_*\rangle : |k - k_*| = 1, j = j_* \vee k = k_*, |j - j_*| = 1\}$$

- insieme delle celle vicinanti di $\langle k, j\rangle$

$$S_V(\langle k, j\rangle) = \{\langle k_*, j_*\rangle : |k - k_*| \leq 1, |j - j_*| \leq 1\}$$

- insieme delle celle di frontiera

$$F = \{\langle k_*, j_*\rangle : cella\langle k_*, j_*\rangle = 0, \exists\langle \bar{k}, \bar{j}\rangle \epsilon S_V(\langle k_*, j_*\rangle) \text{ con } cella\langle \bar{k}, \bar{j}\rangle = -1\}$$

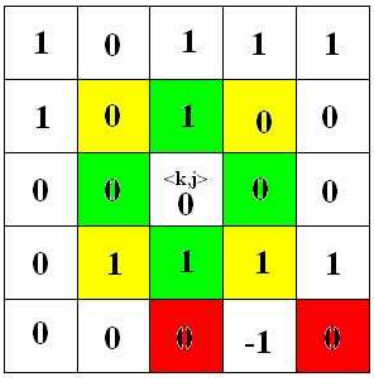


Figura 1: Celle adiacenti di $\langle k, j\rangle$(verdi), e vicinanti di $\langle k, j\rangle$(gialle,verdi) e celle di frontiera (rosse).

(si veda figura 1).

Infine si definiscono i seguenti parametri e funzioni per valutare le prestazioni degli algoritmi di esplorazione:

- N che corrisponde al numero di passi necessari per esplorare tutto l'ambiente
- $$\eta(n) = \frac{n_{expl}}{rc}100$$
  corrisponde alla percentuale di celle esplorate al passo n, dove r, c sono il numero di righe e colonne della grid-map e n_{expl} indica il numero di celle esplorate al passo n-esimo
- $$\gamma(n) = \begin{cases} 0 & n = 1 \\ \left[3 - \frac{\beta(n)}{3 \cdot \text{numero\_observer}}\right] 100 & n > 1 \end{cases}$$
  é la ridondanza percentuale di esplorazione degli observer al passo n dove $\beta(n)$ é il numero di celle scoperte dagli observer al passo n in particolare per l'observer i-esimo si ha $\beta_i(k) \leq 3$ ad eccezione del passo iniziale (si veda la figura 2 dove a) é il passo iniziale, b) e c) i passi successivi). Quindi piú la ridonadanza assume valori bassi e piú é efficiente l'esplorazione.

Algoritmo di esplorazione bruta (EB)

Questo algortimo (inventato dagli autori) si basa sulla costruzione di una mappa dei potenziali (P) che é una matrice di dimensione $r \times c$. Inizialmente essa si inzializza con tutti zeri. Quando gli observer rilevano in corrispondenza di una cella un'ostacolo, il suo attributo in P viene posto ad ∞; ogni volta che un observer attraversa una cella, viene a questa sommata una penalitá (sempre in P). La posizione successiva di un observer é la cella adiacente che ha l'attributo di valore piú basso in P. Perció la penalitá che si somma al passaggio di un'observer ha lo scopo di sfavorire il passaggio successivo degli observer in quanto la zona é gia stata esplorata.
Ora viene presentato il pseudo-codice dell'algoritmo

- per ogni observer OBS con posizione $\langle OBS \rangle$

$$tmp(OBS) = P(\langle OBS \rangle)$$

$$P(\langle OBS \rangle) = \infty$$

- per ogni OBS

$$tmp(OBS) = tmp(OBS) + penalità$$

$$\langle OBS \rangle_{new} = min_{\langle k,j \rangle \epsilon S_A(\langle OBS \rangle)} P(\langle k, j \rangle)$$

$$tmp(\langle OBS \rangle_{new}) = P(\langle OBS \rangle_{new})$$

$$P(\langle OBS \rangle_{new}) = \infty$$

- per ogni OBS

$$P(\langle OBS \rangle) = tmp(\langle OBS \rangle)$$

$$P(\langle OBS \rangle_{new}) = tmp(\langle OBS \rangle_{new})$$

In parole povere questo algoritmo preleva i valori in P delle celle occupate dagli observer e salva tali valori in delle variabili temporanee. Quindi il valore di queste celle in P viene posto uguale ad infinito per evitare che al passo successivo altri observer occupino queste celle (evitando cosí collisioni fra robot). Sempre per lo stesso motivo anche le posizioni successive degli observer vengono messe come ostacoli.

Algoritmo di esplorazione collaborativa (C)

L'approccio di esplorazione implementato consiste nel guidare gli observer a dei punti di target in modo da minimizzare il tempo di esplorazione. Questi punti di target sono scelti nell'insieme delle celle di frontiera F in quanto questo permette agli observer (una volta raggiunta una cella di frontiera) di acquisire informazioni su una zona inesplorata.
Questo algoritmo costruisce una mappa dell'ambiente (indicata con *mappa degli observer*) dalle informazioni fornite dagli observer e sulla base di questa associa una cella di frontiera ad un observer finché a tutti gli observer é associata una cella di frontiera. Questa associazione viene effettuata massimizzando la funzione $U - \alpha V$ (il significato di questi termini verrá fra poco illustrato). Terminata questa fase, per ogni observer si determina il cammino ottimale per raggiungere la corrispondente cella di frontiera; come gli observer raggiungono la prima cella del cammino, si riapplica l'algoritmo, giacché nel frattempo la conoscienza dell'ambiente potrebbe essere aumentata e potrebbero essere stati individuati dei cammini migliori o dei target più proficui.

Funzione costo V

Si definisce come funzione costo dalla cella $\langle l, m \rangle$ alla cella $\langle k, j \rangle$

$$V(\langle k, j \rangle, \langle l, m \rangle) = \frac{\|\langle k, j \rangle - \langle l, m \rangle\|^2}{r^2 + c^2}$$

dove $\langle k, j \rangle$ é la posizione (nella grid-map) di un'observer e $\langle l, m \rangle$ una cella di frontiera.
Quindi V é la distanza euclidea al quadrato tra $\langle k, j \rangle$ e $\langle l, m \rangle$ normalizzata ($V \leq 1$). Ovviamente va osservato che questa non é essattamente proporzionale al quadrato della lunghezza del cammino che l'observer deve effettuare per raggiungere la cella di frontiera in quanto:

- nell'ambiente sono presenti degli ostacoli
- i robot si possono muovere solo in maniera parallela agli assi coordinati[2]

[2]questa formulazione del problema considera le celle dis-

quindi V fornisce una stima grossolana proporzionale al quadrato della lunghezza del cammino tra l'observer e la cella di frontiera. Tuttavia il suo calcolo non é computazionalmente oneroso.

Funzione utilitá U

Si definisce come utilitá della cella di frontiera $\langle k, j\rangle$, e la si indica come $U(\langle k, j\rangle)$, il vantaggio dal punto di vista esplorativo ottenuto dal raggiungimento della cella da parte di un observer.
Questa funzione perció ha come dominio F. Al contrario dell'articolo (in cui si inizializza questa funzione ponendo $U(\cdot) \equiv 1$) si é deciso di inizializzare questa funzione nel seguente modo

$$U(\langle k, j\rangle) = \frac{|\{\langle k_*, j_*\rangle \epsilon S_V(\langle k, j\rangle), \langle k_*, j_*\rangle = -1\}|}{5}$$

in quanto le celle inesplorate vicinanti ad una cella di frontiera possono essere al massimo cinque; questo puó succedere quando la cella di frontiera non é una cella adiacente alla cella dell'observer (figura 2c), perció $U \leq 1$. Dopo aver effettuato una associazione tra una cella di frontiera $\langle k, j\rangle$ e un'observer OBS_i l'utilitá viene aggiornata nella seguente maniera:

$$U(\langle k_*, j_*\rangle) = U(\langle k_*, j_*\rangle) \frac{\| \langle k_*, j_*\rangle - \langle k, j\rangle \|^2}{r^2 + c^2} \quad \forall \langle k_*, j_*\rangle \epsilon F$$

ovvero alle celle di frontiera piú vicine a $\langle k, j\rangle$ si effettua una elevata riduzione di utilitá in quanto é molto probabile che

- OBS_i durante il tragitto per raggiungere $\langle k, j\rangle$ passi per queste celle
- per OBS_i dopo aver raggiunto $\langle k, j\rangle$ sia poco oneroso (dal punto di vista della lunghezza del cammino) raggiungere queste celle.

Infine si osserva che l'associazione observer-cella di frontiera avviene attraverso un compromesso tra

tanziate dalla **Manhattan distance**, la quale definisce la distanza fra due punti, appartenenti allo spazio euclideo con un fissato sistema di assi cartesiani, come la somma delle proiezioni sugli assi cartesiani dei segmenti che congiungono i due punti.

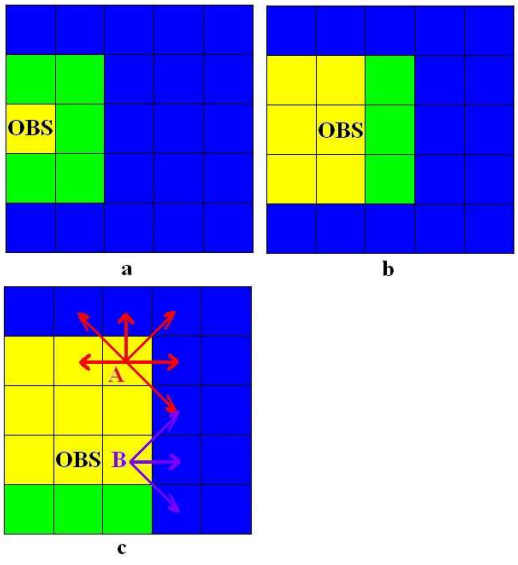


Figura 2: Esplorazione di un'observer (celle esplorate gialle, celle inesoplorate blu, celle appena esplorate verdi) dove $U(A) = \frac{5}{5}$ e $U(B) = \frac{3}{5}$

- condurre l'observer in celle di frontiera non troppo distanti (in quanto durante il cammino non é garantito che l'observer scopra zone inesplorate)
- far raggiungere all'observer celle di frontiera con utilitá alta

Il parametro α indica quanto é importante il primo obbiettivo rispetto al secondo.

