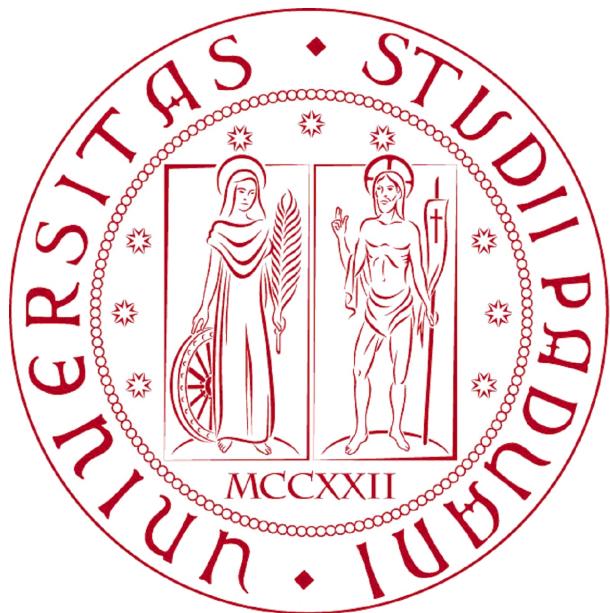


UNIVERSITA' DEGLI STUDI DI PADOVA

CORSO DI ROBOTICA AUTONOMA

ESPERIENZA 1



GRUPPO 11:
Marco Bertagnoli
Matteo Mastellaro
Angelo Trevisol

Anno Accademico 2015-2016

Indice

1 Panoramica	2
1.1 Mappa	2
1.2 Scenario	2
1.3 Robot Lego NXT	3
2 Realizzazione del Software	3
2.1 Mappa (angelo)	4
2.2 Navigazione (angelo)	4
2.3 Spostamento del robot (matteo)	4
2.3.1 Rotazione	4
2.3.2 Movimento in linea retta	5
2.3.3 Avanzamento di una cella	5
2.3.4 Centramento all'interno di una cella	5
2.3.5 Ricerca dei coni	5
3 Test in laboratorio	6
3.1 Descrizione esperienza	6
3.2 Ostacoli mobili	6
4 Conclusioni	7

1 Panoramica

L'obiettivo generale di questa esperienza è implementare un software che sia in grado di guidare un robot Lego Nxt all'interno di una mappa, da una certa posizione detta Start ad una posizione finale, detta Goal, in diversi scenari che racchiudono diverse difficoltà. Tale software sarà implementato in linguaggio C++, orientato agli oggetti, e la comunicazione con il robot sarà effettuata attraverso ROS (Robot Operating System) i cui dettagli verranno discussi più avanti.

1.1 Mappa

La mappa in cui il robot può muoversi è una matrice su sfondo verde, composta da $n \times m$ quadrati, individuati da linee bianche, in cui all'interno di ciascuno è possibile trovare un ostacolo, individuato da un cono. Il movimento all'interno della mappa è consentito soltanto nelle direzioni Nord, Sud, Est e Ovest, ovvero non è consentito il movimento diagonale. Una rappresentazione della mappa presente in Laboratorio è rappresentata in Figura 1.

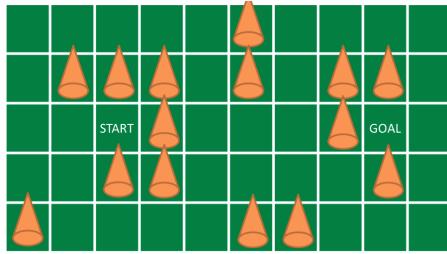


Figura 1: Rappresentazione della mappa presente in laboratorio.

1.2 Scenario

La navigazione del robot Nxt sarà effettuata in diversi scenari. In un primo scenario la mappa non presenterà ostacoli; questo è il caso più semplice infatti l'algoritmo che guida il robot non dovrà preoccuparsi di individuare eventuali presenze di ostacoli, ma potrà liberamente muoverlo verso l'obiettivo.

