POLITECNICO DI MILANO
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Sviluppo di un pianificatore di percorsi geometrici per una carrozzina autonoma

AI & R Lab
Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Relatore: Ing. Matteo Matteucci
Correlatore: Ing. Davide Migliore
Correlatore: Dott. Ing. Simone Ceriani

Tesina di Laurea di:
Marco Assini, matricola 675583

Anno Accademico 2008-2009

A tutti coloro che mi sono vicino

Sommario

L'area nella quale si svolge il lavoro è quello della robotica, ovvero la scienza e la tecnologia che studia macchine in grado di svolgere più o meno indipendentemente un lavoro o un compito che l'uomo giudica troppo faticoso, noioso o pericoloso. La robotica cerca di sviluppare le metodologie che permettano a una macchina (*robot*) di eseguire dei compiti specifici, utilizzando opportuni dispositivi atti a percepire l'ambiente circostante e a interagire con esso, quali sensori e attuatori.

Nella robotica confluiscono gli studi di molte discipline, sia di natura umanistica (come biologia, fisiologia, linguistica, psicologia) sia di natura scientifica (come automazione, elettronica, fisica, informatica, matematica, meccanica). La sottoarea più specifica del lavoro è in questo caso l'informatica.

Lo scopo di questa tesina di laurea è la realizzazione di un pianificatore per una carrozzina elettrica automatizzata di ausilio a persone invalide. Questo pianificatore, conoscendo la mappa del luogo dove agisce il robot, deve produrre un percorso che la carrozzina è in grado di seguire e di eseguire per raggiungere una destinazione.

L'attività svolta comprende l'implementazione in linguaggio C/C++ del software necessario per svolgere questo compito e l'utilizzo di questo a bordo della carrozzina. La parte di codice calcola infatti un percorso formato da una lista di punti compatibili con le caratteristiche fisiche e dinamiche del robot in questione.

La conclusione del lavoro di Tesi ha portato al raggiungimento degli scopi sopra descritti e i risultati sono stati rispondenti alle aspettative sia nell'esecuzione *stand-alone* del software, sia in simulazione su computer, sia in fase di prova a bordo della carrozzina.

Ringraziamenti

Ringrazio innanzi tutto il Prof. Matteo Matteucci, senza il quale tutto quello che segue non avrebbe avuto luogo. A partire dalla straordinaria opportunità di inserirmi nell'AirLab che non è solo un laboratorio di lavoro in senso stretto, ma anche un luogo dove si viene a contatto con molte persone valide, abili, in cui si riesce a perseguire i propri obiettivi professionali pur vivendo in un clima amichevole di grande collaborazione. Tutto questo ha fatto in modo che io potessi vivere con più serenità gli ultimi mesi del mio percorso di studi.

Tra le persone presenti in AirLab ringrazio soprattutto Simone, sempre disponibile nel dispensare preziosi consigli utili a superare gli ostacoli incontrati durante il percorso e nel mettere a mia disposizione il suo tempo e le sue capacità nelle concitate e incalzanti fasi finali del lavoro di tesi. Ringrazio il *capo* Davide, Martino con il suo *TiltOne* e i robot calciatori, il compagno di banco Paolo, i *pesci* Dario, Marco, Andrea, gli altri colleghi al lavoro sulla carrozzina Matteo e Mauro, gli studenti impegnati nei progetti della professoressa Gini, assegnisti di ricerca, i dottorandi e tutti i docenti che gravitano intorno al laboratorio: grazie a tutti per avermi dato la possibilità di apprendere tante, importanti e curiose nozioni in campi ed ambiti che vanno anche al di là dell'informatica, al di là della robotica, al di là dell'ingegneria. E questo è un bagaglio professionale prezioso nonché forte stimolo per affacciarmi con curiosità e interesse verso nuovi settori.

Ringrazio i miei genitori, Ettore e Carolina, per il grande sostegno e la pazienza avuti in tutti questi anni, ringrazio mia sorella Annalisa, mio cognato Giovanni, i miei nonni Rosa e Nicola, per essermi stati vicino.

E ringrazio la mia ragazza Antonella per essermi stata sempre accanto non solo nei momenti belli, ma anche e soprattutto in quelli più difficili e di sconforto, nei quali mi ha sopportato con grande pazienza

e supportato con grande entusiasmo, dandomi validi motivi per andare avanti; e non da meno per aver condiviso i momenti di rigenerazione.

Ringrazio infine i miei amici Maurizio, Ugo, Andrea, Matteo, Cristian, Alberto e Luca con i quali ho trascorso gli ugualmente utili momenti di pausa e intervallo dai momenti di studio. E ringrazio Gabriele, amico nonchè coinquilino con il quale per anni ho condiviso fatiche di studio quotidiano, preparazione di esami, qualche delusione ma anche gioie, grandi soddisfazioni e momenti di sano divertimento. E Mauro, Enrico, Alessio con i quali l'amicizia si è mantenuta anche quando le nostre strade universitarie si sono temporalmente separate.

Tutti dunque mi han regalato qualcosa e come in un puzzle tassello dopo tassello sono arrivato alla fine di questa tesi che lascerà posto all'inizio del mio futuro.

Indice

Sommario **I**

Ringraziamenti **III**

1 Introduzione **1**
1.1 Inquadramento generale 1
1.2 Breve descrizione del lavoro 2
1.3 Struttura della tesi 3

2 Stato dell'arte **4**
2.1 Classificazione dei pianificatori 4
2.1.1 Ricerca basata su roadmap 5
2.1.2 Scomposizione in celle 7
2.1.3 Campi potenziali 8
2.1.4 Metodi probabilistici 9

3 Impostazione del problema di ricerca **10**
3.1 Hardware: pc, sensori, componentistica a bordo della carrozzina 10
3.2 Obiettivo della ricerca 14
3.3 Definizioni e concetti preliminari 14
3.4 Problematiche affrontate 16

4 Tool MSL e Rapidly-exploring Random Tree **21**
4.1 Motion Strategy Library (MSL) 21
4.1.1 Guida a MSL 25
4.1.2 Creazione del problema nella Motion Strategy Library 26
4.2 Rapidly-exploring Random Tree (RRT) 30
4.2.1 Formulazione del problema 31

4.3 Algoritmo 33
4.4 Prove del tool MSL 39

5 Pianificatore basato su RRT e relativo software 48
5.1 Dal codice sorgente originario 48
5.2 Il modello di moto della carrozzina: stato e ingressi . . 49
5.3 RRTBidirBalancedIndip 52
5.4 Struttura e funzionamento del software 55

6 Realizzazioni sperimentali e valutazione 58

7 Conclusioni e sviluppi futuri 68

Bibliografia 72

Capitolo 1

Introduzione

> *"Non già nel seguire il sentiero battuto ma nel trovare a tentoni la propria strada, seguirla coraggiosamente, consiste la vera libertà."*
>
> Gandhi

1.1 Inquadramento generale

Il mio lavoro di tesi di laurea si inserisce all'interno del progetto LURCH (Let Unleashed Robot Crawl the House) [3] sviluppato presso il laboratorio di Intelligenza Artificiale e Robotica del Politecnico di Milano. Questo progetto ha per scopo la costruzione di una carrozzina elettrica autonoma: a una ordinaria carrozzina elettrica presente in commercio sono stati aggiunti nuovi dispositivi di comando e sono stati sviluppati comportamenti semi-autonomi (ovvero funzioni di assistenza alla guida come l'evitamento delle collisioni e degli ostacoli) e autonomi (ovvero l'esecuzione automatica di percorsi in ambienti noti).

Proprio in questo ultimo aspetto si è concentrato il mio lavoro di tesina. Precedentemente il compito di pianificare i percorsi che la carrozzina deve svolgere in modalità autonoma era svolto dal pianificatore *Spike* realizzato nel progetto FollowMe [14]. In questa precedente implementazione, il pianificatore non teneva conto in alcun modo né della cinematica del robot né del suo orientamento.

Il sistema è stato migliorato sostituendo al precedente pianificatore un adattamento dell'algoritmo *Rapidly-Exploring Random Tree* (RRT), sviluppato presso il Motion Strategy Laboratory, Dept. of Computer Science dell' University of Illinois.

La risoluzione del problema della pianificazione del moto dei robot (robot motion planning) è uno degli elementi fondamentali per la creazione di robot autonomi. L'obiettivo generale della pianificazione del moto dei robot è quello di trovare tecniche mediante le quali un robot possa autonomamente determinare un percorso da una posizione iniziale a una finale. Questa definizione del problema è del tutto generale ed è valida sia per i robot mobili sia i per manipolatori. Le condizioni aggiuntive che governano la ricerca di un percorso sono:

- Evitare urti con ostacoli presenti nello spazio di lavoro.
- Prevedere un margine di sicurezza nell'evitare gli ostacoli.
- Garantire la realizzabilità del percorso considerando i vincoli cinematici del robot.
- Ottenere un percorso il più possibile breve e rapido.

La pianificazione in oggetto si occupa solo del calcolo del percorso e non dell'inseguimento, rimandando al controllore del robot il compito di attuare i movimenti necessari ed eventualmente comunicare la necessità di una nuova pianificazione del percorso.

1.2 Breve descrizione del lavoro

La ricerca di un algoritmo di pianificazione che soddisfasse i requisiti richiesti ha portato all'individuazione dell'algoritmo RRT. Il mio lavoro è consistito nell'adattare l'algoritmo RRT, per poterlo usare come software di pianificazione della carrozzina realizzata nel progetto LURCH.