Algoritmo

Ora viene presentato il pesudo-codice dell'algoritmo di esplorazione

- si inizializza la mappa degli observer eguagliandola alla mappa conosciuta dall'unitá centrale
- si impone che le celle occupate dai robot (observer e actor) e dal goal siano identificate come ostacoli nella mappa degli observer
- si determinano le celle di frontiera e si inizializza l'utilitá

- se non sono state trovate celle di frontiera:
  - flag_stop=true
  - le posizioni degli observer non variano
  - return

  altrimenti flag_stop=false
- per ogni observer OBS con posizione $\langle OBS \rangle$
  - per ogni cella di frontiera $\langle k, j \rangle$ si calcola $V(\langle k, j \rangle, \langle OBS \rangle)$
  - flag_target(OBS)=false
- while($\exists\, OBS$: flag_target(OBS)=false)
  - $(\langle k, j \rangle, OBS) = argmax_{\langle k_*, j_* \rangle, OBS_*} \{U(\langle k_*, j_* \rangle) - \alpha V(\langle k_*, j_* \rangle, \langle OBS \rangle)\}$
  - flag_target(OBS)=true
  - target(OBS)=$\langle k, j \rangle$
  - per ogni cella di frontiera $\langle k_*, j_* \rangle$ $U(\langle k_*, j_* \rangle) = U(\langle k_*, j_* \rangle) \frac{\|\langle k_*, j_* \rangle - \langle k, j \rangle\|^2}{r^2 + c^2}$
- per ogni observer OBS
  - si impone che le celle occupate dagli observer siano identificate come libere nella mappa degli observer
  - si applica Dijkstra[3] per trovare il percorso minimo da $\langle OBS \rangle$ a target(OBS) e si pone come cella successiva di OBS da occupare $\langle OBS \rangle_{new}$ la prima cella del percorso minimo trovato
  - si impone che le celle occupate dagli observer siano identificate come ostacoli nella mappa degli observer
  - se la cella $\langle OBS \rangle_{new}$ é un'ostacolo nella mappa degli observer si pone $\langle OBS \rangle_{new} = \langle OBS \rangle$
  - si impone che $\langle OBS \rangle_{new}$ sia identificata come ostacolo nella mappa degli observer

[3] in questa sezione si utilizza la quarta implementazione dell'algoritmo di Dijkstra: vedere sezione sul path planning per dettagli

Va osservato che inizialmente nella mappa degli observer le celle occupate dai robot vengono temporaneamente considerate come ostacoli per evitare scontri tra i robot.
La variabile binaria flag_stop indica se l'esplorazione é terminata o meno, flag_target(OBS) se all'observer OBS é stato giá assegnato un target.
Ad ogni iterazione del ciclo while viene individuato un'observer e una cella di frontiera che massimizzano la funzione $U - \alpha V$. I target trovati generalmente non sono delle celle adiacenti alle posizioni dei rispettivi observer: per tale motivo bisogna applicare l'algoritmo di Dijkstra per pianificare un percorso tra due celle. Per applicare tale algoritmo bisogna considerare le celle occupate dai robot come libere in quanto si suppone che da una cella occupata non si possa raggiungere nessuna altra cella. Successivamente le celle occupate dai robot vengono interpretate come ostacoli per evitare che al passso successivo altri observer occupino queste celle (per evitare scontri), per questo motivo, inoltre, una volta trovata la cella che l'observer dovrá raggiungere al passo successivo, essa viene interpretata come un'ostacolo.
Al termine dell'esplorazione tutti gli observer si fermano: quindi per evitare che uno di essi si fermi nel goal (e che quindi impedisca all'actor di svolgere la sua missione) inizialmente la cella associata al goal si interpreta come ostacolo.
Infine va osservato che l'algoritmo per l'assegnazione dei target viene effettuato ad ogni passo indipendentemente dal fatto che tutti i robot abbiano raggiunto il target precedente o meno.

Algoritmo di esplorazione senza l'approccio collaborativo (NC)

Questo differisce dal precedente esclusivamente per il fatto che non viene considerata l'utilitá nell'assegnazione dei target agli observer: é come se si inizializzasse l'utilitá con $U(\cdot) \equiv 0$.

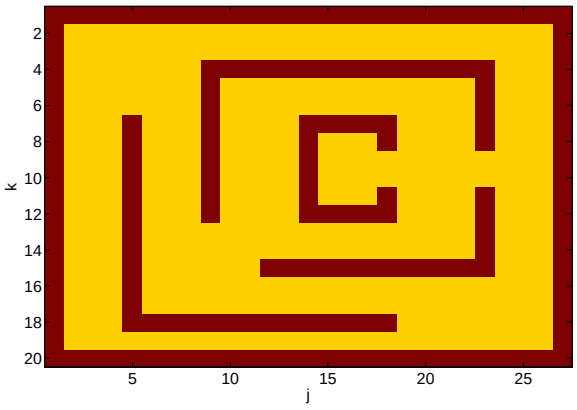


Figura 3: Mappa utilizzata per gli esperimenti.

Analisi delle prestazioni

La mappa con la quale si sono effettuati gli esperimenti é mostrata in figura 3. Ogni esperimento é composto da 20 prove in ciascuna delle quali cambiano le posizioni iniziali degli observer OBS_i

$$\langle OBS_i \rangle = \langle 1 + 2i + j_1, j_2 \rangle \;\; j_1 = 1, ..., 10, \;\; j_2 = 2, 26.$$

Per ciascun esperimento si calcola la media e la deviazione standard campionaria di N (μ_N, σ_N) e la media (campionaria) percentuale di celle esplorate al cinquantesimo passo ($E\eta(50)$). Si é deciso di considerare quest'ultimo parametro in quanto l'esplorazione totale non é lo scopo principale del progetto, ma piuttosto trovare un percorso al goal per l'actor nel piú breve tempo possibile (questo é tanto piú probabile quanto piú $\eta(50)$ é grande). I risultati trovati sono mostrati nella seguente tabella.

OBS.	ALG.	α	μ_N	σ_N	$E\eta(50)$
1	C	1	342	23	21
	C	10	307	17	25
	C	20	**272**	14	26
	C	30	273	15	26
	C	40	273	13	26
	C	50	273	13	26
	NC	-	293	**5**	22
	EB	-	433	23	**28**
2	C	1	182	14	46
	C	10	151	18	**51**
	C	20	151	14	**51**
	C	30	**145**	13	50
	C	40	148	**12**	50
	C	50	149	14	**51**
	NC	-	162	18	50
	EB	-	349	85	39
3	C	1	110	12	67
	C	10	109	13	71
	C	20	107	15	72
	C	30	110	11	**73**
	C	40	108	12	69
	C	50	**103**	15	69
	NC	-	114	**10**	54
	EB	-	195	49	64
4	C	1	84	14	79
	C	10	86	18	83
	C	20	**78**	10	84
	C	30	**78**	10	84
	C	40	85	17	**86**
	C	50	83	14	85
	NC	-	86	**5**	74
	EB	-	160	58	72

La prima cosa che si evince dalla tabella é che all'aumentare degli observer l'esplorazione é ovviamente piú rapida per tutti e tre gli algoritmi. Tuttavia si nota dalla colona degli $E\eta(50)$ che l'utilitá marginale dell'aggiunta di un observer é maggiore per C rispetto a NC: infatti il primo algoritmo, permettendo la coordinazione degli agenti, sa sfruttare al meglio l'aumento degli esploratori. L'algoritmo C presenta una diminuzone di μ_N all'aumentare di α nell'intervallo $[1, 30]$; in $[40, \infty^4]$, invece, μ_N aumenta. Si giustifica questo comportamento in quan-

[4] si osservi che NC equivale a C con $\alpha = \infty$

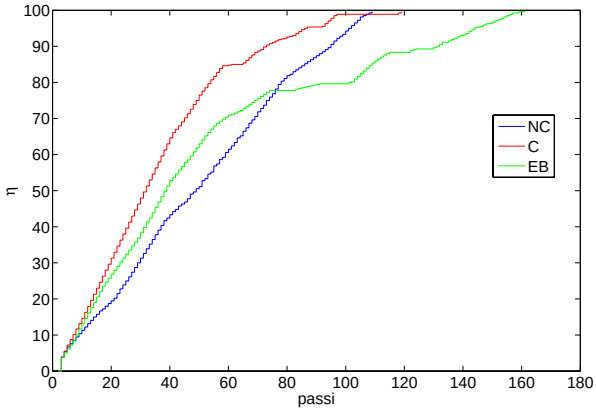


Figura 4: Percentuale di celle scoperte.

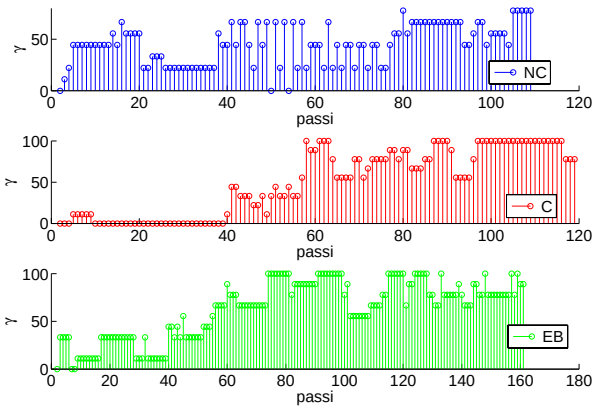


Figura 5: Ridondanza.

to l'ottimizzazione dell'esplorazione richiede una coordinazione di observer che non deve andare eccessivamente a scapito dell'esigenza di trovare target prossimi agli observer. Viaggi eccessivamente lunghi possono infatti rivelarsi inutili se nel bel mezzo del percorso un altro observer si aggiudicasse il target[5]. Per contro, σ_N di C assume valori piú alti di quelli di NC in quanto al variare delle posizioni iniziali degli observer, l'inizializzazione di U ad ogni ciclo puó cambiare in maniera significativa. Di conseguenza questo puó modificare in maniera rilevante l'esplorazione. L'algoritmo EB offre prestazioni peggiori degli altri algoritmi: l'esigenza di ripassare diverse volte su un gruppo di celle vicine prima di attribuire loro un potenziale sufficientemente alto da garantire l'uscita da tale zona, aggiunto alla non collaborativitá degli agenti, giustifica questo fatto.

Nelle figure 4, 5 vengono messi a confronto i tre algoritmi in una prova con tre observer ($\alpha = 30$ per C) Si osserva che inizialmente il comportamento collaborativo degli agenti rende l'esplorazione C piú proficua (ridondanza quasi nulla nei primi 40 passi e maggiore pendenza della curva η rispetto agli altri algoritmi). Tuttavia questo comportamento si rivela controproducente nelle ultime fasi dell'esplorazione: infatti i robot tendono a scambiarsi i target ad ogni passo, rallentando la conclusione dell'esplorazione. Di questo difetto non soffre l'algoritmo NC: infatti l'esplorazone termina prima. L'algoritmo EB inizialmente fornisce prestazioni intermedie rispetto agli altri due: tuttavia, quando si presentano gruppi di celle inesplorate isolate, l'algoritmo fatica a raggiungerle per quanto giá detto sopra. Per lo scopo di questo progetto l'algoritmo C é il piú indicato perché esplora rapidamente un'ampia zona rendendo piú probabile l'individuazione di un cammino verso al goal nei primi passi.

[5]si ricorda che i terget vengono riassegnati ad ogni passo

Path planning

Il problema del path planning é stato affrontato come un problema di ricerca di un cammino minimo in un grafo orientato: ogni cella della griglia é stata considerata come un nodo collegato da archi alle quattro celle adiacenti. La funzione *creaGrafo* é stata pensata per il path planning degli obsevers: essa riceve in ingresso una mappa e ne cataloga le celle dividendole in

1. celle di frontiera

2. celle libere non di frontiera (che chiameremo celle interne)

3. inesplorate o occupate da ostacoli

Dopo questa prima scansione controlla le adiacenze di ogni cella e per ogni celle adiacente libera pone un arco pesato fra le due celle. Quest'arco avrá peso 0.3 qualora la cella di destinazione sia di frontiera, 1 altrimenti. Questa distinzione é stata pensata per favorire il movimento degli observer lungo la frontiera delle celle inesplorate: infatti lo spostamento lungo tre celle frontiera risulta piú conveniente (ovvero a costo complessivo piú basso) di un solo passo in una cella non adiacente ad una cella inesplorata. Il path planning per gli observer non é quindi orientato a velocizzare lo spostamento dei robot da un punto 'a' ad un punto 'b' facendo scegliere loro la rotta piú breve, ma a far compiere loro dei tragitti che massimizzino l'esplorazione della zona. Per l'actor non viene fatta questa distinzione fra celle interne e di frontiera: infatti esso si deve occupare solo di arrivare quanto prima alla cella obbiettivo assegnatagli. La generazione del grafo da una mappa per l'actor si limita quindi a verificare se le celle sono esplorate e libere ed a porre peso unitario ai due archi orientati che uniscono tali celle adiacenti.
Come giá accennato, la ricerca del cammino minimo é stata implementata seguendo la logica dell'algoritmo di Dijkstra. Nonostante in generale la ricerca di un cammino minimo sia un problema NP-difficile, l'assunzione che i costi degli archi del grafo orientato siano tutti maggiori o uguali a zero rende il problema polinomiale. Una formulazione in pseudo-codice dell'algoritmo[6] é riportata di seguito.
In questo codice si assume che:

- **S** é l'insieme dei nodi del grafo che sono collegati al nodo di partenza da un cammino minimo,
- **s** é il nodo di partenza,
- **L[i]** indica il costo per raggiungere il nodo 'i' partendo da 's',

[6]per approfondimenti vedere [**?**]

Dijkstra
begin
$S := s$;
$L[s] := 0$;
$pred[s] := s$;
while $|S| \neq n$ *do*
begin
$(v, h) := argmin\{L[i] + c[i, j] : (i, j) \in \delta^+(S)\}$;
$L[h] := L[v] + c[v, h]$;
$pred[h] := v$;
$S := S \bigcup \{h\}$
end
end

- **pred[i]** tiene memoria del nodo precedente al nodo 'i' nel percorso da 's' a 'i', pemettendo una ricostruzione a ritroso del cammino
- **c[i,j]** indica il costo dell'arco che congiunge il nodo 'i' al nodo 'j'.
- $\delta^+(S)$ indica il taglio di S, ovvero l'insieme degli archi (i,j) che hanno come estremi $i \in S$, e $j \in V/S$, dove V indica l'insieme di tutti i nodi.