In un secondo scenario, la mappa presenta degli ostacoli fissi, la cui posizione viene fornita in anticipo al robot, in modo che quest'ultimo possa calcolare il percorso prima di iniziare a muoversi. Una volta che il percorso è stato valutato, il robot dovrà soltanto seguire il tracciato. Questa seconda situazione presenta una difficoltà aggiuntiva legata alla posizione degli ostacoli, che per esempio potrebbero portare il robot in una strada senza uscita.

Infine l'ultimo e il più complesso scenario, nel quale il robot non conosce in anticipo la presenza degli ostacoli (che potrebbero anche spostarsi nel tempo) e dunque è costretto, passo dopo passo, a rilevare gli ostacoli attraverso i sensori e valutare di volta in volta il percorso per arrivare all'obiettivo.

Una buona pratica per passare informazioni al robot sul tipo di scenario che esso deve affrontare, è quella di passare attraverso un file con estensione .yaml. Dunque la scrittura del file mappa.yaml è stata fatta in questo modo:

```
%YAML:1.0
Dimensione:
altezza: 5
lunghezza: 10
Inizio:
x: 5
y: 2
Arrivo:
x: 8
y: 2
Ostacoli:
numero_ostacoli:0 #15
x:[] # [0, 2, 3, 3, 3, 2, 1, 5, 6, 5, 5, 8, 7, 7, 8]
y:[] # [0, 1, 1, 2, 3, 3, 3, 0, 0, 3, 4, 1, 2, 3, 3]
DrawMap: 1
TerzoPunto: 1
```

In questo script si è specificato nel campo *Dimensione* rispettivamente il numero di righe e di colonne della matrice-mappa, in *Inizio* e *Fine* rispettivamente le celle di start e goal in coordinate cartesiane, e nel campo

Ostacoli vengono specificati il numero e le posizioni degli ostacoli all'interno della mappa. Con il campo *DrawMap* è possibile attraverso il valore 1, comunicare al software di voler aggiungere una mappa grafica, creata mediante il tool *Opencv*, che visualizza graficamente i progressi del robot, ad ogni istante temporale. Il suo funzionamento verrà chiarito nella sezione di test in laboratorio. Per concludere, con il valore 1 è possibile attivare il campo *TerzoPunto*, con cui si comunica al robot che la mappa è a priori ignota, ed è dunque necessario attivare i sensori ad ultrasuoni per riconoscere gli ostacoli e valutare il percorso ad ogni passo (ultimo scenario descritto).

1.3 Robot Lego NXT

Il robot Lego NXT in Figura 2 include gli strumenti e i sensori per affrontare l'esperienza negli scenari descritti in precedenza. Nel nostro caso, il robot è dotato di :

- **Due ruote motrici** azionate elettricamente, che permettono al robot di avanzare, retrocedere, e ruotare su se stesso.
- **Sensore ad ultrasuoni** per rilevare la presenza di ostacoli nella mappa, esso è montato su supporto che può ruotare di 360 gradi, grazie ad un apposito motore elettrico.
- **Sensore di luce** per rilevare il colore del terreno, utile per individuare quadrati (individuati da linee bianche) sulla matrice verde.
- **Odometria:** è uno strumento molto utile che il robot fornisce, con questo infatti esso, sommando i molteplici spostamenti relativi, tiene traccia dello spostamento globale effettuato, rispetto alla sua posizione di partenza. In questo modo è possibile sapere, istante per istante la posizione del robot e la rispettiva orientazione rispetto a quelle di partenza.



Figura 2: Mettere una foto con due ruote !!!

2 Realizzazione del Software

Come già accennato nella sezione introduttiva, l'implementazione degli algoritmi viene eseguita in C++, mentre l'interfacciamento con il robot viene fatto mediante la utility ROS. Quest'ultimo si basa sulla comunicazione di diversi nodi, che vengono creati in fase di implementazione del software. Dunque per lanciare il software si è innanzitutto creato un file launcher (chiamato `esperienza1.launch`) che si occupa di avviare tre nodi principali che sono stati individuati, ovvero:

- **Robot:** si occupa della gestione dei movimenti del robot;
- **Mapping:** si occupa della creazione e della gestione della mappa;
- **Navigation:** si occupa dell'algoritmo di navigazione che porta il robot dallo start al goal.