Il punto di partenza è stato il lavoro del MSL (*Motion Strategy Laboratory*) che ha realizzato un tool che consente un facile sviluppo e la sperimentazione di algoritmi di pianificazione del movimento per una vasta gamma di applicazioni. L'architettura software è orientata agli oggetti e l'impianto generale è altamente modulare. Il software è stato sviluppato su un sistema Linux che utilizza GNU C++, STL, e la FOX GUI Toolkit. Il primo passo è stato proprio quello di eliminare qualsiasi riferimento all'interfaccia grafica, visto che il compito del pianificatore è quello di fornire una traiettoria che colleghi il punto di partenza col punto di arrivo desiderato e di fornire il risultato al controllore e non quello di visualizzare la pianificazione sull'interfaccia grafica. La parte

grafica inoltre appesantiva molto l'algoritmo e di conseguenza i tempi di pianificazione venivano dilatati abbondantemente.

Per il suo corretto funzionamento, l'algoritmo RRT ha bisogno del modello preciso del robot per il quale deve trovare la pianificazione. Si è quindi creato il *Model* del problema, in termini di variabili di stato, variabili di ingresso ed equazioni di transizione per lo stato.

Infine nè l'algoritmo base RRT nè le sue varianti già implementate sono risultate adatte alla risoluzione del problema, quindi si è provveduto alla creazione di una nuova versione dell'algoritmo RRT creata appositamente per la pianificazione del percorso per la carrozzina autonoma.

1.3 Struttura della tesi

La tesi è strutturata nel modo seguente:

- nel capitolo due si mostra lo stato dell'arte del settore e un inquadramento dell'area di lavoro;
- nel capitolo tre si illustrano l'impostazione del problema di ricerca, le problematiche affrontate e le definizioni preliminari;
- nel capitolo quattro si illustra la MSL e in particolare l'algoritmo scelto: RRT (Rapidly-exploring Random Tree)
- nel capitolo cinque si mostra il modello di moto della carrozzina, la versione di RRT creata e il funzionamento del software.
- nel capitolo sei si mostra il progetto dal punto di vista sperimentale e si spiegano i risultati ottenuti attraverso una loro valutazione critica.
- infine nel capitolo sette si mostrano le prospettive future di ricerca.

Capitolo 2

Stato dell'arte

"I computer sono incredibilmente veloci, accurati e stupidi. Gli uomini sono incredibilmente lenti, inaccurati e intelligenti. L'insieme dei due costituisce una forza incalcolabile."

Albert Einstein

Il problema della pianificazione dei percorsi [1] è stato largamente affrontato e molte soluzioni sono state proposte ([5], [10], [7], [8]). La pianificazione del moto ha molte applicazioni nel campo dell'intelligenza artificiale ([6]) e della robotica: robot autonomi, automazione [13], ecc. ecc. Un problema base della pianificazione del moto consiste nel produrre una traiettoria continua o un percorso che connette una configurazione di partenza (*start*) con una configurazione di arrivo (*goal*) evitando collisioni con ostacoli noti.

In questo capitolo si riporta lo stato dell'arte del settore e un inquadramento dell'area di ricerca della problematica affrontata.

2.1 Classificazione dei pianificatori

Introducendo la definizione di *Configuration Space* C come l'insieme di tutte le possibili configurazioni che descrivono la posizione del robot e di C_{free} come sottoinsieme di C in cui le configurazioni non sono in collisione con gli ostacoli, i pianificatori di percorsi e di traiettorie si dividono in quattro grandi categorie:

- basati su una roadmap (mappa stradale) dello spazio di configurazione libero (C_{free});

- basati su una decomposizione in celle di C_{free};
- basati sull'uso di potenziali artificiali;
- probabilistici (randomizzati) per elevate dimensioni.

In quest'ultima categoria rientra l'RRT (Rapidly-exploring Random Tree), che è stato scelto come algoritmo con le caratteritiche più adatte al problema in questione, quindi verrà spiegato più avanti nella sezione 4.2 nella pagina 30.

Tutti questi metodi sono stati proposti in origine per risolvere il problema base della pianificazione, in particolare per problemi in R^2 e oggetti poligonali. La valutazione dei metodi si basa sulle seguenti caratteristiche:

- completezza: terminazione dell'algoritmo con una soluzione (se esiste) o riportando il fallimento
- complessità temporale / spaziale e valutazione delle prestazioni in termini di tempo in cui l'algoritmo arriva a terminazione e risorse utilizzate per implementarlo
- flessibilità: possibilitá di estendere l'algoritmo a situazioni più complesse, quali spazio degli stati R^3 o robot articolati.

2.1.1 Ricerca basata su roadmap

I pianificatori che si basano su una roadmap (Figura 2.1) dello spazio di configurazione libero C_{free} sono di due tipi: quelli che si fondano su griglia o grafo e quelli che si fondano sui diagrammi di Voronoi. Gli approcci basati su grafo sovrappongono una griglia al *Configuration Space* e richiedono come ipotesi che ciascuna configurazione sia identificata con un punto sulla griglia dal quale il robot può muoversi verso i punti adiacenti nella griglia stessa se la linea che li collega é completamente contenuta nello spazio libero. Questo metodo trasforma l'insieme delle operazioni da continue nella loro controparte discreta in modo tale che algoritmi di ricerca su grafo come A* riescano a trovare un percorso da *Start* a *Goal*.

Questi approcci richiedono la scelta di una risoluzione opportuna per la griglia. La ricerca è più veloce con griglie grossolane, ma l'algoritmo non riesce a trovare traiettorie attraverso passaggi stretti. Inoltre, il numero di punti sulla griglia cresce in modo esponenziale con la

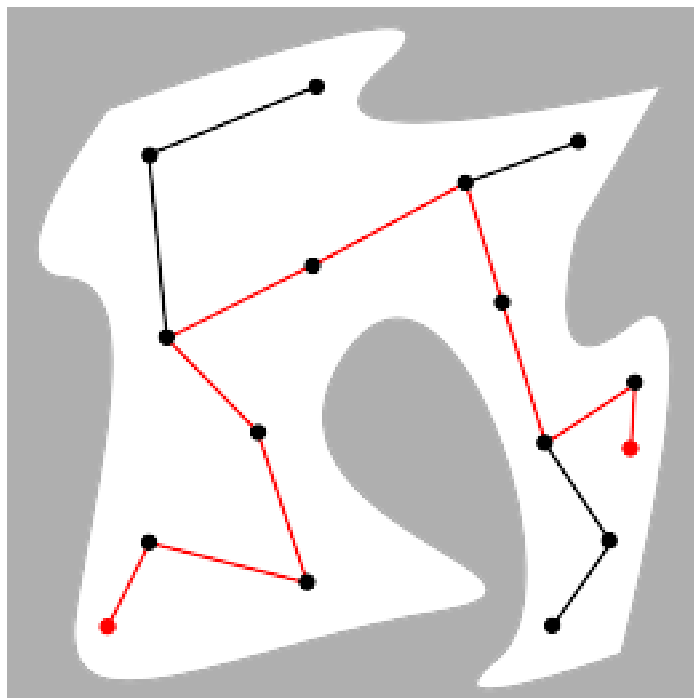


Figura 2.1: Esempio di roadmap

dimensione del *Configuration Space*, e per questo motivo tale approccio risulta inadatto per problemi in più dimensioni.

Gli approcci tradizionali alla ricerca basata su griglia producono percorsi che hanno angoli di sterzata vincolati a essere multipli di un dato angolo base e per questo motivo spesso portano a percorsi non ottimali.

Altri approcci migliorano il risultato propagando informazioni lungo i nodi della griglia. Un esempio di pianificatore che si basa su questa tipologia di algoritmo è *Spike* [14]. In questo caso la suddivisione della mappa in celle per la ricerca del percorso è effettuata con l'uso di zone quadrate, il cui lato può essere impostato. Le celle sono collegate tra loro in base all'adiacenza dei lati che le costituiscono. La ricerca di un cammino da un punto di partenza a una destinazione è effettuata con il noto algoritmo $A\star$, che permette di calcolare percorsi tra celle non occupate da ostacoli.

L'altro approccio si fonda sui diagrammi di Voronoi. Un diagramma di Voronoi (anche detto tassellatura di Voronoi) è un particolare tipo di decomposizione dello spazio determinato dalle distanze rispetto a un determinato insieme discreto di punti dello spazio. Nel caso lo spazio

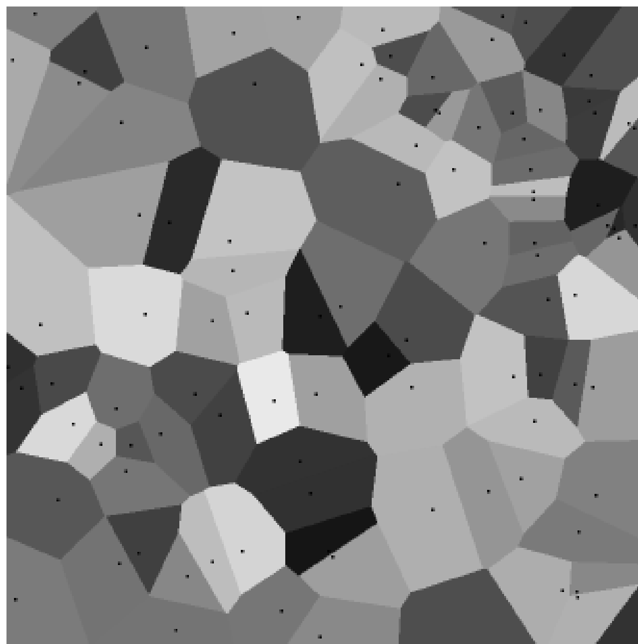


Figura 2.2: Diagramma di Voronoi in spazio a 2 dimensioni

preso in consderazione sia due dimensioni (Figura 2.2), dato un insieme finito di punti S, il diagramma di Voronoi per S è la partizione del piano che associa una regione $V(p)$ ad ogni punto $p \in S$ in modo tale che tutti i punti di $V(p)$ siano più vicini a p che ad ogni altro punto in S.