L'idea dell'algoritmo é di aggiungere ad ogni iterazione del ciclo while il nodo 'h' collegato a S dall'arco di costo minimo fra quelli uscenti da S. Fatto ció, si aggiorna il costo del cammino che porta al nodo 'h', ponendolo uguale a quello che raggiunge il suo predecessore piú il costo dell'arco che li unisce. L'algoritmo si ferma quando il numero dei nodi contenuti in S é pari al numero totale dei nodi. Se con 'm' denotiamo il numero degli archi del grafo, scandire tutti gli archi per trovarne uno a costo minimo ignorando quelli che non appartengono a $\delta^+(S)$, comporta una complessità $O(m)$. Reiterare questa procedura per n-1 archi (tanti sono necessari per avere un albero che raggiunge ogni nodo), dove 'n' é il numero dei nodi del grafo, comporta una complessitá $O(m * n)$
Prevedendo di usare massicciamente quest'algoritmo in questo progetto, si é cercato di pensare a qualche modifica che permettesse di velocizzarlo quanto piú possibile nell'applicazione in analisi.

Nella **prima versione** dell'algoritmo si inizializza il grafo ponendo per ogni cella 'j'

- pred[j]=start, dove start é la cella di partenza del cammino
- L[j]=c[start,j]. Se 'j' non fosse adiacente allo start, si pone $c[start, j] = \infty$

Tramite due cicli *for* annidati si esegue una scansione lungo le due coordinate della mappa per individuare la cella 'h' non ancora appartenente ad S con L minimo per aggiungerla ad S. Questa operazione richiede un tempo $O(n)$. Si verifica quindi per ogni cella 'j' adiacente ad 'h' non ancora entrata in S se $L[h] + c[h, j] < L[j]$: se questa condizione fosse verificata, si aggiornerebbe L[j] come $L[j] = L[h] + c[h, j]$ e si porrebbe $pred[j] = h$. Queste operazioni sono $O(1)$. Tutto ció viene ripetuto fino all'individuazione di un cammino minimo dalla cella 'start' a quella 'goal': quindi si può concludere che questa versione dell'algoritmo ha complessità $O(n^2)$.

Una **seconda versione** si differenzia dalla prima per l' utilizzo di una coda di prioritá implementata con una struttura ad heap[7]: ogni nodo dell'albero contiene informazioni sulle coordinate di una cella non ancora appartenente all'insieme S e sul costo del cammino per raggiungerla. Questo heap viene inizializzato con le celle libere adiacenti allo start. La cella a costo minimo si trova sempre nella radice dell'albero (in caso ci fossero piú celle di pari costo, ha la precedenza la cella inserita da piú tempo nell'heap) e da lí viene eliminata una volta entrata in S. Appena una nuova cella entra in S, le celle adiacenti non appartenenti ad S vengono aggiunte all'heap. Se il costo di una cella giá nella coda di prioritá dovesse variare, un'apposita funzione ne aggiorna il valore. L'utilizzo di una coda di prioritá permette di evitare la scansione tramite i cicli *for* delle celle, velocizzando l'algoritmo quando il numero delle celle della mappa é elevato. Infatti definita con $|Q|$ la dimensione della coda, questa struttura dati permette di effettuare in tempo $O(log|Q|)$ sia l'inserimento che l'eliminazione di una cella in Q. Quindi la complessitá totale dell'algoritmo risulta essere $O(n * log(n))$

Si vedrá ora un esempio del funzionamento dell'algoritmo per queste due prime versioni. Sia in figura 6 il rettangolo rosso un observer a cui sia stato richiesto di dirigersi nella casella identificata dalla X rossa. Le caselle gialle indicano celle libere, quelle marroni ostacoli, quelle blu celle inesplorate e la casella verde rappresenta un altro robot.

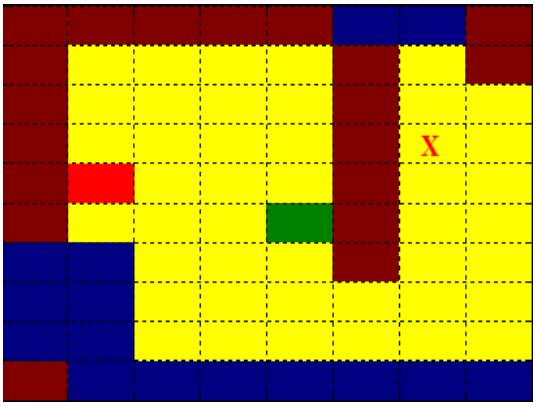


Figura 6: esempio

Subito si analizzano i costi delle celle libere adiacenti alla cella dove si trova il robot rosso. La cella di costo 0.3 (di costo inferiore alle altre due perché adiacente ad una cella inesplorata) viene aggiunta all'insieme S, al momento contenente la sola cella start. Nelle iterazioni successive entreranno in S nell'ordine: le due celle di costo 1, quella di costo 1.3, quella di costo 1.6, etc.. creando una sorta di nuvola di celle attorno alla casella iniziale che tenderá a svilupparsi sopratutto lungo il fronte delle celle inesplorate (vedi figura 7).

[7]Dicesi *heap* un albero binario i cui nodi, contenenti dati, sono caratterizzati da un campo 'chiave' che soddisfa le seguenti proprietá:

1. tutte le foglie dello heap hanno altezza (cioé numero di predecessori nel risalire verso la radice) pari ad h o h-1 per un valore opportuno di h;
2. se esistono foglie di altezza h-1, esse sono a destra delle foglie di altezza h (ovvero ogni livello viene riempito da sinistra verso destra);
3. per ogni nodo, la chiave del nodo é minore o uguale alla chiave dei figli.

Quindi la radice é il nodo con la chiave minima.

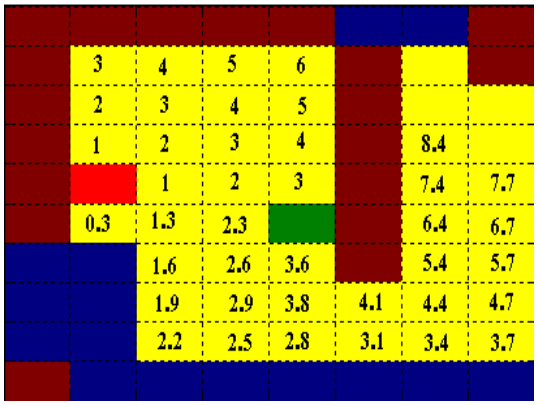


Figura 7: costo caselle con una nuvola

Quando lo sviluppo di S permette di dare un costo anche alla cella contrassegnata dalla X, l'espansione della nuvola si arresta e si puó ricostruire il cammino a costo minimo dalla cella di origine alla X (figura 8). Notare come questo percorso permetta di aumentare l'esplorazione della mappa costeggiando quanto piú possibile la zona inesplorata.

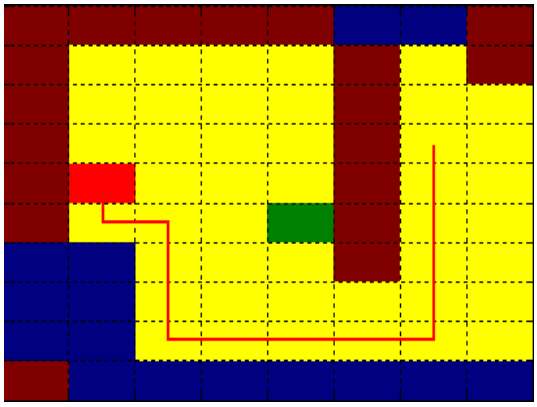


Figura 8: cammino minimo alla cella X

L'implementazione di una **terza variante** si differenzia dalla prima versione per il tentativo di ridurre il numero di celle da analizzare prima di arrivare alla formulazione di un cammino. Osservando la figura 9 si puó intuire che se oltre alla nuvola di celle attorno allo start (rappresentante l'insieme S) si aggiungesse una seconda nuvola attorno alla cella 'goal', si potrebbe ridurre il numero complessivo di celle candidate ad appartenere al cammino fino a quasi la metá, riducendo cosí il numero delle scansioni della mappa.

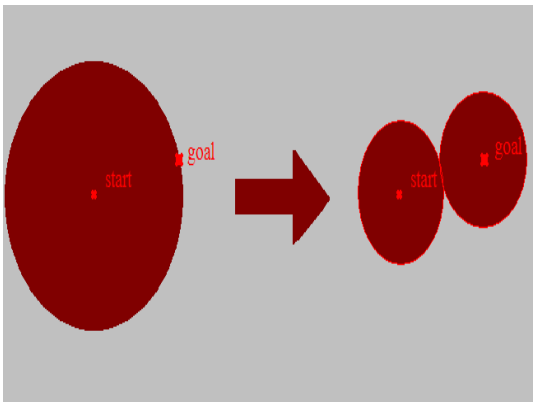


Figura 9: nuvole

L'algoritmo, in questo caso, si ferma al termine del ciclo in cui ad una cella é attribuito sia un costo dal goal che uno dallo start. Qualora alla fine del ciclo più di una cella soddisfacesse questo requisito, viene scelta come 'cella anello' quella che minimizza il costo del cammino. Il cammino viene poi formulato a partire dai percorsi a ritroso dall'anello al goal e dall'anello allo start. Questa semplificazione in certe mappe puó portare alla formulazione di un cammino leggermente diverso da quella che verrebbe trovato dalle versioni precedenti dell'algoritmo: ció é dovuto

1. a possibili selezioni diverse tra celle candidate di pari costo[8]

2. al fatto che per gli observer il cammino minimo dal goal alla cella anello non é detto sia uguale a quello dalla cella anello al goal, a causa della assimmetria dei costi nei passaggio da cella interna a cella frontiera e da cella frontiera a cella interna.

[8] utilizzando la distanza di Manhattan, puó capitare che due punti possano essere uniti da percorsi anche molto diversi, aventi, nonostante ció, la stessa lunghezza.