Come si può intuire ciascun nodo per gestire il suo task, ha bisogno di informazioni che derivano da altri nodi, perciò questi ultimi necessitano di comunicare costantemente durante l'esecuzione. Tali comunicazioni verranno approfondite più avanti, ma avvengono attraverso servizi di diverso tipo, che vengono richiesti e forniti da ciascun nodo che, all'occorrenza farà da client o da server. La creazione e la gestione dei nodi è stata realizzata attraverso una classe dedicata per ciascun nodo. [AGGIUNGERE SCHEMA DI COMUNICAZIONE DEI NODI (RQT PLOT)]

2.1 Mappa (angelo)

Per la creazione della mappa è stata definita la classe “Cella” che contiene un intero che rappresenta la distanza dall’arrivo, un booleano che indica se la cella è occupata da un qualche ostacolo ed infine un altro intero che funge da contatore per la scomparsa automatica del blocco, nel caso i blocchi non siano fissi, così da evitare che la cella resti interdetta per sempre.

È stata poi creata la classe “Mappa” che è una matrice di celle della dimensione della mappa dove dovrà poi muoversi il robot. Una volta creata la matrice, e definita la cella arrivo, viene poi “popolata” cioè vengono assegnate le distanze alle varie celle: inizialmente si assegna a tutte le celle la distanza massima possibile (dette l ed a le dimensioni della matrice tale valore sarà $l \times a$), si attribuisce all’arrivo 0 e lo si inserisce in una coda dalla quale si estrae, finché ce n’è, il primo elemento e si assegna alle celle adiacenti a quest’ultimo una distanza pari a quella della cella in questione incrementata di un’unità e messe a loro volta nella coda (a patto che queste celle vicine abbiano un peso strettamente maggiore e non siano blocchi). Quando la coda è vuota l’intera mappa è stata popolata. Con questo procedimento non si hanno direzioni privilegiate di esplorazione della mappa ma si procede in maniera spiraliforme e si evita, con la diseguaglianza stretta, di incorrere in loop dovuti ad eventuali percorsi multipli.

Da un’analisi sull’algoritmo risulta una complessità di $O(n)$ con n il numero di celle della matrice mappa.

2.2 Navigazione (angelo)

La parte di navigazione è relativamente semplice sfruttando la mappa definita poc’anzi, basta infatti spostarsi verso la cella adiacente che ha il peso minore, a patto di non trovarsi già all’arrivo (ed, ovviamente, restando nella mappa).

Se la cella dove è previsto muoversi risulta occupata da un ostacolo di qualche tipo si provvede ad aggiornare la mappa ed a ricalcolarla.

Ogniqualvolta il movimento verso una nuova cella risulta compiuto viene decrementato il contatore di tutte le celle bloccate: questo serve perché se si imbocca un vicolo cieco e mentre si retrocede la strada viene bloccata, dopo un po’ si ritenteranno percorsi prima considerati bloccati.

Questo sistema di navigazione risulta essere molto efficiente poiché’ la scelta del percorso richiede, praticamente, solo un’operazione basilare e le operazioni più complesse di popolamento della mappa avvengono abbastanza di rado ma sono comunque nell’ordine di grandezza della mappa stessa.

Inoltre, a differenza di altri algoritmi tipo A*, questo sistema di esplorazione (sarebbe meglio dire popolamento) anche in caso di investigazione di una strada non ottimale non la esplora completamente

prima di accorgersi che è tale, nel caso peggiore arriverà a metà della stessa da due direzioni differenti. Va tenuto in considerazione però che A* nel caso ottimo è molto più veloce esplorando molte meno strade.