Nella ricerca del percorso si può sfruttare la struttura di un diagramma di Voronoi per scoprire il punto di S più vicino a un punto dato x senza calcolare a ogni richiesta la distanza di x da ogni elemento di S.

L'approccio basato sui diagrammi di Voronoi è utilizzato in molti ambiti della pianificazione ([12]), non solo per quanto riguarda i percorsi in senso stretto, ma anche per il movimento di bracci meccanici.

2.1.2 Scomposizione in celle

La suddivisione della mappa in celle di forme differenti, marcate come libere o occupate, permette di risolvere il problema della pianificazione con una ricerca su griglia. Con la scomposizione in celle non uniformi, la mappa viene partizionata in tante zone tali che ogni zona non si intersechi con nessun altra e l'unione dello spazio coperto dalle singole zone copra tutta la mappa.

In linea di principio la forma e le dimensioni delle celle possono essere di qualunque natura, ma per semplicità si utilizzano principalmente forme poligonali regolari. Ogni cella, oltre alle informazioni sulla sua posizione, dimensione, forma e connessioni con le celle adiacenti, mantiene anche uno stato che può assumere due valori: libera o occupata. Una cella è occupata quando il suo spazio è intersecato da uno o più ostacoli; e ovviamente una cella è libera quando non esistono intersezioni. La ricerca di un percorso da un punto di partenza ad un punto destinazione avverrà con una ricerca su grafo tra celle adiacenti libere.

È facile immaginare come il numero delle celle influenzi la rapidità di costruzione della mappa e il numero di strade da esplorare nella ricerca di un percorso. Più alto è il numero delle celle, più numerosi sono i controlli di intersezione con gli ostacoli da effettuare per valutare lo stato delle celle. Inoltre più numerose sono le celle, più il numero di passaggi intermedi che un percorso considera è elevato. Un numero troppo elevato di celle comporta quindi tempi di calcolo molto lunghi. Al contrario, un numero di celle troppo basso può rendere non connesse due zone della mappa che in realtà lo sono. Per ovviare a questo problema sono state sviluppate soluzioni che utilizzano celle di diverse dimensioni a seconda della zona della mappa che si considera: celle più piccole sono utili su zone più ostruite e dense di ostacoli e celle più grandi su zone completamente libere.

2.1.3 Campi potenziali

Un altro approccio è quello di trattare la configurazione del robot come un punto in un campo potenziale che combina attrazione verso il *Goal* e repulsione dagli ostacoli.

La forma del potenziale assegnato deve garantire la località, ovvero l'effetto repulsivo di un ostacolo deve esaurirsi a una certa distanza dall'ostacolo stesso. Anche il potenziale attrattivo gode della stessa proprietà, ma il suo decadimento deve essere più lento, per garantire che, anche partendo da un punto lontano nella mappa, l'attrazione verso il *goal* sia significativa. L'output della traiettoria risultante è il percorso con le caratteristiche desiderate: la funzione potenziale complessiva terrà conto di tutte le componenti repulsive e di quella attrattiva.

L'algoritmo che guida il movimento del robot è solitamente semplice e basato su metodi matematici noti di minimizzazione come la

discesa del gradiente. La traiettoria è quindi prodotta con poca computazione, ma il problema principale legato all'uso di metodi basati su campi potenziali è dato dai possibili minimi locali che si possono incontrare lungo il percorso. In queste situazioni l'algoritmo di discesa del gradiente termina la sua esecuzione, ma il punto raggiunto non è il minimo globale posto in corrispondenza del *goal*, cioè non viene generato un percorso valido.

2.1.4 Metodi probabilistici

L'idea alla base dei metodi probabilistici (come il Probabilistic Roadmap, PRM) é quella di prendere campioni casuali dal *Configuration Space* del robot, testare se questi sono nel *free space* e usare un pianificatore per tentare di connettere queste configurazioni ad altre configurazioni nelle vicinanze.

Le configurazioni di *start* e *goal* vengono aggiunte al grafo e tramite un algoritmo di ricerca basato su grafo viene determinato un percorso che connette la configurazione di start con quella di goal. Tipicamente il percorso è ottenuto tramite il *Dijkstra's shortest path*. Il *probabilistic Roadmap* si compone quindi di due fasi: nella prima si costruisce la roadmap (il grafo), nella seconda si effettua la ricerca. Ci sono molte varianti sul metodo di base PRM, alcune molto sofisticate, che variano la strategia di campionamento e la strategia di connessione per ottenere prestazioni più veloci.

Capitolo 3

Impostazione del problema di ricerca

"Arthur Dent: Marvin?
Marvin: Stavo parlando col computer di bordo.
Arthur Dent: E...
Marvin: Mi odia!"

Guida galattica per autostoppisti

In questa sezione si illustrano i concetti preliminari sia dal punto di vista dell'hardware sia dal punto di vista delle definizioni teoriche che stanno alla base delle problematiche affrontate.

3.1 Hardware: pc, sensori, componentistica a bordo della carrozzina

La carrozzina è già operativa e assolve tutte le funzionalità per poter essere usata da parte di un utente quali guida con diversi dispositivi di controllo, guida assistita, movimento autonomo. Per adempiere alla funzionalità di movimento autonomo precedentemente era utilizzato il pianificatore Spike, che non teneva conto della dinamica della carrozzina e del suo ingombro.

La carrozzina commerciale a disposizione per lo sviluppo del sistema è una *Rabbit*, prodotta dalla ditta tedesca *Otto Bock*. Come visibile in Figura 3.1, è una carrozzina adatta sia ad ambienti indoor sia outdoor in quanto presenta due ruote posteriori di dimensioni generose e con pneumatici tassellati. La trazione è realizzata con due motori

Figura 3.1: Otto Bock Rabbit

indipendenti che agiscono su ciascuna delle ruote posteriori, mentre le ruote anteriori sono libere e non comandabili: la direzione viene quindi imposta dalle ruote posteriori.

I motori assorbono una potenza di circa 200W ciascuno quando utilizzati a piena potenza. L'alimentazione è fornita da due batterie da $12V$ e $80Ah$ collegate in serie e poste sotto il sedile di guida. Il sistema di controllo dei motori, di comando con il joystick e l'insieme di tutte le componenti elettroniche presenti sulla carrozzina sono parte della linea *DX System* prodotta dalla *Dynamic Controls* .

La carrozzina è dotata di sensori di distanza, sistema di odometria,

Figura 3.2: Il computer di bordo "PCBrick", montato in una struttura in profilati di alluminio

sistema di localizzazione, computer di bordo, periferiche di input output e dispositivi di comando. Ai fini della pianificazione non vengono usati i sensori, in quanto la traiettoria è decisa basandosi sulla mappa degli ostacoli statici. L'effettivo inseguimento della traiettoria pianificata, invece, si basa sui dati provenienti dall'odometria e dal sistema di localizzazione.

Per raccogliere ed elaborare i dati sensoriali, produrre l'azione di controllo per comandare la carrozzina e pianificare il percorso è necessario disporre di un'unità di calcolo. Dovendo montare il computer a bordo della carrozzina, particolare attenzione deve essere posta nella scelta dell'hardware per quanto riguarda dimensioni e soprattutto consumo di corrente.

Per il progetto RAWSEEDS il Politecnico di Milano ha assemblato un computer x86 compatibile basato su una scheda VIA EN15000 che rispecchia lo standard mini-ITX (170x170mm). Come computer di bordo della carrozzina è stato assemblato presso il Laboratorio di Intelligenza Artificiale e Robotica lo stesso computer ed è stato montato su una struttura in profilati di alluminio. Per la componibilità che offre tale tipo di assemblaggio del pc gli è stato dato nome *PCBrick* (Figura 3.2). Grazie a un ampio dissipatore metallico, necessita solo di una

piccola ventola per il raffreddamento che risulta assolutamente silenziosa. Le interfacce messe a disposizione da questa scheda madre sono numerose, tra cui 6 porte USB 2.0, supporto per una porta FireWire, scheda Audio a 6 canali integrata, connettore Gigabit LAN Ethernet, video VGA e due porte PS/2. Il processore utilizzato è un VIA C7 con frequenza di clock da 1.5GHz. Alla scheda madre sono collegati 1 GB di ram DDR2 e un hard disk da 2.5 pollici da 80GB a 7200 giri al minuto. Il *PCBrick* è alimentato in corrente continua e, grazie a una scheda DC/DC, accetta tensioni di alimentazioni comprese tra 6V e 24V.

La potenza massima richiesta in condizioni di massimo carico del computer di bordo è di $25W$ sperimentalmente misurata. La scheda madre e tutti i componenti citati sono montati in una struttura di profilati di alluminio di dimensioni 225x195x135mm. Il sistema operativo installato su *PCBrick* è Linux, nella distribuzione Xubuntu 8.10 con interfaccia grafica Gnome.

Nell'ambito del progetto MRT (Milan Robocup Team) è stato dimostrato che è semplice e molto efficace utilizzare più *PCBrick* collegati in rete per gestire il calcolo in parallelo. A bordo della carrozzina sono presenti due *PcBrick* in modo da incrementare la potenza di calcolo. Il processore del secondo pc è un *Core Duo*, proprio per garantire che il software, specialmente nelle sue parti che riguardano la sicurezza dell'utente, non sia rallentato dalla mancanza di potenza.

Il sistema di localizzazione e posizionamento è fondamentale per il buon funzionamento dell'algoritmo di pianificazione. In particolare serve per garantire una buona inizializzazione del punto di partenza e per verificare la corretta esecuzione del piano calcolato.