Questa versione supera le precedenti per quanto riguarda i tempi di calcolo nelle mappe con molte celle e pochi ostacoli.
In una **quarta variante** si é aggiunto allo schema a doppia nuvola l'utilizzo di due code di prioritá, in modo simile a ció che é stato fatto nella seconda versione. Nella figura 10 si può osservare come nella terza e quarta versione dell'algoritmo le due nuvole si sviluppino parallelamente, attribuendo alle celle costi relativi allo start, se denotati in nero, o relativi al goal, se denotati in rosso. La cella che presenta entrambi i costi é la 'cella anello'. Da notare che sebbene siano state marcate sei celle in meno rispetto alla figura 7, verrá prodotto il medesimo percorso della figura 8.

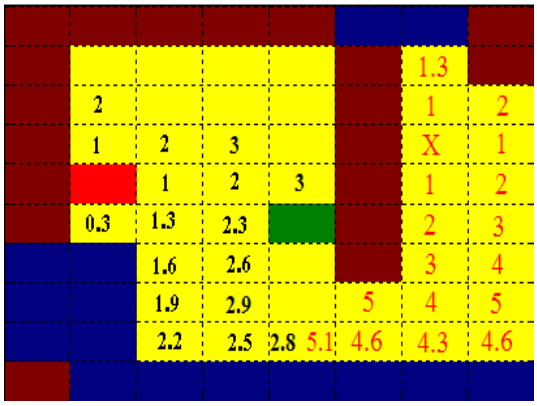


Figura 10: costo caselle con due nuvole

Un riassunto delle varie versioni dell'algoritmo di Dijkstra prodotte é riportato nella tabella 1. Basandosi

versioni	caratteristiche	file
prima	1 nuvola, scansione	Dijkstra.m
seconda	1 nuvola, 1 heap	DijkstraH.m
terza	2 nuvole, scansione	DijkstraDoppio.m
quarta	2 nuvole, 2 heap	Dijkstra2H.m

Tabella 1: versioni dell'algoritmo

sui risultati delle prove e delle simulazioni effettuate, si é concluso che la crescente complessitá del codice nelle varie versioni rende meno prestanti le implementazioni avanzate quando queste vengono applicate su mappe composte da poche celle. Infatti, per mappe di piccole dimensioni quali possono essere i labirinti realizzati su mezza pedana del NAVLAB ($2m \times 1.3m$ per un totale di 260 celle con lato di 10cm), é stata la prima e piú semplice versione dell'algoritmo a fornire i migliori risultati in termini di durata media delle iterazioni del ciclo della simulazone. Su una pedana completa a 540 celle, invece, di solito risulta migliore la terza versione, mentre su un'ipotetica pedana di 1080 celle (di dimensione doppia a quella del NAVLAB), la migliore sarebbe la quarta versione. A seconda del tipo di mappa da esplorare si potrebbe quindi scegliere quale algoritmo usare. Nella tabella 2 vengono riportati degli esempi di durate medie dei cicli di programma al variare della dimensione della pedana e del tipo di implementazione della funzione per la sintesi del cammino minimo per due observer.

versioni	260 celle	540 celle	1080 celle
prima	7.5ms	17.3ms	50.2ms
seconda	8.5ms	20.4ms	47.7ms
terza	8.5ms	15.1ms	48ms
quarta	10.9ms	16.9ms	41.6ms

Tabella 2: durate medie dei cicli

L'actor, finché un percorso che lo conduca al goal non é stato battuto, non ha informazioni su come raggiungere l'obbiettivo: in questa situazione si puó decidere se lasciarlo fermo fino all'individuazione di un cammino per il goal, oppure muoverlo cercando di avvicinarlo all'obbiettivo. Questi comportamenti caratterizzano rispettivamente l'algoritmo NMA e l'algoritmo MA. Sia NMA che MA utilizzano la versione a due nuvole e due heap di Dijkstra. In MA l'actor si dirige alla cella esplorata e libera piú vicina all'obbiettivo. Questa posizione viene determinata analizzando una nuvola di celle in espansione attorno al goal: non appena una cella della nuvola risulta essere esplorata e libera, l'actor vi si dirige. Questa nuvola viene implementata con una coda FIFO inizializzata con il solo elemento goal. Quando una cella viene estratta dalla coda, vengono inserite quelle celle ad essa adiacenti che non sono mai state ancora

presenti nella coda (condizione necessaria per evitare loops).
Qualora gli epuck non riescano a trovare un percorso ad un robot a causa di un altro robot che gli blocca il cammino, si é deciso di far restare fermo il primo robot finché il secondo non si é spostato. Da come é stato implementato l'algoritmo di esplorazione, l'actor, essendo l'ultimo robot a cui viene assegnata la destinazione, vedrá ogni altro robot occupare due celle.

Analisi prestazioni dell'esplorazione e del conseguente movimento dell'actor

Anche in questo caso ogni esperimento é composto da 20 prove in ciascuna delle quali cambiano le posizioni iniziali degli observer OBS_i e dell'actor ACT nella seguente maniera:

$$\langle OBS_i \rangle = \langle 1 + 2i + j_1, j_2 \rangle \;\; j_1 = 1, .., 10, j_2 = 3, 25.$$

$$\langle ACT \rangle = \begin{cases} \langle 1 + j_1, 2 \rangle & j_1 = 1, .., 10, j_2 = 3 \\ \langle 1 + j_1, 26 \rangle & j_1 = 1, .., 10, j_2 = 25. \end{cases}$$

e il goal é sempre in posizione $\langle 9, 15 \rangle$. Per ciascun esperimento si calcola la media e la devizione standard campionaria di N_A e P_A ($\mu_{N_A}, \sigma_{N_A}, \mu_{P_A}, \sigma_{P_A}$) dove N_A é il numero di passi della simulzione (che termina non appena l'actor raggiunge il goal), mentre P_A é il numero di celle visitate dall'actor. I risultati trovati sono mostrati nella seguente tabella.

OBS.	ALG.	μ_{N_A}	σ_{N_A}	μ_{P_A}	σ_{P_A}
1	NMA	145	**33**	**41**	22
	MA	**116**	42	55	**12**
2	NMA	109	34	**28**	**13**
	MA	**91**	**29**	59	19
3	NMA	92	26	**27**	**11**
	MA	**80**	**21**	58	18
4	NMA	86	19	**28**	**11**
	MA	**75**	**16**	56	13

Si puó notare come sia conveniente per raggiungere rapidamente il goal muovere l'actor fin da subito secondo il criterio dell'algoritmo MA. Tuttavia il numero di passi dell'actor é piú elevato, giacché spesso percorsi che all'inizio erano promettenti vengono resi obsoleti da altri percorsi che giungono ad una cella piú vicina al goal. Al variare delle posizioni iniziali dei robot le celle esplorate dagli observer, e dunque le celle a disposizione dell'actor per muoversi, variano molto e quindi il percorso iniziale dell'actor, prima che gli observer scoprano il goal, puó variare rilevantemente: questo si rispecchia in un elevato valore di σ_{P_A} per MA (ad eccezione del caso di un observer).

Modello cinematico dell'uniciclo

I robot e-Puck utilizzati per le prove in laboratorio hanno una struttura meccanica molto semplice, modellizzabile direttamente come un uniciclo. La posizione di un uniciclo su un piano è univocamente determinata dalle coordinate:

$$q = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

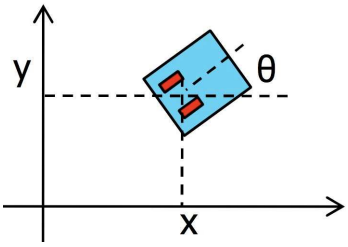


Figura 11: *Schema della struttura di un unicilo.*

Lo stato di un uniciclo è descritto dalle equazioni:

$$\begin{aligned} \dot{x} &= v\cos(\theta) \\ \dot{y} &= v\sin(\theta) \\ \dot{\theta} &= \omega \end{aligned}$$

dove v è la velocità lineare e ω la velocità angolare.

Il vincolo anolonomo imposto dalle ruote può essere espresso dall'equazione:

$$\dot{x}\sin(\theta) - \dot{y}\cos(\theta) = 0$$

che espressa in forma matriciale diventa:

$$\begin{bmatrix} \sin(\theta) & -\cos(\theta) & 0 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = 0$$

Per controllare un uniciclo esistono svariate tipologie di controllo e la letteratura a riguardo è ampia. Si è utilizzato per questo progetto il controllo con linearizzazione dinamica[9], già utilizzato con buoni risultati per i robot e-puck e la strumentazione del laboratorio.

Per il controllo é conveniente introdurre uno stato aggiuntivo $\xi = v$ e si ottiene:

$$\begin{cases} \dot{\xi} = u_1\cos\theta + u_2\sin\theta \\ v = \xi \\ \omega = \frac{(u_2\cos\theta - u_1\sin\theta)}{\xi} \end{cases} \tag{1}$$

dove $u_1 = \ddot{x}$ e $u_2 = \ddot{y}$ sono posti come ingressi. Considerando l'errore di inseguimento di traiettoria così definito:

$$e(t) = \begin{bmatrix} e_x(t) \\ e_y(t) \end{bmatrix} = \begin{bmatrix} x(t) - x_r(t) \\ y(t) - y_r(t) \end{bmatrix} \tag{2}$$

è possibile stabilizzare il sistema mediante un PD così costruito:

$$\begin{cases} u_1 = \ddot{x_r} + K_{p1}(x_r - x) + K_{d1}(\dot{x_r} - \dot{x}) \\ u_2 = \ddot{y_r} + K_{p2}(y_r - y) + K_{d1}(\dot{y_r} - \dot{y}) \end{cases} \tag{3}$$

con $K_p, K_d > 0$.

Come spiegato in [**?**] quello considerato è un controllo che permette l'inseguimento di una traiettoria con buone prestazioni, a patto di avere a disposizione una misura precisa della posizione e dell'angolo θ.

Il controllo così implementato ha una singolarità quando si tenta di far muovere il robot nella direzione del vincolo anolonomo, ovvero quando il riferimento è lungo la retta passante per gli assi delle ruote. Le equazione porterebbero in tal caso ad una velocità angolare infinita. Per evitare quest'evenienza si è impostato un limite inferiore alla velocità ξ sotto al quale non viene calcolata la velocità angolare. Il robot in questa evenienza non si muoverà ($\xi \simeq 0$ e $\omega = 0$ perchè imposto).

Nel tipo di esplorazione a celle considerato in questo progetto una svolta di 90° è un evento molto frequente. Tuttavia, a causa del rumore di misura, un riferimento esattamente a 90° rispetto alla direzione precedente di marcia in pratica non si verifica mai. Con un tempo di campionamento adeguato i robot sono in grado di girarsi in maniera soddisfacente, come si può vedere dai dati sperimentali di figura 12. Per maggior chiarezza della figura sono plottati soltanto gli istanti di campionamento pari.