2.3 Spostamento del robot (matteo)

Per lo spostamento del robot all’interno della mappa si sono innanzitutto individuati i singoli movimenti necessari a farlo muovere correttamente all’interno dell’ambiente. Si sono isolate 5 azioni:

- Rotazione;
- Movimento in linea retta;
- Avanzamento di una cella;
- Centramento all’interno della cella;
- Riceca dei coni.

Si è scelto allora di creare una classe (*Navigate_Nxt*) che ci permettesse di accedere velocemente a tutti questi movimenti in modo da poterli comporre per far spostare il robot all’interno della mappa. La classe prende in ingresso un nodo ros su cui poi andrà a lavorare. Innanzitutto in fase di costruzione va ad iscrivere il nodo a tutti i topic necessari al controllo del robot. Sarà quindi iscritto come publisher ai topic `\cmd_vel` ed `\angle` per poter far muovere i motori delle ruote e del giunto centrale, e come subscriber ai topic `\odom`, `\intensity_sensor`, `\joint_states` e `\ultrasonic_sensor` per poter raccogliere i dati forniti dai sensori del robot.

2.3.1 Rotazione

Il movimento di rotazione del robot è stato implementato nel metodo *Navigate_Nxt::rotate(char direction)*. Con questo metodo il robot può cambiare il proprio orientamento nella mappa. Si è scelto di utilizzare un sistema di riferimento assoluto. È allora possibile comunicare al robot di girarsi allineandosi con le quattro direzioni parallele ai limiti della mappa: nord sud ovest ed est semplicemente dando il giusto ingresso al metodo. Grazie a

questa rappresentazione si semplifica di molto la parte di navigazione del robot, infatti una volta trovata la cella in cui spostarsi basta solo valutarne la posizione rispetto alle coordinate attuali del robot e dire a quest'ultimo di girarsi nella direzione esatta senza doversi preoccupare dell'orientamento attuale. Così facendo, inoltre, non si rende necessaria la conversione dell'informazione contenuta nel quaternione restituito dall'odometria del robot in un angolo vero e proprio.

Si è deciso di dividere il movimento di rotazione in due parti, una prima fase in cui il robot si porta vicino alla posizione desiderata ed una seconda fase in cui si effettua un aggiustamento della posizione in modo da ottenere un'orientazione finale più precisa. Per fare ciò si sono innanzitutto definiti i target di orientamento in base alla direzione verso cui ci si vuole allineare, non essendo il dato di orientamento lineare rispetto alla posizione in gradi del robot per ogni target si è definita anche la soglia di errore entro la quale si vuole stare. Si può allora dividere l'algoritmo in step:

1. In base all'ingresso della funzione si selezionano target e soglia desiderati; alla direzione sud sono associati i target ± 1 per risolvere l'ambiguità si sceglie il valore con segno concorde all'orientamento attuale.
2. Dal momento che il verso di rotazione è legato alla differenza tra target e orientamento attuale ci si deve preoccupare che nel caso di rotazione da sud verso est od ovest il valore di orientazione sia concorde al target, altrimenti il robot girerà nel verso in cui il movimento risulta più lungo. In questo caso si fa girare il robot fino a che l'orientamento non cambia segno. Il movimento in questa fase è comunque molto contenuto, essendo già orientato a sud si è molto vicini ai 180° e quindi al punto in cui l'orientamento cambia segno.
3. A questo punto si valuta la differenza tra target e orientamento del robot e si avvia la rotazione fino a che il valore assoluto del parametro differenza è minore di 0.1 o fino a che non cambia segno.
4. Ora il robot è vicino alla posizione desiderata, in modo analogo a prima, applicando però accelerazioni più basse, si aggiusta l'orientamento del robot fino a che il valore assoluto del parametro differenza non è minore del valore della soglia.

2.3.2 Movimento in linea retta

Quest'azione è implementata nel metodo *Navigate_Nxt::robotGoStraight(double length)*.