Attualmente la localizzazione è basata sull'utilizzo di marker artificiali riconoscibili con sistemi di visione. Questi landmark sono tipicamente attaccati al soffitto e il sistema di visione è costituito da una telecamera rivolta verso l'alto. Questo sistema assicura una posizione assoluta immune da errori e che non diverge nel tempo, ma fornita a bassa frequenza. Al sistema basato su marker è stato abbinato un sistema odometrico basato su encoder che rilevano la rotazione delle ruote. Quest'ultimo sistema, al contrario del precedente, accumula errore, ma è a frequenza più alta.

La localizzazione è ottenuta quindi con un filtraggio attraverso un filtro di Kalman tra la posizione assoluta rilevata dal sistema basato

su landmark e il movimento stimato dall'odometria.

3.2 Obiettivo della ricerca

Lo scopo del presente lavoro è implementare un pianificatore di percorso per una carrozzina autonoma che tenga conto della dinamica del robot e del suo ingombro. La carrozzina infatti ha alcuni limiti nel movimento legati alla cinematica del robot stesso e semplificare il problema assumendola come un punto materiale sarebbe troppo riduttivo.

Quello che è stato realizzato è un pianificatore di percorso, non di traiettorie: determina un insieme di punti che realizza un collegamento tra *start* e *goal*; le coordinate di questi punti non sono considerate in funzione del tempo.

L'algoritmo di pianificazione in realtà tiene conto anche del tempo (visto che il percorso deve poter essere eseguito dalla carrozzina, e che il vincolo sull'angolo di sterzata dipende dalla velocità, quindi dal tempo), ma, a pianificazione avvenuta, il tempo non viene più considerato, rimandando all'esecutore il compito di gestire la traiettoria.

L'introduzione del nuovo pianificatore basato su RRT migliora il risultato e fornisce una traiettoria che la carrozzina può seguire tenendo conto dei suoi limiti. La struttura stessa dell'algoritmo RRT permette di risolvere molti problemi e di giungere ad una soluzione del problema di pianificazione. Prende infatti come input lo stato di partenza, lo stato di arrivo, la mappa degli ostacoli, la forma e l'ingombro del robot.

3.3 Definizioni e concetti preliminari

Un problema base della pianificazione consiste nel produrre un percorso che connetta una configurazione d'inizio S con una configurazione obiettivo G, evitando collisioni con ostacoli noti. In generale i robot e gli ostacoli sono descritti in un *workspace* 2D o 3D, mentre il percorso è rappresentato con la lista di punti nel *Configuration Space* che lo compone.

Una configurazione descrive la posizione del robot; e il *Configuration Space* C è l'insieme di tutte le possibili configurazioni. Per esempio, a seconda dei diversi gradi di libertà (3 DoF, 6 DoF):

- se il robot è un singolo punto (zero-dimensionale) che trasla in

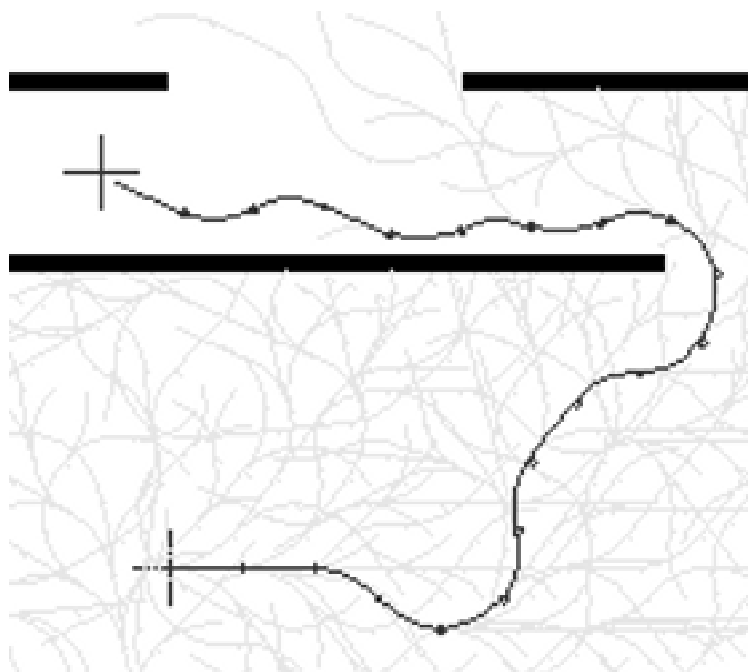


Figura 3.3: Pianificazione

un piano a 2 dimensioni, C è un piano e una configurazione può essere rappresenatata usando 2 parametri (x, y)

- se il robot ha una forma bidimensionale e può traslare e ruotare, il workspace è ancora bidimensionale, ma in questo per rappresentare la configurazione servono 3 parametri (x, y, θ).
- se il robot è tridimensionale e può traslare e ruotare, il workspace è tridimensionale e la configurazione necessita di 6 parametri per essere rappresentata: (x, y, z) per la traslazione e gli angoli $(\alpha,\ \beta,\ \gamma)$.

L'insieme di configurazioni che evita le collisioni con gli ostacoli è chiamato *Free space*, indicato con C_{free}. Il complemento del C_{free} in C è chiamato ostacolo o regione proibita. Spesso è molto difficile esprimere la forma di C_{free}, o comunqe testare se una data configurazione è in C_{free}. Per prima cosa la cinematica determina la posizione spaziale del robot, e l'evitamento degli ostacoli testa se la geometria del robot collide con la geometria dell'ambiente

3.4 Problematiche affrontate

Un elemento fondamentale per un algoritmo di plannig è il formato della mappa, che comprende la planimetria dello spazio in cui si deve muovere il robot e gli ostacoli statici presenti nell'ambiente, oltre che i punti che possono fungere da partenza o destinazione.

Il formato della mappa nel nostro caso deve anche sottostare ad alcune caratteristiche che la rendono usabile per la progettazione dell'interfaccia grafica realizzata per il progetto di guida tramite BCI (Brain Computer Interface) [4].

Per questi motivi la prima parte del lavoro di Tesi ha riguardato la ricerca di un formato appropriato per la mappa e solo in un secondo momento si è passati alla ricerca e allo sviluppo del pianificatore vero e proprio.

Sono stati proposti e studiati diversi tipi di formati: AutoCAD DXF, PostScript, Scalable Vector Graphics (SVG) e SVG Mobile fino a giungere al formato che si è rivelato più adatto, ovvero XML.

AutoCAD DXF (Drawing Interchange Format, o Drawing Exchange Format) è un formato per i file di tipo CAD, sviluppato da Autodesk come soluzione per scambiare dati tra il programma AutoCAD e altri programmi.

PostScript è un linguaggio di descrizione di pagina interpretato particolarmente adatto alla descrizione di pagine ed immagini, sviluppato da Adobe Systems ed inizialmente usato come linguaggio per il controllo delle stampanti.

SVG è l'acronimo di (Scalable Vector Graphics), è uno standard aperto e le sue specifiche sono definite e gestite dal World Wide Web Consortium (W3C).

SVG è una piattaforma per la gestione di formati grafici bidimensionali, costituita da una serie di specifiche che descrivono la componente grafica usando il fornato XML e da una serie di API per lo sviluppo di applicazioni. Le funzionalità chiave includono figure, testo, trame con differenti stili di disegno. Ha un supporto completo per le animazioni e per il linguaggio di scripting ECMAScript.

SVG è utilizzato in diversi settori che includono la grafica Web, le animazioni, le interfacce grafiche, la stampa e il disegno di alta qualità. Inoltre è indipendente e libero da royalty e vanta il supporto di numerose industrie come Adobe, Agfa, Apple, Canon, Corel, Ericsson, IBM, Kodak, Macromedia, Microsoft, Nokia, Sharp e Sun Microsys-

tems. I viewer SVG sono distribuiti su molti desktop, preinstallati sulla maggior parte delle distribuzioni Linux e lo standard è supportato da diversi produttori di tools di authoring.

SVG è disponibile anche per il mondo mobile, infatti nel 2001 l'industria ha scelto questo standard per le sue piattaforme grafiche. Molte aziende leader del settore hanno aderito al consorzio collaborando attivamente per lo sviluppo della piattaforma indirizzata a tutti i dispositivi mobili con limitate risorse e funzionalità, come telefoni cellulari e PDA. E' nata cosi la specifica SVG Mobile che raggruppa i profili delle piattaforme SVG Tiny e SVG Basic.

La specifica SVG Mobile è stata adottata da 3GPP, OMA (Open Mobile Alliance) e da altre organizzazioni, come standard grafico per i cellulari di nuova generazione. Questo formato è stato scartato in quanto la scelta di SVG sembrava avere più svantaggi che vantaggi.

Il formato della mappa che è stato ritenuto più adatto dopo l'analisi delle problematiche è risultato XML. Si è optato per un *xml custom* per il quale è disponibile il *parser*, la struttura può essere creata a piacere, e anche la visualizzazione su video risulta facilmente attuabile.

XML (acronimo di *eXtensible Markup Language*) è un metalinguaggio di markup, ovvero un linguaggio marcatore che definisce un meccanismo sintattico che consente di estendere o controllare il significato di altri linguaggi marcatori. Sono stati presi in considerazione anche i linguaggi aggiuntivi per l'XML come XSL e XSLT. XSL (acronimo di *eXtensible Stylesheet Language*): è il linguaggio con cui si descrive il foglio di stile di un documento XML. La sua versione estesa è l'XSLT (dove la T sta per *Transformations*).