[9] Dynamic Feedback Linearization, DFL

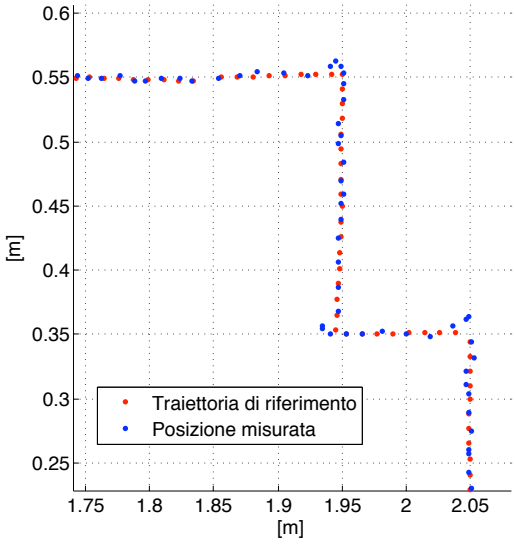


Figura 12: *Traiettorie misurate in uno degli esperimenti presso il laboratorio Navlab. La direzione del movimento del robot è dal basso a sinistra verso in alto a destra.*

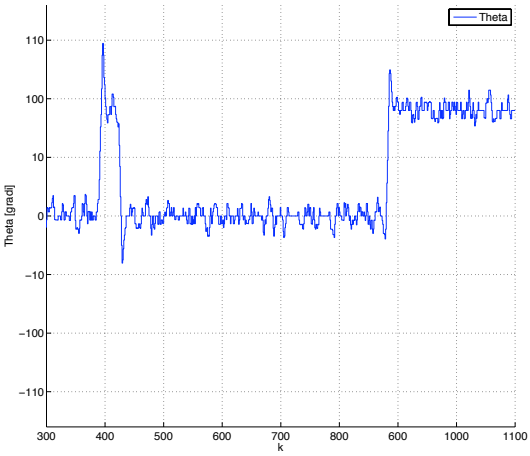


Figura 13: *Andamento dell'angolo θ di un robot misurato in Navlab*

L'andamento nel tempo dell'angolo θ di un robot è plottato in figura 13. L'andamento irregolare è spiegato più avanti. In questa sede si può notare come vi sia ad ogni cambio di direzione una sovraelongazione abbastanza accentuata, tuttavia, l'interesse in questo progetto era la precisione sulla posizione, al fine che un robot rimanga all'interno delle celle coinvolte nello spostamento.

Il controllo è stato implementato nel file `controllore.m`, il quale restituisce in uscita direttamente le velocità da applicare alle ruote dell'e-Puck.

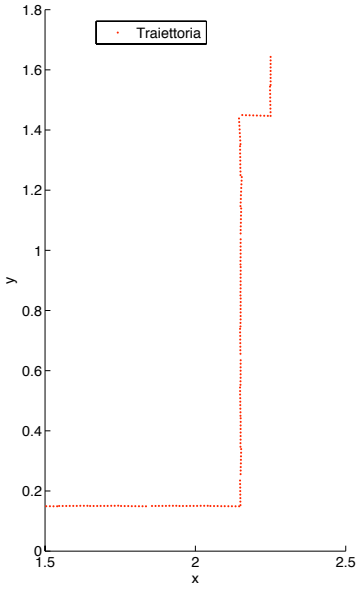


Figura 14: *Traiettoria relativa all'angolo theta di figura 13. Il movimento del robot è dall'alto verso il basso*

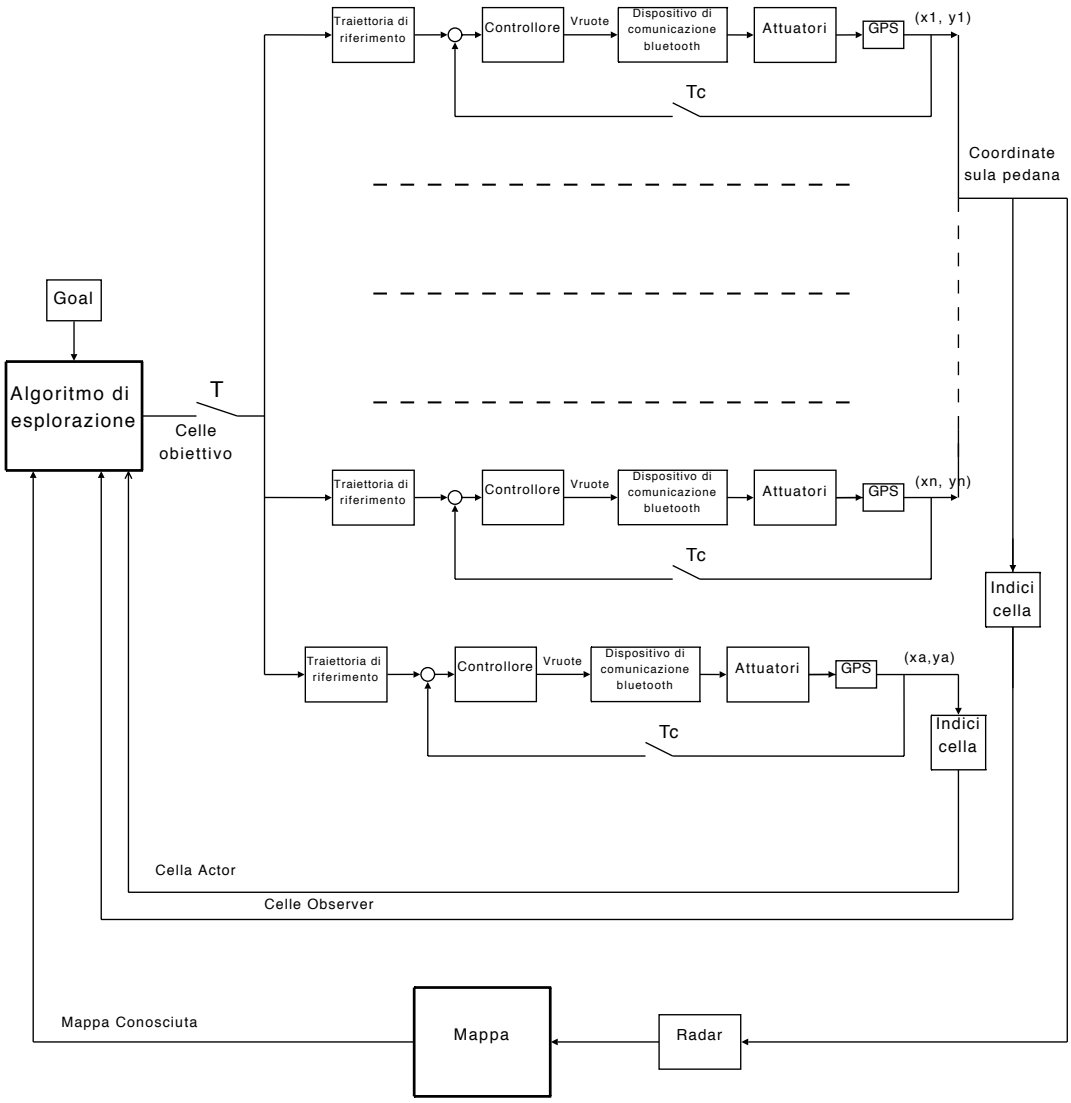


Figura 15: *Schema della struttura del controllo di n robot observer e un actor.*

Struttura di controllo

Lo schema di controllo ed esplorazione implementato per gli esperimenti in laboratorio è rappresentato in figura 15. Ad ogni passo l'algoritmo di esplorazione assegna ad ogni robot la cella successiva che deve raggiungere. Tale scelta avviene ogni T secondi. In laboratorio si è usato principalmente $T = 6\ s$. In base alla cella da raggiungere viene calcolata la traiettoria di riferimento (per evitare discontinuità nel movimento il riferimento è un segmento congiungente il centro della cella obiettivo e l'ultima posizione del robot misurata). Il controllo in catena chiusa è attuato ad ogni periodo di campionamento T_c. In laboratorio abbiamo utilizzato un periodo di campionamento pari a 0.2 secondi. Pertanto ad ogni passo vi sono mediamente $\frac{T}{T_c} = 30$ istanti di campionamento.

Alla conclusione della traiettoria desiderata tutti i robot vengono momentaneamente fermati. Questo permette l'esecuzione di task complessi di pianificazione a medio termine per un numero di robot elevato e allo stesso tempo di mantenere un tempo di campionamento relativamente basso durante i movimenti dei robot (sperimentato $T_c = 0.2\ s$ fino a 7 robot, limite massimo del protocollo Bluetooth sul numero di periferiche collegate). Inoltre si controlla che la posizione raggiunta appartenga alla cella pianificata. Se così non è, perchè il robot è stato bloccato o avviene un guasto, viene rilevata nuovamente la cella in cui si trova.

Come si può notare dallo schema di figura 15 vi è un ciclo esterno che gestisce i task di esplorazione, e N cicli interni di retroazione (uno per robot) che permettono il controllo di traiettoria per i veicoli.

Misura della posizione dei veicoli

La conoscenza della posizione del robot, o una sua stima, e' un dato fondamentale nei problemi di mappatura, che e' il compito al quale sono assegnati gli observer. In generale essa puo' essere ricavata da sensori interni al robot (bussola, odometro...), dal riconoscimento di landmarks, dei quali e' nota a priori la posione, oppure da un sistema di rilevamento esterno, ad esempio il GPS.

La posizione del robot, in un certo sistema di riferimento, serve anche per il moto del robot, in particolare quando si tratta di inseguire una traiettoria non nota a priori, oppure una traiettoria nota ma su terreno sconosciuto o entrambe sconosciute.

Sono stati considerati tre metodi:

- calcolo della posizione raggiunta tramite odometro.
- rilevazione della posizione tramite sistema GPS.
- metodo ibrido.

Nel primo caso e' necessario conoscere la posizione iniziale del robot e la sua orientazione. Le posizioni successive vengono calcolate sulla base degli encoder presenti sui singoli motori. E' cosi' possibile sapere la distanza percorsa e la traiettoria seguita dal robot. Nel caso dell'e-puck questo e' un metodo che porta a buoni risultati, grazie alla meccanica molto semplice del robot e alla precisione elevata della posizione delle ruote dovuta ai motori passo passo (un millesimo di giro di ruota). E' pero' necessario che la superficie su cui si muove il robot sia perfettamente liscia. Tale metodo infatti fallisce se l'ambiente e' irregolare o se le ruote slittano.

Dai test effettuati in Navlab abbiamo riscontrato che tale metodo non poteva essere applicato a causa di saltuari slittamenti delle ruote degli e-puck sulla pedana, dovuti alla polvere presente combinata al peso ridotto del robot. La posizione calcolata, rilevata in varie simulazioni, si discostava troppo da quella realmente raggiunta dal robot. Data la durata degli esperimenti (una decina di minuti) e il fatto che questi errori vanno via via a sommarsi non si e' considerato tale metodo idoneo.

Nel secondo caso la posizione e' ricavata grazie a degli apparati esterni al robot (satelliti per il GPS terrestre, telecamera nel caso del Navlab, reti di sensori ecc...). In generale questi metodi hanno degli errori piu' grandi rispetto all'utilizzo di un odometro (utilizzando la telecamera del navlab l'errore lungo ciascun asse e' di $\pm 2\ mm$). D'altra parte l'incertezza e' sempre la stessa ad ogni misura e permette di evitare derive sulla misura. Tale metodo consente inoltre di implementare un controllo di posizione in catena chiusa.

Data la robustezza di questo metodo, e l'importanza della questione nel progetto, lo si e' utilizzato per le sperimentazioni in laboratorio.

In generale tali metodi potrebbero non essere sempre disponibili (ad esempio per occlusione, guasti temporanei...) o essere dispendiosi in termini di energia spesa e di carico computazionale. Il terzo metodo consiste percio' in una combinazione dei due precedenti: il robot usa principalmente l'odometro per stimare la sua posizione. Quando ritiene che la stima non sia piu' affidabile interroga il dispositivo GPS per aggiornare la sua posizione corrente. Tale aggiornamento puo' essere eseguito ad intervalli regolari oppure secondo necessità (momenti particolarmente delicati, terreni irregolari...).

La pianificazione della traiettoria avviene ad ogni passo, dopo che l'algoritmo di esplorazione ha determinato la cella successiva che dovrà raggiungere ogni singolo robot. Grazie alla capacità dei robot e-puck di ruotare su loro stessi [10] si è potuto semplificare il problema considerando solamente traiettorie rettilinee. Si evitano così approcci meno adatti all'impostazione data al problema come, ad esempio, quelli di concatenazione di primitive di movimento per veicoli di Dubins [**?**].