Questa funzione permette di far muovere il robot in linea retta per la lunghezza specificata in ingresso. Il risultato è ottenuto pubblicando una velocità lineare positiva fino a che il valore di distanza percorsa non è pari o superiore a quello desiderato.

2.3.3 Avanzamento di una cella

Implementata nel metodo *Navigate_Nxt::robotGo()*.

Per far avanzare il robot di una cella si è deciso di farlo avanzare linearmente fino a che non riesce a trovare la linea bianca che delimita le varie caselle, a quel punto si invoca robotGoStraight dando in ingresso la metà della lunghezza di una cella. In questo modo si riesce ad ovviare ad eventuali errori nei movimenti del robot, a patto non siano troppo elevati, infatti ogni qual volta si effettuano due movimenti perpendicolari del robot questo si ritroverà al centro della cella.

2.3.4 Centramento all'interno di una cella

Quest'azione è implementata nel metodo *Navigate_Nxt::center()*.

Con questo metodo è possibile ricentrare il robot nella cella in cui si trova. Se invocato il robot si allinea verso nord, avanza fino a vedere la linea bianca, torna indietro di mezza cella e ripete la stessa operazione per la direzione est, alla fine si riallinea con l'orientazione che aveva prima di invocare il metodo. Questo metodo è stato sviluppato per poter far fronte ad eventuali deviazioni non volute nel movimento del robot, si è comunque osservato che il solo metodo robotGo è sufficiente per raggiungere questo scopo.

2.3.5 Ricerca dei coni

È possibile invocare questa azione attraverso il metodo *Navigate_Nxt::findCones()*.

La funzione non richiede alcun ingresso e restituisce in uscita un vettore contenente 4 valori booleani che indicano la presenza o meno di un cono nelle posizioni sud, est, nord ed ovest rispetto alla cella attuale del robot, i valori sono inseriti nel vettore sempre in questo ordine così da rendere più semplice l'aggiunta degli ostacoli nella mappa.

Per avere una detection più robusta dell'ostacolo innanzitutto si attende che il sensore sia allineato con la direzione da ispezionare, si aspettano quindi altri 2s in quanto il sensore tende ad oscillare attorno alla posizione

desiderata, attendendo le oscillazioni diminuiscono, ed infine si esegue la ricerca per 1s, in quest'ultimo lasso di tempo si esaminano i valori di range ottenuti e si salva il più piccolo, se questo è minore di una certa soglia allora l'ostacolo viene settato.

3 Test in laboratorio

3.1 Descrizione esperienza

Una volta implementato il software secondo le logiche precedentemente descritte, è possibile passare alla fase successiva di test del programma, sul robot Lego-Nxt in laboratorio. Di seguito viene riportata l'esperienza svolta nell'ultimo scenario descritto, cioè con i parametri *DrawMap* e *TerzoPunto* settati ad 1; si ricorda che qui, il robot utilizzerà i sensori ad ultrasuoni per riconoscere gli ostacoli, e valuterà il suo percorso istante per istante, visualizzando su schermo i suoi progressi.

1. Si procede dunque posizionando il robot sulla cella di *Start*, con orientamento verso Nord (Figura 3a), dopodichè si avvia il file *esperienza1.launch*. La prima azione che il robot mette in pratica, è quella di cercare eventuale presenza di ostacoli nelle celle adiacenti alla sua posizione.

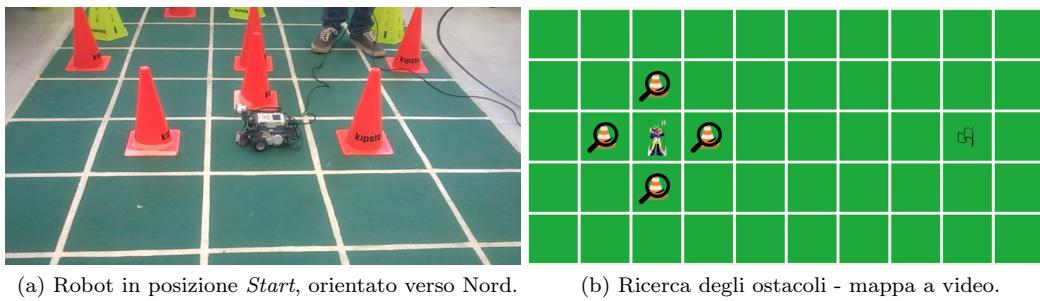


Figura 3: Robot in posizione iniziale.