La struttura XML della mappa è descritta nella Figura 7.1 nella pagina 70. In essa è descritto l'intero mondo in cui può muoversi la carrozzina e sono formalizzate tutte le caratteristiche utili alla pianificazione. Il mondo è formato da zone o portali (collegamenti tra zone diverse della mappa, con associato un parametro di costo). Le zone possono contenere ricorsivamente altre zone, o le mappa della zona stessa. Ogni mappa contiene una o più stanza. Ed ogni stanza può contenere ricorsivamente altre stanze.

Le informazioni che sono utili alla pianificazione sono incluse nella stanza: la polilinea che definisce il contorno, che definirà la mappa degli ostacoli per il pianificatore, e le *location* che possono fungere da punti di partenza o di arrivo per la pianificazione.

Per quanto riguarda la ricerca del planner, sono stati presi in considerazione $A*$, VFH, per giungere poi all'algoritmo prescelto, ovvero RRT.

L'algoritmo $A*$ è stato descritto nel 1968 da Peter Hart, Nils Nilsson, e Bertram Raphael. $A*$ è un algoritmo di ricerca su grafi che individua un percorso da un dato nodo iniziale verso un dato nodo *goal* (o che passi un test di *goal* dato). Utilizza una stima euristica che classifica ogni nodo attraverso una stima della strada migliore che passa attraverso tale nodo e la visita del nodo è effettuata in base a tale stima euristica secondo l'approccio *best-first*.

Un planner basato su $A*$ è indicato per percorsi semplici, realizzando bene la parte di evitamento ostacoli, ma risultando inadatto per la ricerca di percorsi complessi.

L'algoritmo *Vector Field Histogram* (VFH) è stato proposto da Borenstein e Koren [18] nel 1991. Propriamente si tratta di un algoritmo *real-time* di *collision avoidance*, ma non di planning, sebbene consenta di raggiungere un obiettivo a partire dalla posizione corrente. Questo metodo usa un'*histogram grid* su due dimensioni come modello del mondo, che viene continuamente aggiornato dalle misure che si ricevono dai sensori. Il metodo VFH impiega un processo a due passi per ottenere il comando da dare al robot. Nel primo passo l'*histogram grid* è ridotto a un *polar histogram* a una dimensione costruito intorno alla posizione momentanea del robot. Ogni cella dell'istogramma contiene un valore che rappresenta la densità polare di ostacoli in quella direzione. Nel secondo passo l'algoritmo seleziona il settore piú adatto tra tutti quelli dell'istogramma, cioé quello con la piú bassa densità di ostacoli, e il robot viene allineato con questa direzione.

Nel 1998 Ulrich, in collaborazione con lo stesso Borenstain, presentano una seconda stesura dell'algoritmo detto VFH+. Questo algoritmo è pensato in maniera specifica per applicazioni robotiche per guidare persone cieche. Nell'articolo [16] gli autori, infatti, hanno testato l'algoritmo su un robot mobile detto *GuideCane*, prototipo di un possibile strumento di supporto alle persone con problemi alla vista.

Il concetto del VFH+ è simile all'algoritmo originale VFH. L'input dell'algoritmo è una mappa dello spazio circostante il robot, chiamata *histogram grid*. Il metodo impiega quattro stadi di riduzione dei dati per ottenere una nuova direzione di movimento. Nei primi tre passi la

mappa bidimensionale viene ridotta a un'istogramma polare monodimensionale intorno alla zona di interesse. Nel quarto passo l'algoritmo seleziona la direzione preferibile basandosi sull'istogramma polare e una funzione di costo.

Una prima differenza di questo algoritmo rispetto a quello di partenza è che tiene esplicitamente in considerazione le dimensioni del robot, mentre il metodo base ne teneva conto tramite l'aggiunta di un filtro passa-basso scelto in modo da migliorare le prestazioni, ma, sfortunatamente, molte scelte risultavano problematiche quando il robot si trovava in prossimità di angoli. Gli autori hanno quindi pensato di introdurre la dimensione del robot sin dal primo passo. Ogni cella occupata da un ostacolo dovrebbe essere ingrossata di una quantità pari al raggio della dimensione del robot, ma in realtà hanno preferito privilegiare la *safety* allargandola di una distanza di sicurezza che il robot deve mantenere dagli ostacoli. Altra importante miglioria è stata introdotta nella fluidità del movimento, grazie al peso dato alla dinamica e ai vincoli cinematici del robot, fino a quel momento ignorati. Anche i cambi di direzione sono stati resi più adattabili alla realtà, essendo ipotizzati come archi di circonferenze anziché come istantanei in qualsiasi direzione.

L'ultimo sviluppo dell'algoritmo VFH, denominato $VFH*$ è stato proposto dagli stessi autori. La nuova versione è stata presentata nel 2000 durante l'*International Conference on Robotics and Automation* di San Francisco con un *paper* dal titolo *VFH*: Local Obstacle Avoidance with Look-Ahead Verification* ([17]). Questo metodo ha un buon comportamento nei casi in cui gli algoritmi di *collision avoidance* locali hanno problematiche. Il $VFH*$ verifica che una direzione candidata guidi il robot intorno gli ostacoli. La verifica è fatta utilizzando un algoritmo di ricerca $A*$ insieme ad una funzione di costo *ad hoc* e a funzione euristiche. Si capisce quindi che anche questo planner è maggiormente indicato per problemi di evitamento ostacoli, ma non riesce a produrre percorsi complessi.

Nel prossimo capitolo viene approfondito e presentato nel dettaglio l'algoritmo RRT, che è stato scelto come pianificatore. In breve, Rapidly-exploring Random Tree (RRT) è un algoritmo costruito per la ricerca efficiente in spazi non convessi ad alta dimensionalità. Dato un insieme di punti di via, RRT connette ogni punto nello spazio ai suoi "vicini più stretti" basandosi su una costruzione randomizzata di alberi. Questo

metodo può esser visto come un metodo Monte Carlo (metodo statistico non parametrico basato su un algoritmo che genera una serie di numeri tra loro incorrelati) per limitare l'area di ricerca in una regione di Voronoi molto estesa.

Capitolo 4

Tool MSL e Rapidly-exploring Random Tree

> *"Pensiero profondo: La risposta alla domanda fondamentale sulla vita, l'universo e tutto quanto è... 42. Sì, ci ho pensato attentamente è questa, 42. Certo sarebbe stato più semplice se avessi conosciuto la domanda.*
> *Loonquawl: Ma era LA domanda, la domanda fondamentale di tutto quanto!*
> *Pensiero profondo: Questa non è una domanda! Solo quando conoscerete la domanda comprenderete la risposta.*
> *Loonquawl: E dacci la domanda fondamentale.*
> *Pensiero profondo: Non posso, ma c'è qualcuno che può, un computer che calcolerà la domanda fondamentale. Un computer di tale e infinita complessità che la vita stessa farà parte della sua matrice operativa e voi assumerete nuove e più primitive forme ed entrerete nel computer a navigare i dieci milioni di anni di vita del suo programma. Io progetterò questo computer per voi e si chiamerà... [fine del documentario]"*
>
> Guida galattica per autostoppisti

In questo capitolo si descrive il tool sviluppato presso il *Motion Strategy Laboratory* in cui è inclusa l'implementazione dell'algoritmo RRT utilizzato per la pianificazione del percorso nel presente lavoro.

4.1 Motion Strategy Library (MSL)

Presso il *Motion Strategy Laboratory* (MSL) del dipartimento di *Computer Science* dell'Università dell'Illinois è stato sviluppato un interessante tool denominato anch'esso MSL, acronimo di *Motion Strategy*

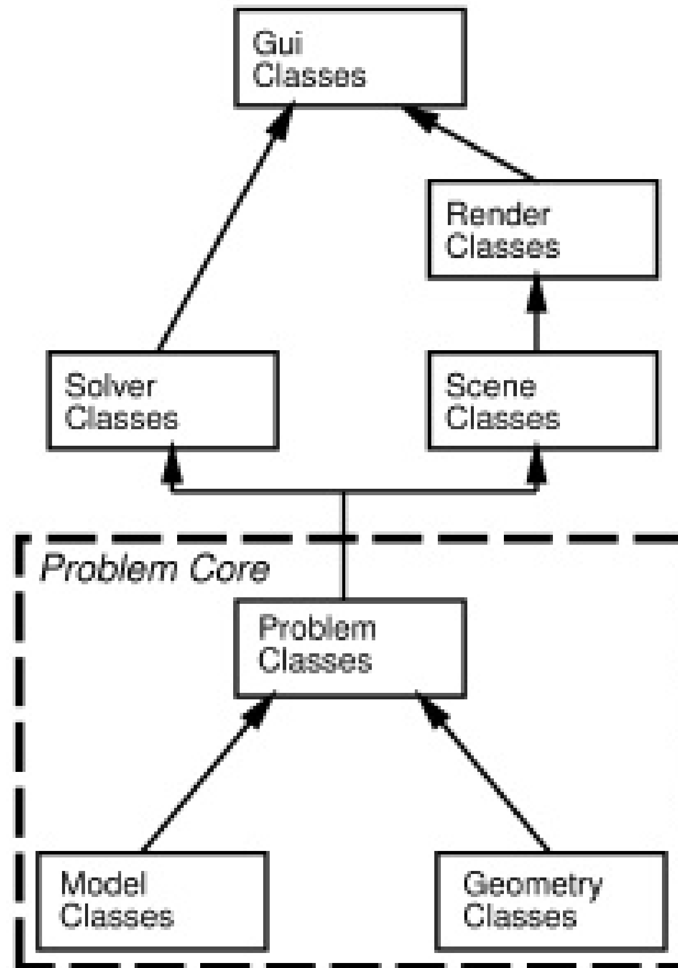


Figura 4.1: Relazioni funzionali delle classi che compongono MSL

Library, in cui è implementato anche l'algoritmo RRT [2]. Questo tool è nato in ambiente Linux, è scritto in C++ e la versione originale comprende una interfaccia grafica basata su *FOX* (un toolkit open source, cross-platform, C++ based per sviluppo di interfacce grafiche). La verisione attuale fornita dal laboratorio MSL, oltre a pianificatori basati su RRT, include Probabilistic Roadmaps (PRMs) (introdotti nel Paragrafo 2.1.4) e Forward Dynamic Programming (FDP) (basato sull'approccio di Barraquand, Latombe, Algorithmica, 1993, ma non trattato in questa Tesi).