Non sono stati imposti vincoli sulla direzione di movimento del veicolo, ovvero non vi é una direzione preferenziale. Sia la marcia in avanti che la retromarcia sono ammesse. Questo è giustificato dalla simmetria della struttura dell'e-puck dal fatto che la visuale del radar simulato per l'esplorazione copre ogni direzione attorno al robot. Tale ipotesi rimane sensata anche nel caso si vogliano usare gli otto sensori di prossimità di cui è dotato l'e-puck, essendo questi disposti tutto attorno ad esso.

La successione di riferimenti per la traiettoria viene calcolata interpolando con una retta il punto in cui si trova il robot P_{in} e il punto di arrivo P_{fin} situato al centro della cella obiettivo. Il riferimento potrebbe essere calcolato interamente ad ogni passo ma, per ridurre l'utilizzo di memoria e semplificare il codice, viene calcolato in maniera incrementale ad ogni istante di campionamento secondo la formula:

$$P_r(k+1) = P_r(k) + V_r(k)T_c \tag{4}$$

$$V_r(k+1) = \frac{P_{fin} - P_r(k+1)}{T - kT_c} \tag{5}$$

Dove con T_c si indicato il tempo di campionamento, con T il tempo a disposizione per compiere il tratto assegnato e con V_r la velocità di riferimento relativa alla legge di moto desiderata.

Il tipo di controllo utilizzato permette l'inseguimento di una traiettoria rettilinea con discreta precisione, eccetto nella parte iniziale del movimento a causa del transitorio iniziale. La dimensione della cella della mappa e' di 10 cm, in modo tale da avere un margine di sicurezza per gli errori nell'inseguimento della traiettoria. Si e' riscontrasto sperimentalmente che in tutte le manovre considerate tale dimensione e' adeguata e permette al robot di non uscire dalle celle sicure. Tuttavia si è sperimentato che l'approcio utilizzato è molto sensibile agli errori di misura sull'orientazione del robot. Questo perchè il riferimento viene dato istante per istante in un punto vicino al centro del robot. Un piccolo errore sulla posizione lungo direzione ortogonale a quella del riferimento porta il robot a orientarsi nella direzione del punto obiettivo. Pertanto si riscontrano delle oscillazioni attorno al centro del robot. Tali oscillazioni non comportano uno spostamento del centro del robot rispetto alla traiettoria desiderata, ma ne alterano solo l'orientazione. Tale fenomeno porta ad un inutile dispendio di energia e puo' creare degli inconvenienti se si utilizzano in contemporanea altri tipi di sensori, ad esempio bussola, telecamera ecc. Per limitare questo inconveniente si sono applicate le seguenti correzioni:

- riduzione del numero di riferimenti, ovvero si e' aumentata la distanza tra i punti che descrivono la traiettoria desiderata. Questo avviene limitando il rapprto $\frac{T}{T_c}$ dato che lo spazio da percorrere in un passo e' sempre 10 cm, ovvero la dimensione della cella. Cosí ci si e' allontanati dalla posizione di singolarita', che avviene quando il punto di riferimento successivo si trova a coincidere con il centro del robot. In questo caso la direzione voluta non sarebbe specificata.

[10] non si é imposto pertanto un raggio minimo di curvatura

- Limitazione della velocita' angolare calcolata dal controllore. Questa non linearita' nel controllo e' pero' un parametro delicato da tarare e se limitata troppo puo' portare ad instabilita'. In generale va aumentata al diminuire del tempo di campionamento e va ridotta all'aumentare del tempo T. Una limitazione di questo tipo ovviamente porta a dei transitori di durata maggiore, ovvero nei cambi di direzione si ha un errore iniziale di posizione maggiore, ma permette di descrivere delle traiettorie più dolci.

- Miglioramento della precisione nel rilevamento della posizione tramite telecamera. Si e' modificato l'algoritmo per la simulazione di un sistema di rilevamento GPS, introducendo un filtraggio dell'immagine prima di rilevare la posizione dei led. Questo ha permesso di rilevare con buona precisione il centro della nuvola di punti corrispondente a un led nella foto e quindi di ridurre il rumore di misura. Per ulteriori dettagli vedi appendice .

Nell'attuazione del controllo si sono trascurati i ritardi dovuti all'aquisizione e all'elaborazione dell'immagine, nonche' i tempi di comando dell'epuck tramite il protocollo di comunicazione bluetooth. Per dare il comando di velocita' delle ruote dell'epuck si e' preferito usare il comando di basso livello 'write(epuck,d v_left v_right])', sebbene nella guida disponibile on-line consigliano di usare comandi di alto livello. Si è riscontrato pero' che con quest'ultimi il tempo durante il quale viene impegnato Matlab e' di qualche decimo di secondo, dovuto al fatto che attende una risposta dall'epuck di effettiva attuazione del comando.

Riconoscimento della posizione dei robot tramite telecamera

I file per il riconoscimento della posizione dei robot tramite telecamera, già disponibili, sono stati modificati in alcuni punti, con l'obiettivo di migliorare la precisione della misura.

Nel file principale `main()` viene scattata ad ogni istante di campionamento una fotografia della pedana, alla quale viene sottratta la foto della pedana vuota ottenuta in fase di inizializzazione. Tale fotografia è passata come parametro alla funzione `findPoint()` la quale cerca la posizione di $3N_{robot}$ led. I punti così ottenuti tuttavia non corrispondono esattamente alla posizione del baricentro del robot sulla pedana. Questo perché i led dell'e-puck sono ad un'altezza di circa 4 cm sopra al piano di movimento (figura 16). Questa differenza porta ad un errore di posizione variabile, nullo al centro della pedana e che aumenta linearmente verso i bordi fino ad un valore massimo di circa 3 cm.

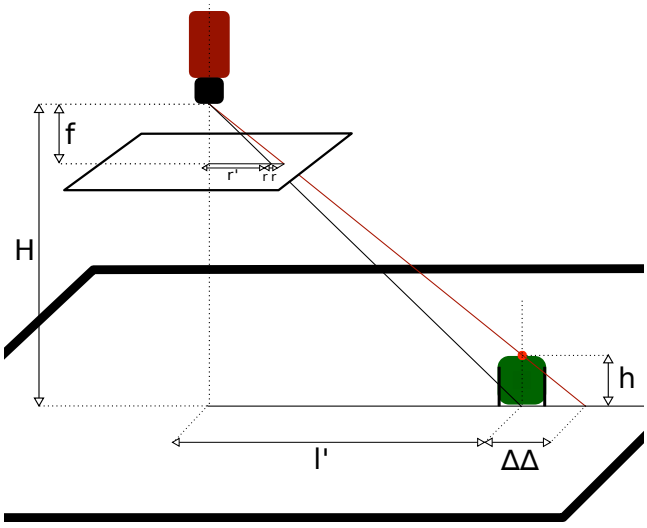


Figura 16: *Correzione della prospettiva*

Perciò si è migliorato l'algoritmo introducendo un fattore di correzione prospettica, ricavato dalle relazioni tra triangoli simili:

$$l = l' + \Delta l$$
$$l = r\frac{H}{f}$$
$$\Delta l = r\frac{h}{f}$$
$$r' = l'\frac{f}{H}$$

e infine:

$$r' = \frac{(l - \Delta l)f}{H} = \frac{r}{H}(H - h)$$

Si nota che se fosse $h = 0$ (led a livello pedana) non vi sarebbe bisogno di tale correzione. L'equazione trovata non dipende dalla lunghezza focale della telecamera, ma solo dal modello che si è utilizzato per rappresentarla, ovvero quello a buco di spillo.

Il vettore di punti così ottenuto è passato alla funzione `findRobot()`, la quale raggruppa i punti trovati a 3 a 3, in base alla distanza relativa che devono avere per essere i 3 led di un e-puck. Questa è una fase delicata in quanto, se due robot fossero vicini (i.e. su celle adiacenti), potrebbe esservi un'associazione errata dei led. In realtà, visto che l'algoritmo raggruppa solo i led che si trovano a distanze fissate, questo è un evento alquanto improbabile e, come si può vedere in alcuni spezzoni di filmati, il sistema funziona anche con robot a distanza ravvicinata.

Successivamente, per ogni tripletta valida trovata, viene chiamato il costruttore dell'oggetto `Robot` il quale individua posizione e orientazione dell'epuck (l'orientazione consiste nella posizione del led centrale).

Rispetto all'esperienza descritta in [**?**], la necessitá di avere vincoli meno blandi sulle distanze minime fra epuck ed una divisione di ruoli fra robot ha richiesto di rivedere certi files e procedure usati in [**?**]. In tale esperienza, infatti, la metodologia per evitare gli ostacoli e la disposizione in formazione degli epuck garantiva che i robot non si avvicinassero mai troppo: nelle situazioni qui studiate, invece, spesso i robot hanno dovuto destreggiarsi in labirinti che obbligava loro a procedere a distanze esigue l'uno dagli altri. Ció puó causare problemi nel riconoscimento di epuck vicini da parte della telecamera. Si é quindi pensato di rinunciare ai due led posti alle estremitá del diametro in favore dei tre led posti 'a prua' (i tre led rossi di figura 17). Ció consente di tenere una maggiore distanza fra i led di due robot che procedono affiancati. Se nel precedente progetto il centro del robot veniva individuato come il punto medio del segmento che univa i due led posti agli estremi del diametro, ora si pone come centro il secondo estremo del segmento $\overline{AD}$ uscente dal led centrale e passante per il punto

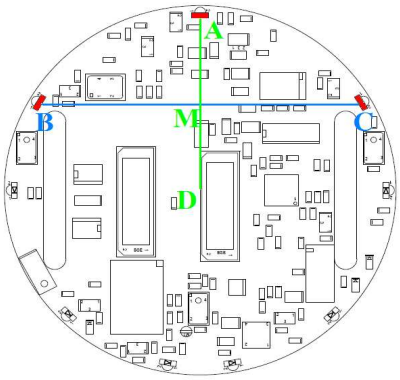


Figura 17: centro del robot

medio del segmento $\overline{BC}$ unente gli altri due led. Si é posto $\overline{AM} = \overline{MD}$.

Si osservi ora la figura 18: si tratta di un particolare di un fotogramma ripreso dalla telecamera del NAVLAB durante la sperimentazione di un labirinto.

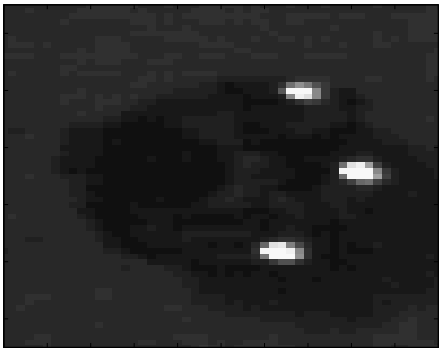


Figura 18: fotogramma telecamera NAVLAB

Come giá precedentemente accennato, un altro problema affrontato ha riguardato la diversificazione dei ruoli fra epuck. Mentre nel precedente progetto l'indifferenziazione dei robot rendeva ininfluente il fatto che essi venissero indicizzati in modo assolutamente casuale durante l'inizializzazione del programma, in questa sede é stato necessario avere controllo

sull'assegnazione dell'indice dell'actor. La soluzione trovata é stata quella di modificare l'inizializzazione agendo su un robot alla volta, accendendone i led, rilevando la posizione ed infine spegnendone i led. Questo procedimento permette di associare univocamente la posizione dell'oggetto Robot al rispettivo oggetto di tipo `ePicKernel` attraverso il quale si è dato il comando di accensione dei led.

In uscita alla funzione `findForm()` si ha, dunque, un vettore di N_{robot} oggetti di tipo Robot.