2. A questo punto il robot trova che l'unica cella priva di ostacolo, è quella alla sua sinistra, quindi a video, il software visualizza la posizione degli ostacoli trovati all'interno della mappa (Figura 3b); ora il robot può ruotare in direzione Ovest, e procedere per avanzare nella cella successiva (Figura 4a).

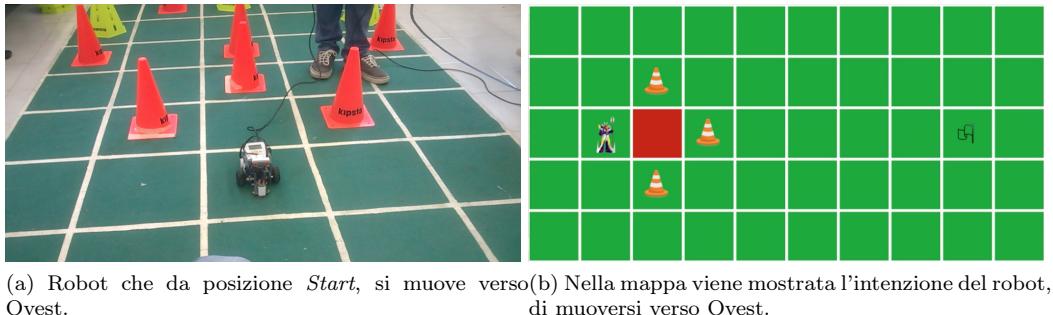


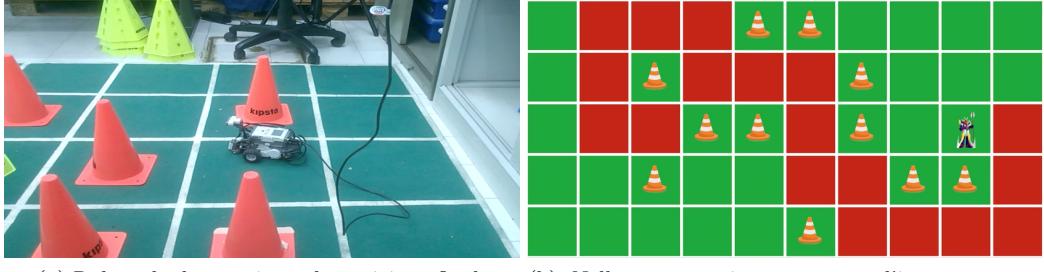
Figura 4: Robot in movimento.

3. Ora che il robot è giunto nella cella successiva, l'algoritmo ripete iterativamente le operazioni di ricerca ostacoli e calcolo del movimento successivo, finché il robot non raggiunge la cella *Goal*.

Alla fine dell'algoritmo iterativo, il robot è finalmente giunto nella cella finale (Figura 5a) e nella mappa grafica viene mostrato l'intero percorso valutato passo dopo passo dal robot (Figura 5b).

3.2 Scenario con ostacoli mobili

Abbiamo visto come l'algoritmo di navigazione in questa ultima sezione, sia robusto a variazioni sulla mappa, in quanto questo non richiede che la mappa sia a priori nota. Una ulteriore complicazione all'algoritmo è possibile aggiungerla facendo sì che la mappa cambi nel tempo. Supponiamo ad esempio che, ad un fissato istante temporale, il robot non trovi un percorso per arrivare al *Goal*. Con le ipotesi precedenti però, si prende



(a) Robot che ha raggiunto la posizione finale. (b) Nella mappa viene mostrato l'intero percorso svolto dal robot, con i relativi ostacoli individuati.