La struttura portante della *Motion Strategy Library* è costituita da sette classi, le cui relazioni funzionali sono mostrate in Figura 4.1:

- *Model*: è la classe base che definisce la cinematica e la dinamica di un oggetto. Sono già stati sviluppati molti modelli per vari sistemi, e.g., semplici punti geometrici, automobili, autoarticolati, etc.

- *Geom*: definisce la rappresentazione geometrica degli oggetti, siano essi ostacoli presenti nell'ambiente o parti del robot. I metodi presenti consentono al pianificatore di determinare se alcune delle parti del robot sono in collisione tra loro o con ostacoli nell'ambiente.

- *Problem*: questa è una classe che funge da interfaccia per il pianificatore: astrae l'algoritmo di pianificazione da dettagli quali il rilevamento delle collisioni e dalla simulazione della dinamica. Ciascuna istanza del problema include sia una istanza di Model e di *Geom*. Uno stato finale e uno stato iniziale sono altresì inclusi. Queste informazioni consentono al risolutore (l'algoritmo di pianificazione) di risolvere il problema.

- *Solver*: definisce il risolutore nei suoi aspetti generali; i diversi tipi di pianificatore derivano in modo gerarchico da questa classe. Un *Solver* è inizializzato con un'istanza di *Problem*, e un metodo di ricerca della strategia del moto che risolve il problema.

- *Scene*: questa è una classe interfaccia che calcola le configurazioni di tutti i corpi in modo che siano visualizzabili dal render. *Scene* riceve la maggior parte delle informazioni che servono da *Problem*, ma include anche informazioni rilevanti per il rendering, come il punto di vista della telecamera.

- *Render*: questa classe contiene le differenti implementazioni per le richieste grafiche del render. Per esempio quando una interfaccia grafica (GUI) richiede che il percorso risultante dal planning sia animato, un metodo nella classe *Render* visualizza i corpi in movimento usando le configurazioni ottenute dalla classe *Scene*. Ciascuna classe derivata da *Render* corrisponde a differenti sistemi grafici. Attualmente ci sono render per SGI IRIS Performer, Open Inventor, Open GL. La flessibilità offerta da tali classi permette facili estensioni per altre librerie grafiche e altre piattaforme.

- *Gui*: La Graphic User Interface (GUI) è progettata come una gerarchia di classi che permettono di disegnare interfacce grafiche

specifiche per gli utenti per una gran varietà di problemi di strategia del moto e algoritmi di pianificazione. Attualmente, vi è una classe derivata che serve come interfaccia grafica per tutti i pianificatori basati su RRT. Ogni istanza di *Gui* include una istanza di un pianificatore RRT e una istanza di una classe *Render*. Questa progettazione grafica di base può essere utilizzata indipendentemente dalla particolare modalità di rendering.

Librerie usate dalla MSL

MSL è stata sviluppata su un pc con Red Hat Linux, ma versioni del software sono state installate con successo anche su molte altre distribuzioni di Linux, SGI, SunOS e Windows 2000.

Le seguenti librerie sono usate da MSL:

- *FOX C++ GUI Toolkit*: libreria gratuita per l'uso sotto licenza LGPL per creare interfacce grafiche.
- *Proximity Query Package* (PQP): questo pacchetto è stato sviluppato presso l'Università della Carolina del Nord, ed è gratuito per uso non commerciale. Svolge un'efficiente rilevamento delle collisioni e calcolo della distanza per un insieme di triangoli in un mondo 3D. MSL si sta predisponendo per avere la propria implementazione dell'algoritmi di rilevamento delle collisioni, e vi è la possibilità che la futura versione MSL non richiederà PQP.
- Open GL e GLUT: Open Graphics Library è una specifica che definisce una API per più linguaggi e per più piattaforme per scrivere applicazioni che producono computer grafica 2D e 3D. *OpenGL Utility Toolkit* è una libreria che semplifica l'accesso alle funzionalità di OpenGL. È anche possibile usare Mesa come implementazione open-source della specifica OpenGL. Le librerie necessarie possono essere ottenute gratuitamente su più piattaforme (ad esempio, sono incluse nelle distribuzioni RedHat Linux). In questo caso si è scelto di usare GLUT come implementazione di OpenGL. Se si sta usando Linux e si vogliono migliori prestazioni grafiche, si deve prestare particolare attenzione all'integrazione di una buona scheda grafica e di un buon driver Linux, GLX, e GL. Risultati migliori si sono ottenuti utilizzando le migliori schede nVidia, perché molti driver Linux sono disponibili sulla pagina

web di NVIDIA. Ulteriore utile supporto si ha dalle pagine web riguardanti OpenGL, GLUT e Mesa3D (*Mesa3D* è un'alternativa a *OpenGL*).

- Open Inventor: Open Inventor è un object oriented 3D toolkit progettato da SGI per fornire un livello superiore di programmazione per OpenGL, cioè è una libreria di oggetti e metodi per la creazione di applicazioni grafiche interattive tridimensionali, che fonisce risultati più accurati. Questa libreria è necessaria solo se si desidera utilizzare il render *Inventor-based* (al posto del *GL-based renderer*). *Open Inventor* è disponibile come Open Source sotto la licenza LGPL.

- OpenGL Performer: questa libreria è necessaria solo se si desidera utilizzare il *Performer-based renderer* per sviluppare grafica ad alte prestazioni e ad alto realismo. È inoltre necessaria, naturalmente, se si desidera utilizzare i modelli *Performer*. Al momento, Performer è disponibile per SIG (Silicon Graphics, Incorporated) ed è gratuito per Linux.

4.1.1 Guida a MSL

Diversi file README sono contenuti all'interno della distribuzione di MSL. Questi contengono informazioni utili per quanto riguarda l'installazione, l'esecuzione di esempi e lo sviluppo del proprio codice che utilizza la libreria.

Nella configurazione di default, vengono generati due file eseguibili, *plangl* e *planleda*. Quando si esegue uno di questi, deve essere specificato il percorso nel quale è contenuta la descrizione di un problema. Sono forniti molti problemi di esempio nella directory dei dati. Per eseguirne uno, bisogna ad esempio digitare da Terminale

```
plangl data/cage1
```

nel caso si voglia risolvere il problema denominato *cage1*.

La Figura 4.2 mostra l'interfaccia grafica della Motion Strategy Library . Questa interfaccia è comune a tutti i tipi di problema, quindi alcuni comandi sono utili solo in alcuni casi:

- bottone *Construct*: genera il grafo di pianificazione, ma non tenta di risolvere il problema. Serve solo per algoritmi di tipo PRM, mentre i planner RRT-based non richiedono questo passaggio;

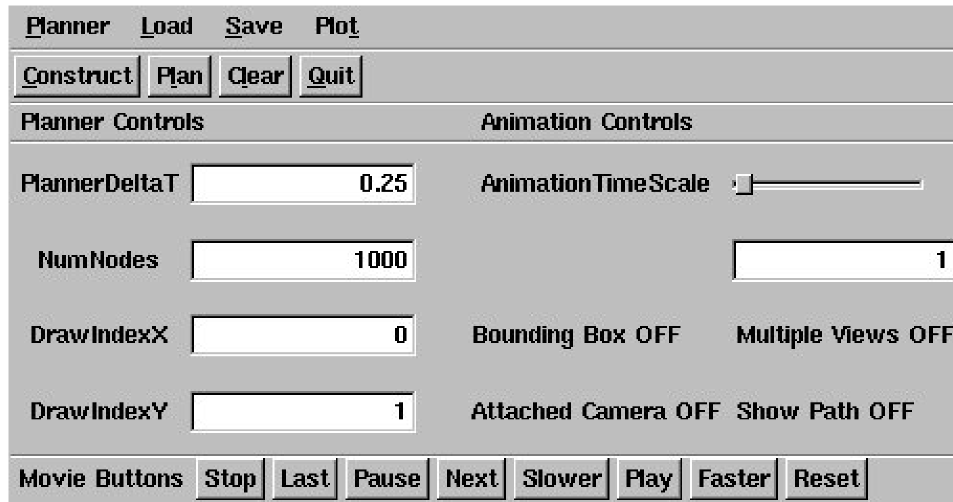


Figura 4.2: La finestra di GuiPlanner

- bottone *Plan*: tenta di trovare una percorso che collega *InitialState* con *GoalState*. Se la pianificazione fallisce, può essere nuovamente premuto e così facendo MSL continuerà a ricercare utilizzando il precedente grafo di pianificazione come punto di partenza;
- menu *Planner*: consente di selezionare diversi tipi di planner;
- altri parametri sono modificabili per fare in modo che la pianificazione sia visualizzata secondo le proprie preferenze.
- altre opzioni sono presenti e servon per leggere e scrivere file, e alcuni parametri di pianificazione, come DeltaT (lo step del tempo), NumNodes (numero di nodi da generare ogni volta che viene premuto Plan), e GapDist (la soglia di distanza da utilizzare per stabilire se, nell'algoritmo di ricerca a doppio albero, i due alberi possano essere connessi).