Come ultimo passo si associano le posizioni appena trovate a quelle precedenti secondo il criterio della minima distanza. Tale compito è svolto dalla funzione `assRobot()`.

Per rilevare la posizione dei robot all'inizio della prova sperimentale, si esegue un ciclo nel quale si accendono, un robot alla volta, i led per la localizzazione. Viene poi richiamata la funzione `findForm()`, la quale restituisce un unico oggetto di tipo robot.

Un'ulteriore miglioria apportata all'algoritmo di visione è nella funzione `findPoint()` che precedentemente cercava il massimo valore della nuvola di punti corrispondente ad un led. Tuttavia a causa del rumore presente nella foto tale metodo può individuare un pixel che non corrisponde esattamente al centro del led. Filtrando l'immagine con un filtro dello stesso tipo del target che si sta cercando, si accentua l'intensità in corrispondenza del centro del target, ovvero si ottiene un picco nel punto che fitta meglio la funzione filtrante. Si è dunque fatta l'ipotesi che un led possa essere rappresentato da una funzione gaussiana bidimensionale, ovvero che:

$$I(x,y) = I_c \; e^{-\frac{(x-x_c)^2+(y-y_c)^2}{\sigma^2}}$$

dove $I(x,y)$ indica l'intensità dei pixel nell'immagine, I_c è l'intensità nel centro della gaussiana e (x_c, y_c) le sue coordinate.

Si applica pertanto all'immagine un filtro gaussiano di dimensione 3x3, sufficiente per i nostri scopi. Un filtro di dimensioni maggiori aumenterebbe considerevolmente i tempi di elaborazione, senza portare miglioramenti apprezzabili. In figura 19, nella prima riga, ci sono le immagini originarie di due led diversi. Nella seconda riga le stesse immagini filtrate. I pallini

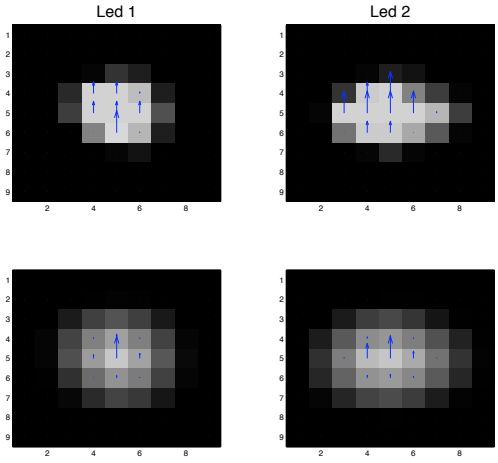


Figura 19: *Confronto tra i due metodi di rilevazione dei led*

indicano i punti di massima intensità. Nel caso del led 1 il picco di intensità non corrisponde al centro della nuvola. Nel caso del led 2 ci sono ben cinque pixel alla stessa intensità massima. Nel primo caso il pixel scelto come riferimento non sarebbe quello ottimo, nel secondo caso non è chiaro quale pixel verrebbe scelto come centro del led. Si vede chiaramente, nella seconda riga, come un filtraggio dell'immagine risolva correttamente entrambi i problemi.

Nella pratica il processo di identificazione dei led si è rilevato particolarmente delicato in quanto molto sensibile ai riflessi che compaiono nello spazio di lavoro in quanto sono confusi con i led dall'algoritmo di visione. Per limitarli è stato necessario coprire con del nastro nero le parti metalliche o molto lucide dei robot presenti in pedana. Un'altra fonte di errori, per lo stesso motivo dei riflessi, è la luce del sole diretta.

Risultati sperimentali

In laboratorio abbiamo sperimentato gli algoritmi messi a punto in simulazione. Si sono fatte varie prove con un numero di observer variabile da uno a sei e con vari parametri per gli algoritmi di esplorazione.

Con un observer e un actor si è utilizzata solamente mezza pedana per non rendere troppo lunga la prova. In questo caso si è usato $\alpha = 1$ e, come spiegato in precedenza, per questo motivo l'esplorazione ottenuta non è ottima (figura 20).

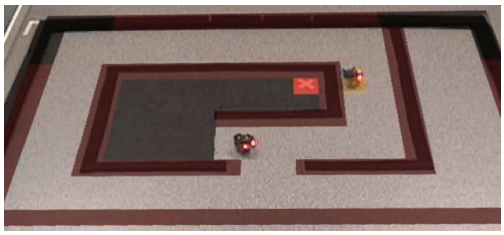


Figura 20: *Esplorazione con un observer e un actor. Il goal é indicato in rosso*

In alcuni dei filmati proposti non si erano ancora implementati le correzioni della prospettiva della telecamera. Si vede in questi casi come alcuni robot passino molto vicino agli ostacoli ai bordi della pedana. Inoltre, sempre per lo stesso motivo, si nota un decadimento delle prestazioni del controllo di inseguimento di traiettoria man mano che ci si allontana dal centro del campo di lavoro.

Con più robot si è poi sempre utilizzata la pedana intera. Date le sue dimensioni e quelle delle celle si è visto che il miglior bilancio tra tempo di completamento della missione e numero di observer impiegate si ha con un numero pari a 3 (figura 21).

Ovviamente le prestazioni dipendono in grande misura dalla mappa da esplorare. Se la mappa è abbastanza libera un numero maggiore di observer permette di diminuire i tempi di esplorazione, se la mappa invece è complessa un numero elevato di robot può portare a problemi di traffico. Si vede infatti nei filmati con 6 e 7 robot che in certi momenti alcuni veicoli rimangono fermi sia per evitare collisioni, sia perchè hanno la strada occupata (figura 22 e 23). La

Figura 21: *Esplorazione con tre observer e un actor. Il goal é indicato in rosso. In giallo la strada percorsa dall'actor.*

scelta é stata fatta in quanto è preferibile aspettare l'evoluzione della scena piuttosto che muovere il robot in punti che potrebbero portare ad intasamento o ad un'esplorazione ridondante.

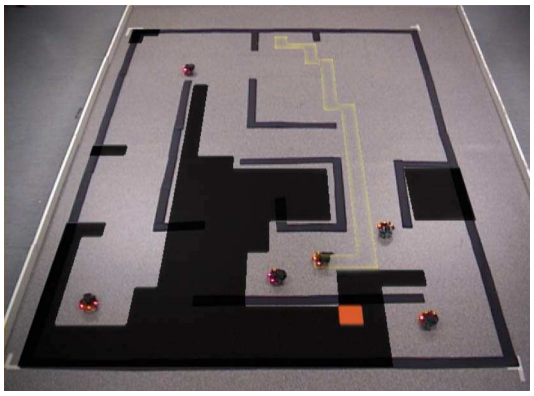


Figura 22: *Esplorazione con cinque observer e un actor. Il goal é indicato in rosso. In giallo la strada percorsa dall'actor.*

Si è infine simulato il guasto di un observer bloccandolo con una trappola. In questo caso l'esplorazione prosegue comunque, con un altro observer che va a coprire la zona rimasta inesplorata (figura 24).

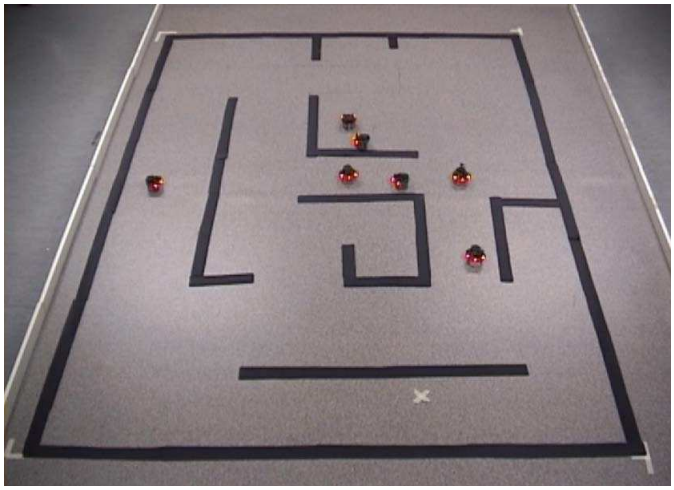


Figura 23: *Esplorazione con sei observer e un actor.*

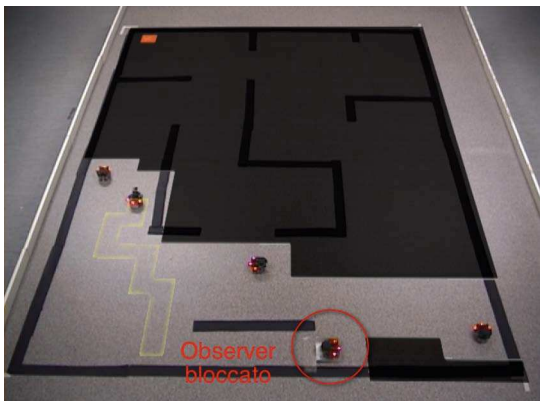


Figura 24: *Esplorazione con cinque observer e un actor. L'observer in basso è bloccato da un ostacolo non rilevabile. Si vede come l'area di sua competenza venga assegnata ad un altro robot.*

Conclusioni

In questo progetto si sono affrontati i seguenti temi:

1. sviluppo di una tecnica di controllo che permetta di avere tempi di campionamento sufficientemente bassi pur applicando algoritmi onerosi dal punto di vista computazionale
2. confronto e sviluppo di diverse tecniche di esplorazione
3. implementazione di un algoritmo di path planning sufficientemente veloce a livello computazionale.

Per quanto riguarda il primo tema, si sono raggiunti risultati soddisfacenti: la snellezza del controllo della movimentazione permette ai robot di spostarsi di cella in cella utilizzando un tempo di campionamento piuttosto basso (0,2 secondi), ottenendo traiettorie precise e la sicurezza che i robot non vadano in celle inesplorate o occupate da ostacoli. Questo ha inoltre permesso di lavorare col massimo numero di e-puck possibili senza eccessivi problemi di comunicazione durante la sperimentazione. L'unico aspetto sfavorevole é l'arresto dei robot per l'applicazione del'algoritmo di esplorazione e di path planning che peró, tranne in casi in cui siano presenti molte celle frontiera, non é cosí evidente a livello visivo.
Modifiche quali lo spostamento dei led per l'identificazione dei robot e la taratura ad hoc dei parametri secondo la luminositá della telecamera hanno permesso ai robot di avvicinarsi molto senza creare problemi.

Per quanto riguarda l'esplorazione, é stato inventato un algoritmo originale (l'algoritmo EB), sebbene non molto efficiente. Inoltre é stato sviluppato un'algoritmo ispirandosi a quello proposto in [**?**]. In particolare si é assegnata una funzione costo piú grossolana ma computazionalmente piú efficiente; inoltre l'inizializzazione e l'aggiornamento della funzione utilitá é stata sviluppata diversamente. Tuttavia non é stato possibile confrontare i risultati riportati in [**?**], in quanto il contesto applicativo é molto diverso. Si é concluso che l'approccio collaborativo nell'esplorazione multi robot é il piú produttivo. Per quanto riguarda il path planning si é cercato di snellire per quanto possibile i tempi di computazione, non raggiungendo peró una formulazione che si dimostrasse superiore alle altre in ogni possibile mappa. Quindi a seconda delle dimensioni della mappa si dovrebbe scegliere oppotunamente

una delle quattro versioni. L'utilizzo di pesi diversi per raggiungere celle interne e celle di frontiera nell'algoritmo di Dijkstra per gli observer, ha dato buoni risultati dal punto di vista esplorativo. La tecnica di path planning per le prime fasi della simulazione in cui non si conosce ancora un cammino al goal per l'actor, ha ridotto la durata delle simulazioni, pur peggiorando la lunghezza del percorso dell'actor.