Figura 5: Robot in posizione finale.

in considerazione il fatto che, in un altro istante temporale successivo, la mappa cambi e si crei un percorso per arrivare alla fine. In queste ipotesi il robot è costretto, dopo un certo intervallo temporale, a ripetere l'operazione di ricerca degli ostacoli, per accertarsi di eventuali cambiamenti intercorsi. Di seguito dunque, si propone proprio tale scenario, in cui il robot viene chiuso all'interno di un percorso senza uscita, e ad un certo istante temporale, viene aperta una strada verso l'obiettivo finale. In Figura 6a, viene mostrato nella

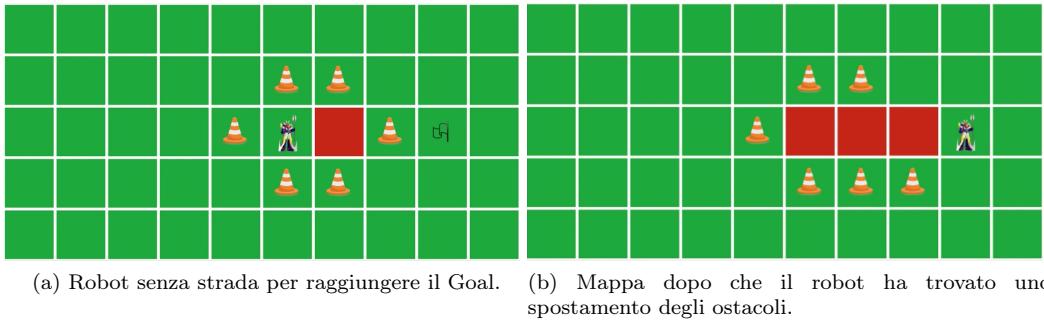


Figura 6: Mappa prima e dopo lo spostamento dell'ostacolo.

mappa come, all'inizio il robot continua a navigare all'interno di un percorso chiuso, e continua a trovare ostacoli nella stessa posizione di Prima. Figura 6b invece, mostra come, dopo lo spostamento di un ostacolo, il robot raggiunga ugualmente la cella finale.

4 Conclusioni

A conclusione di questa esperienza, si sono dunque raggiunti gli obiettivi di muovere il Lego-Nxt da una parte all'altra della mappa, utilizzando un'opportuno algoritmo per il calcolo del percorso. Quest'ultimo si basa sul sensore ad ultrasuoni, per il riconoscimento degli ostacoli, e sul sensore di luce per il riconoscimento dei quadrati nella griglia. Per la comunicazione con il robot Lego-Nxt si è fatto uso del software Ros, attivando attraverso un file launcher, 3 diversi nodi per la gestione della mappa, dei movimenti del robot, e dell'algoritmo di navigazione, ed è stato fatto uso dei topic di Ros per comunicare con i diversi sensori del robot. In seguito è stata data dimostrazione del funzionamento in laboratorio, in uno scenario in cui al robot venivano passate informazioni soltanto sulle celle di start, di goal, e sulle dimensioni della mappa, attraverso un file *.yaml*. Una volta che sono stati posizionati gli ostacoli in diversi punti della mappa, si è osservato come il robot abbia correttamente riconosciuto questi ultimi, e sia riuscito a trovare un percorso per arrivare alla cella *Goal*. In aggiunta, attraverso la libreria *OpenCV* è stata creata una mappa, in grado di mostrare iterativamente a video i progressi del robot.

Come ultima dimostrazione, si è voluto considerare il caso di mappa che cambia nel tempo, mettendo il robot in un circuito chiuso senza che abbia una possibilità di arrivare al goal. In questa situazione si è osservato come il robot, dopo non essere riuscito a trovare un percorso disponibile, continua a verificare se siano intercorsi cambiamenti nella mappa, anche tornando su strade già percorse. Infatti togliendo manualmente un ostacolo creando un percorso disponibile, in un certo istante di tempo, il robot se ne è accorto ed è potuto arrivare a destinazione.