4.1.2 Creazione del problema nella Motion Strategy Library

Ciascun problema è descritto da un certo numero di file raggruppati in una directory. La maggior parte di essi sono file di testo che possono essere facilmente letti e scritti. Per creare un nuovo problema, alcuni file sono necessari, mentre altri sono opzionali. Valori di default dei parametri sono assunti quando i file opzionali non sono presenti. Alcuni

particolari modelli potrebbero aver bisogno di file che non sono richiesti in altri modelli.

Per creare un problema nella MSL, i seguenti file sono necessari:

- *GeomDim*: la dimensione dell'ambiente (2 o 3)
- *ModelXXX*: un file denominato esattamente come il modello da utilizzare. Esempi sono *Model2DRigid* e *Model3DRigidMulti*. In questo file si impostano i paramentri caratteristici del robot, come, ad esempio, l'angolo di sterzata che il robot è in grado di eseguire (*MaxSteeringAngle*).
- GeomXXX: un file denominato esattamente come la geometria del modello, come *GeomPQP3DRigid* per un corpo rigido fatto di triangoli in un mondo in 3D (con PQP utilizzato per il rilevamento delle collisioni).
- IntialState: lo stato iniziale per il problema.
- GoalState: la meta desiderata.
- Robot: un modello di robot, specificato tramite un elenco di poligoni se GeomDim = 2, o una lista di triangoli 3D se GeomDim = 3. In caso di problemi che coinvolgono un numero k di robot, questi sono chiamati $Robot_0, Robot_1, \ldots, Robot_k$.

I seguenti file sono facoltativi:

- *ModelDeltaT*: l'incremento del tempo da usare per l'equazione della transizione dello stato (equazioni del moto).
- *PlannerDeltaT*: l'incremento di tempo da usare per costituire un passo di pianificazione.
- *Obst*: un elenco di ostacoli statici, specificati come un elenco di poligoni, se GeomDim = 2, o una lista di triangoli 3D se GeomDim = 3.
- *EnvList*: un elenco di nomi di file che corrispondono a modelli geometrici statici che possono essere caricati e inclusi nel render; questi modelli non vengono utilizzati per il rilevamento delle collisioni. Se EnvList non esiste, allora Obst viene caricato e incluso nel render.

- *BodyList*: un elenco di nomi di file che corrispondono ai corpi mobili. Se BodyList non esiste, allora *Robot* (oppure $Robot_0, Robot_1, \ldots, Robot_k$) viene caricato.
- *LowerState*: limite inferiore dello stato. Ogni elemento è il valore più piccolo per ogni elemento della variabile di stato.
- *UpperState*: limite superiore dello stato. Ogni elemento è il più grande valore per la variabile di stato. Insieme a LowerState definisce il Configuration State del problema.
- *RRTXXX*: selezione del planner con cui si risolve il problema, come *RRTExtExt* o *RRTConCon*.
- *Inputs*: una lista di vettori di ingresso che devono essere applicati all'equazione della transizione di stato. Con questo file, si possono ignorare le impostazioni predefinite dalla classe *Construct*. Per esempio, una macchina che può andare avanti o all'indietro può essere convertita in una macchina che si muove solo in avanti semplicemente modificando questo file.
- *ViewingPosition*: la posizione della camera da utilizzare per il rendering.
- *ViewingDirection*: l'orientamento della camera da utilizzare per il rendering.
- *ViewingOrientation*: la rotazione della direzione per la fotocamera.
- *LowerWorld*: limite inferiore dell'ambiente descritto.
- *UpperWorld*: limite superiore dell'ambiente descritto.
- *Holonomic*: permette al planner di sapere se il problema è da risolvere assumendo che i vincoli di movimento siano solo olonomi.

Il modello e la geometria sono definiti tramite due file che risiedono nella stessa cartella del programma. Per esempio come Modelli 2D possiamo scegliere tra Point, PointCar, Rigid, RigidCar, RigidCarForward, RigidCarSmooth, RigidCarSmoothTrailer, RigidCarSmooth2Trailers,

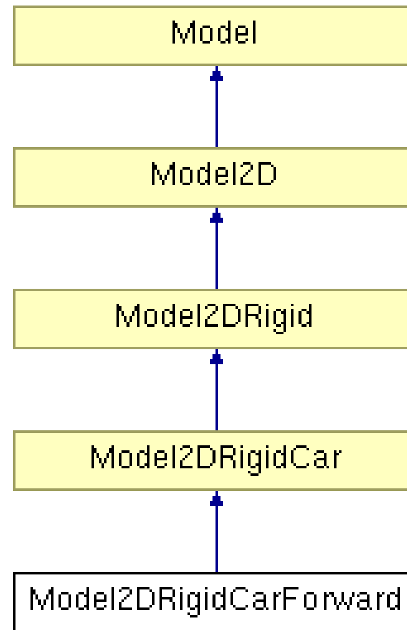


Figura 4.3: Diagramma di ereditarietà per la classe Model2dRigidCarForward

RigidCarSmooth3Trailers, RigidDyncar, RigidDyncarNtire, RigidMulti, RigidChain; o come modelli 3D possiamo scegliere tra Rigid, RigidMulti, RigidChain, RigidTree o modelli che caratterizzano più precisamente il robot come Car, CarSmooth, CarDyn, CarDynNtire, CarDynRollover, CarDynSmoothRollover.

Allo stesso modo, come *Geom* si può scegliere tra PQP2DRigid (corpo rigido 2D), PQP2DRigidMulti (collezione di corpi rigidi 2D), PQP3DRigid (corpo rigido 3D), PQP3DRigidMulti (collezione di corpi rigidi 3D).

All'interno della *Motion Strategy Library* non esiste un modello adatto per descrivere il moto della carrozzina. Per le prove preliminari si è scelto di usare il modello *Model2DRigidCarForward*, il cui diagramma di ereditarietà è mostrato nella Figura 4.3, ma in seguito abbiamo dovuto implementare un nuovo modello che eredita da *Model2DRigid* nel quale la cinematica è stata descritta in modo più aderente alla caratteristiche della carrozzina. A questo proposito si veda il paragrafo 5.2 nella pagina 49, dove sono descritte anche le equazioni sottostanti a tale modello.

All'interno del file *planner.c* è impostato un altro parametro fondamentale per la pianificazione: *NumNodes*, che definisce il numero di nodi che si generano ogni volta che è richiesta una pianificazione.

Nel frammento di codice che segue (contenuto della classe *Problem*) si vede come viene letto il parametro *InitialState*

```
READ_PARAMETER_OR_DEFAULT(InitialState,M->LowerState + 0.5*(M->
 UpperState - M->LowerState));
```

con riferimento alla *macro* contenuta nel defs.h

```
#define READ_PARAMETER_OR_DEFAULT(F,D)
_msl_fin.clear();
_msl_fin.open((FilePath+""#F"").c_str());
if (_msl_fin) {_msl_fin >> F;}
else F = D;
_msl_fin.close();
```

e allo stessto modo per *GoalState*

```
READ_PARAMETER_OR_DEFAULT(GoalState,M->LowerState);
```

Anche quest'ultima caratteristica è stata modificata in modo da rendere possibile il settaggio dei parametri di *InitialState* e *GoalState* da codice, come descritto nel Paragrafo 5.4 nella pagina 55.

4.2 Rapidly-exploring Random Tree (RRT)

Rapidly-exploring Random Tree (RRT) è una struttura dati e un algoritmo per la ricerca di pianificazioni in uno spazio multi dimensionale. Rapidly-exploring Random Tree è stato proposto e studiato da Steve LaValle a partire dal 1998 [9] [15].

L'algoritmo si basa sulla generazione di un albero, che si espande di nodo in nodo in modo casuale (come in Figura 4.4), ma sottostando ad alcuni vincoli che possono essere imposti in base al modello e alla geometria del problema. In parole semplici, l'albero è costruito in modo tale che ogni campione nello spazio è aggiunto collegandolo al più vicino campione già presente nell'albero.

I vincoli tenuti in considerazione per la generazione dell'albero riguardano le dimensioni del robot (descritte tramite un file che ne descrive la forma) e la cinematica dello stesso. Tramite RRT con questi due vincoli è facile verificare che vengono trovate buone pianificazioni

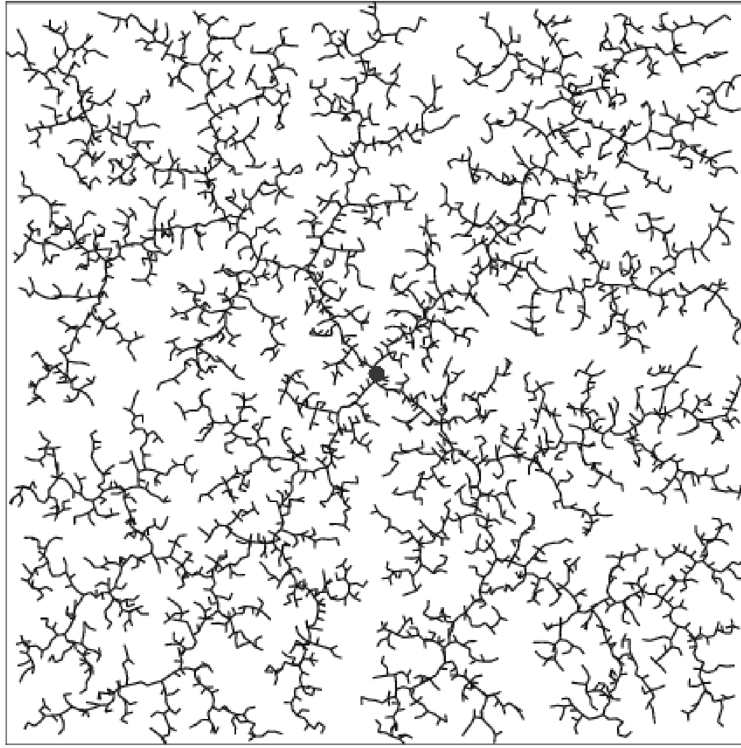


Figura 4.4: Rapidly-exploring Random Tree

anche in situazioni non convenzionali, quali un'automobile che sterza solo a sinistra (Figura 4.5) o lo spostamento di un oggetto ingombrante e dalla forma irregolare come un pianoforte.