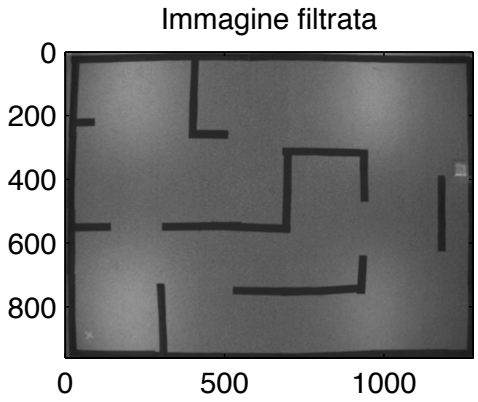
Sbocchi futuri

Un'ulteriore proseguimento del progetto potrebbe essere una modifica dell'algoritmo di esplorazione e di path planning che permetta ai robot di muoversi non solo nelle celle adiacenti ma anche in quelle vicinanti. Inoltre per rilevare gli ostacoli si potrebbe utilizzare i sensori a infrarossi di cui sono dotati gli e-puck. Per quanto riguarda l'esplorazione, un'alternativa interessante da testare sarebbe quella di non riassegnare i target ad ogni passo, lasciando che gli observer li raggiungano (ed in questo caso si potrebbero studiare delle traiettorie e delle leggi del moto piú complesse per fare ció). Una mappatura differente si potrebbe ottenere dall'utilizzo della tecnica quadtree (una mappatura di tipo grid-map che prevede la decomposizione delle celle che comprendono sia spazio libero che ostacoli in celle pú piccole): questa permetterebbe di avere un'informazione piú dettagliata sulla configurazione degli ostacoli senza che ció comporti l'utilizzo di una griglia per la mappa eccessivamente fitta e quindi onerosa da rielaborare. Sarebbe interessante valutare la situazione in cui gli observer sono eterogenei, ovvero hanno raggi di visibilitá diversi: in questo caso bisognerebbe strutturare diversamente l'aggiornamento della funzione utilitá. Si potrebbe poi impostare il problema in maniera probabilistica: in tal caso si considerano i dati forniti dai radar come affetti da errore di misura. Si potrebbe supporre inoltre di non conoscere la posizione relativa tra i robot: in questo caso peró il problema diventa assai complesso. Infine sarebbe interessante considerare un approccio decentralizzato per l'esplorazone e il mapping.

APPENDICE

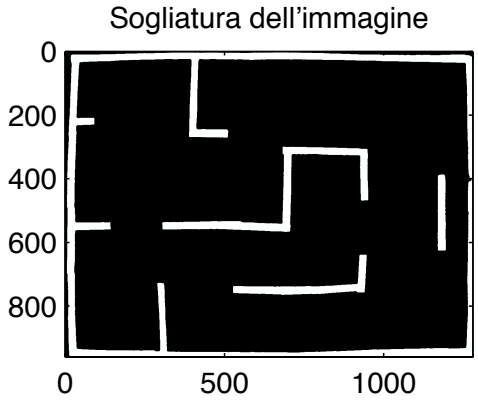
Riconoscimento degli ostacoli

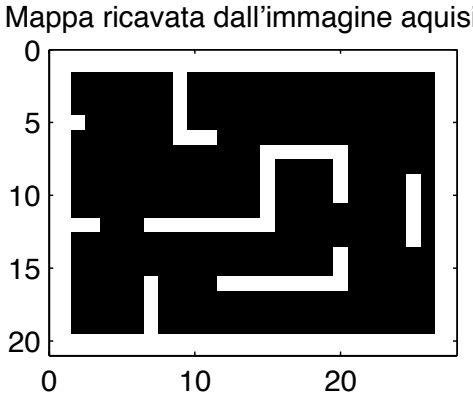
In fase di inizializzazione del sistema sperimentale, viene fotografata la pedana senza robot. Gli ostacoli sono rappresentati da delle strisce di cartone nero. Sono rilevati automaticamente dalla funzione `rileva_ostacolo()`, la quale esegue una sogliatura sulla luminosità della foto. La soglia varia in base all'intensità media della foto, in modo da rendere il procedimento affidabile in qualsiasi condizione di luce. Se nell'area coperta da una cella un certo numero di pixel supera la soglia, essa viene considerata come cella occupata (figura). Dalla fotografia si nota una certa deformazione dell'immagine, soprattutto ai bordi, dovuta all'obiettivo. Tuttavia questo tipo di problema è più complesso computazionalmente da correggere rispetto a quello della prospettiva, pertanto lo si è trascurato.



E-puck

Nei test sperimentali del progetto sono stati utilizzati alcuni veicoli mobili denominati e-puck: trattasi questi di robot pensati a fini didattici di livello universitario dall' Ecole Polytechnique Fédérale de Lausanne. Sono strumenti che, grazie ai molti sensori embadded ed alla possibilitá di aggiungere diversi moduli, permettono attivitá di studio e di ricerca in molti settori, quali la robotica coordinata, la programmazone real-time o l'iterazione uomo-macchina. La struttura della macchina é basata su un corpo unico del diametro di $70mm$ a cui sono fissati i supporti dei motori, il circuito e la batteria. La batteria, allocata al di sotto del motore, é collegata al circuito con due contatti verticali e puó facilmente essere estratta per essere ricaricata. Questa batteria dovrebbe essere sufficiente (stando a quanto riportato dai costruttori) a 2-3 ore di uso intensivo: tuttavia si é notato che e-puck muniti di batterie non campletamente cariche spesso creano problemi, dando problemi di comunicazione o spegnendo i led. Si consiglia quindi,





per quanto possibile, di lavorare con batterie sempre fresche di ricarica.

Figura 25: e-puck

Questi robot utilizzano due motori a passo con riduttore di marcia. Essi realizzano 20 passi per rivoluzione e la marcia possiede una riduzione di 50 : 1. Le due ruote hanno un diametro di circa $41mm$, sono munite di un sottile pneumatico, distano circa $53mm$ e possono raggiungere la massima velocitá di 1000 passi al secondo, ovvero un giro completo di ruota al secondo. Gli e-puck possono comunicare tramite un chip bluetooth, che permette di accedere al robot come se fosse connesso ad una porta seriale (pur mandando particolari comandi per la connessione e sconnessione e lavorando ad una distanza tra i 15cm e i 5m). I bordi della superficie superiore presentano 8 led rossi controllati utilizzando 24 segnali (Pulse Width Modulated) separati, generati da un microcontrollore posizionato internamente all'e-puck. I led possono essere controllati singolarmente, oppure in maniera sincronizzata, semplicemente settando pochi parametri (come la massima luminositá, il periodo di intermittenza etc.). Altre dotazioni (non utilizzate in questo progetto) sono:

- tre microfoni a 33kHz di massima frequenza di acquisizione,
- un accelerometro 3D che misura le micro-accellerazioni e micro-decellerazioni del movimento del robot,
- 8 sensori di prossimitá IR (Infrared proximity sensors),
- una telecamera a colori con risoluzione $640(h) \times 480(l)$ pixel.

Come giá accennato, é inoltre possibile montare sul robot alcuni moduli quali:

- un circuito con tre sensori IR puntati verso terra
- una torretta con 8 led multi-color
- una torretta che monta tre telecamere per ottenere una visione quasi omnidirezionale.

Strumentazione hardware

Il laboratorio *Navlab* dove si è sperimentato il progetto è dotato di una pedana rialzata e piana dove si possono muovere i robot e-Puck. La porzione di pedana coperta dalla telecamera fissata al soffitto, e quindi l'area utile di lavoro per i nostri test, misura $2.7\ m \times 2\ m$. I veicoli mobili utilizzati sono dei robot e-Puck.

Il calcolatore utilizzato per le prove in *Navlab* è dotato di processore Intel Core 2 Duo a 2 GHz con 4 MB di memoria cache L2 e 2 GB di RAM DDR2 a 667 MHz. Si è preferito l'utilizzo di tale computer, piuttosto di quello presente in Navlab, in quanto piu' veloce sia nell'elaborazione delle immagini sia nella comunicazione bluetooth con i robot. Questo ha permesso di abbassare il tempo di campionamento e di conseguenza di aumentare la velocita' media di navigazione degli e-puck.

La telecamera in dotazione è una AVT Marlin F-131B (http://www.alliedvisiontec.com). Si tratta di una telecamera con interfaccia firewire, acquisisce immagini in scala di grigi ad una risoluzione massima di 1280 x 960 pixel. È caratterizzata da un basso ritardo di acquisizione, quindi è particolarmente indicata per applicazioni real-time. L'obiettivo è un 8 mm dotato di due ghiere di regolazione, una per la messa a fuoco e l'altra per l'otturatore, permettendo così di tarare con precisione la luminosità dell'immagine al fine di evitare saturazioni con conseguente perdita di definizione. È possibile per lo stesso motivo usarla

con diverse condizioni di luce ambientale (luci accese, spente, giorno, notte...).

Per acquisire le immagini si è sfruttato il file *vcapg2.dll* scritto in linguaggio C++ dall'autore Kazuyuki Kobayashi, il quale permette di acquisire un frame dalla telecamera collegata ogni volta che viene chiamato dal file in esecuzione in Matlab. Un'altra possibilità è quella di usare le funzioni del pacchetto "imaqtool" di Matlab. Queste ultime però hanno dei tempi di elaborazione dell'ordine di qualche decimo di secondo, rispetto a qualche centesimo di secondo della libreria "vcapg2.dll". Pertanto non è utilizzabile per un'applicazione real-time come quella implementata in questo progetto.

Riferimenti bibliografici

[1] W. Burgard, M.Moors, D. Fox R., Simmons, S. Thrun, Collaborative Multi-Robot Exploration, IEEE Aprile 2000.

[2] Jiming Liu, Jianbing Wu Multi-Agent Robotic Systems CRC Press, 2001

[3] G. Barbera, P. Calore, G. Cosi. Approccio comportamentale al controllo coordinato di WMR. Università degli studi di Padova, Marzo 2008.

[4] I.M. Rekleitis, G. Dudek, E.E. Milios Multi-robot exploration of an unknown environment, efficiently reducing the odometry error. In Proc. of International Joint Conference in Artificial Intelligence (IJCAI), 1997.

[5] I.M. Rekleitis, G. Dudek, and E.E. Milios. Accurate mapping of an unknown world and online landmark positioning. In Proc. of Vision Interface (VI), 1998

[6] K. Singh and K. Fujimura. Map making by cooperating mobile robot. In Proc. of the IEEE International Conference on Robotics & Automation (ICRA), 1993.

[7] B. Yamauchi. Frontier-based exploration using multiple robot. In Proceedings of the Second International Conference on Autonomous Agents, pages 47, 53, 1998.

[8] Matteo Fischetti, Lezioni di Ricerca Operativa Edizioni Libreria Progetto, Padova 1999

[9] Alessandro Agnoli, Controllo di veicoli anolonomi su ruota, Tesi di laurea Specialistica, Universitá degli studi di Padova - Dipartimento di Ingegneria dell'automazione, Ottobre 2007 .

[10] www.e-puck.org

[11] K. Kakusho, T. Kitahashi, K. Kondo, J. C. Latombe. Continuous Purposive Sensing and Motion for 2D Map Building

[12] I. Rekleitis, G. Dudek, E. Milios. Multi-robot collaboration for robust exploration, Annals of Mathematics and Artificial Intelligence, 2001.

[13] S Thrun - Exploring Artificial Intelligence in the New Millennium, 2002

[14] G.A. Bekey. Autonomous Robots, Massachussets Institute of technology 2005.

[15] Jean-Claude Latombe Robot Motion Planning