I *Rapidly-exploring random tree* sono costruiti in modo incrementale in modo che si riduca rapidamente la distanza tra un punto scelto in modo casuale e l'albero. RRTs sono particolarmente adatti per problemi di pianificazione che comprendono ostacoli e vincoli differenziali (nonholonomic o kinodynamic). RRTs può essere considerato come una tecnica per la creazione di traiettorie in anello aperto per sistemi non lineari con vincoli di stato. RRT svolge anche il compito di esplorare l'ambiente e, intuitivamente, può essere considerato come un metodo di Monte-Carlo per guidare la ricerca in larghe regioni di Voronoi.

4.2.1 Formulazione del problema

I problemi risolvibili tramite RRT possono essere formulati nei termini dei seguenti sei componenti:

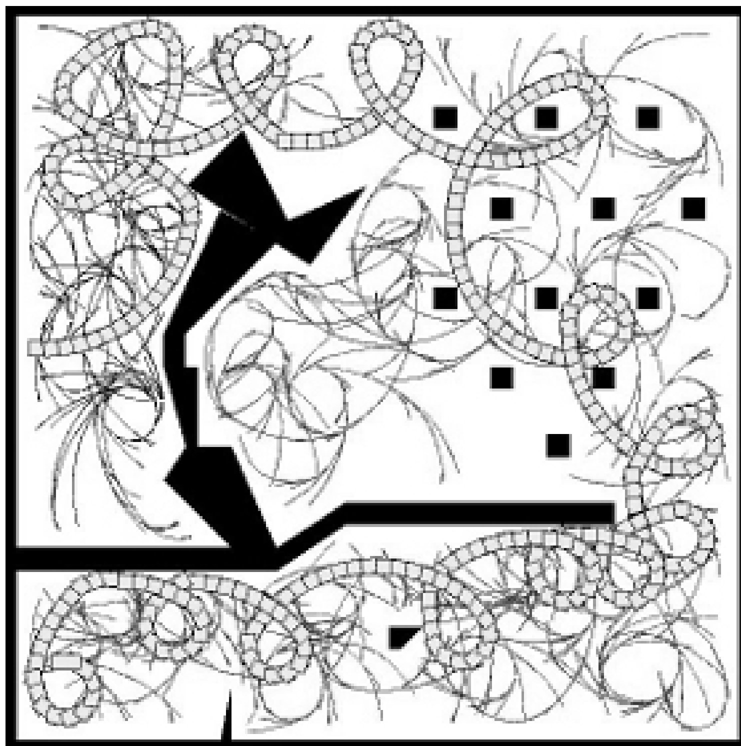


Figura 4.5: Pianificazione per un robot capace di sterzare solo a sinistra

- Spazio degli Stati: uno spazio topologico, X
- Valori ai bordi: $x_{init} \in X$ e $X_{goal} \subset X$.
- Rilevamento delle collisioni: una funzione $D : X \rightarrow \{true, false\}$ che determina se i vincoli globali sono rispettati dallo stato x
- Input: un insieme U che specifica completamente l'insieme dei controlli e delle azioni che caratterizzano uno stato
- Equazioni di transizione di Stato: dato lo stato corrente $x(t)$ e gli ingressi applicati in un intervallo di tempo $\{u(t') \mid t \leq t' < t+\Delta t\}$ calcola $x(t + \Delta t)$
- Metrica: una funziona reale $\rho : X \times X \rightarrow [0, \infty)$ che calcola la distanza tra una coppia di vettori di descrizione dello stato in X.

4.3 Algoritmo

Per un *Configuration Space C* l'algoritmo RRT è descritto in pseudocodice nel riquadro seguente. Nell'algoritmo mostrato, la funzione

```
Algorithm BuildRRT
 Input: Initial configuration qinit, number of vertices in RRT
  K, incremental distance delta_q)
 Output: RRT graph G

 G.init(qinit)
 for k = 1 to K
    qrand = RAND_CONF()
    qnear = NEAREST_VERTEX(qrand, G)
    qnew = NEW_CONF(qnear, delta_q)
    G.add_vertex(qnew)
    G.add_edge(qnear, qnew)
 return G
```

"RAND_CONF" prende una configurazione casuale q_{rand} in C. In linea teorica, questo può essere sostituito con una funzione denominata "RAND_FREE_CONF" che utilizza campioni in C_{free}, al posto di scartare quelle in C_{obs} utilizzando alcuni algoritmi di rilevamento delle collisioni.

"NEAREST_VERTEX" è una funzione che percorre tutti i vertici v nel grafico G, calcola la distanza tra q_{rand} e v secondo la metrica utilizzata così da restituire il vettore più vicino.

"NEW_CONF" seleziona una nuova configurazione q_{new} spostando una distanza incrementale $delta_q$ da q_{near} in direzione di q_{rand}.

La Figura 4.6 mostra la crescita dell'albero RRT dopo 45 e 390 iterazioni.

Come spiegato da Steve La Valle in [15], l'idea alla base di RRT è quella di guidare l'esplorazione verso aree inesplorate dello spazio, campionando punti nello spazio degli stati e cercando di attirare l'albero di ricerca verso questi punti. La Figura 4.7 descrive la base dell'algoritmo per la costruzione dell'RRT. Una singola iterazione viene eseguita a ogni passo in cui si tenta di estendere l'albero RRT, aggiungendo un nuovo vertice. La funzione EXTEND, mostrata in Figura 4.8, seleziona il nodo più vicino al nodo campionato fra quelli già presenti nell'albero, dove per "vicino" si intende vicino secondo la metrica scelta ρ. La fun-

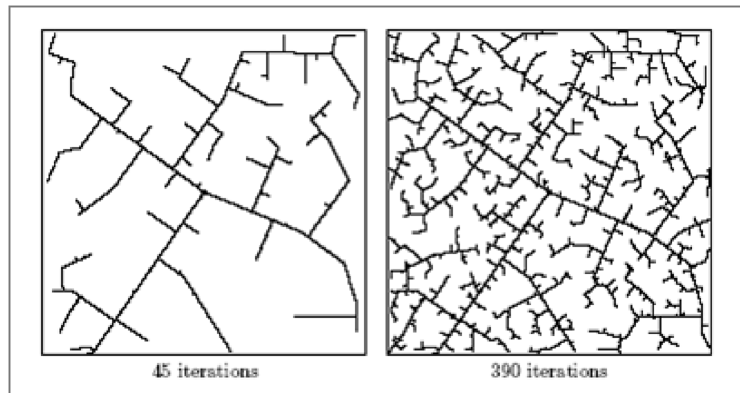


Figura 4.6: RRT dopo 45 e 390 iterazioni

zione NEW_STATE esegue una mossa nella direzione di x applicando un input $u \in U$ per un certo incremento di tempo Δt.

Questo input può essere scelto in modo casuale o selezionato provando tutti i possibili input e scegliendo quello che produce un nuovo stato più vicino possibile al sample x. NEW_STATE inoltre usa implicitamente la funzione di evitamento ostacoli per determinare se il nuovo stato soddisfa i vincoli globali. Se NEW_STATE va a buon fine, il nuovo nodo e l'input sono rappresentati in x_{new} e u_{new} rispettivamente. Tre situazioni possono ora accadere:

- Reached: il nuovo nodo raggiunge il sample x
- Advenced: il nuovo vertice x_{new} diverso da x è aggiunto all'albero RRT
- Trapped: NEW_STATE fallisce nel trovare uno stato giacente in X_{free}

La probabilità che un nodo sia selezionato per l'estensione è proporzionale all'estenzione della sua area di Voronoi, così come mostrato in Figura 4.17 nella pagina 43). Questo porta l'albero RRT a esplorare rapidamente lo spazio. Si nota inoltre che RRT arriva a coprire uniformemente lo spazio da esplorare.

Un altro problema da considerare è la lunghezza del passo usato per la costruzione dell'albero. Questo dovrebbe essere scelto dinamicamente durante l'esecuzione sulla base della funzione di calcolo della distanza usata per il rilevamento delle collisioni. Se i corpi sono lontano dal collidere, allora passi più grandi possono essere scelti. A parte il

```
BUILD_RRT(x_init)
 1  T.init(x_init);
 2  for k = 1 to K do
 3      x_rand ← RANDOM_STATE();
 4      EXTEND(T, x_rand);
 5  Return T
```

```
EXTEND(T, x)
 1  x_near ← NEAREST_NEIGHBOR(x, T);
 2  if NEW_STATE(x, x_near, x_new, u_new) then
 3      T.add_vertex(x_new);
 4      T.add_edge(x_near, x_new, u_new);
 5      if x_new = x then
 6          Return Reached;
 7      else
 8          Return Advanced;
 9  Return Trapped;
```

Figura 4.7: Algoritmo alla base della costruzione dell'albero RRT

seguire questa indicazione per calcolare il passo, quanto lontano x_{new} dovrà apparire da x_{near}? Invece di tentare di estendere l'albero RRT con un passo incrementale, EXTEND può essere iterata finchè il random state o un ostacolo siano raggiunti, come mostrato nell'algoritmo CONNECT descritto in Figura 4.9.

CONNECT può sostituire EXTEND producendo un albero RRT che cresce molto velocemente, se ciò è reso possibile dai vincoli della *collision detection*. Uno dei vantaggi fondamentali della funzione CONNECT è quello che permette di costruire lunghi percorsi con solo una chiamata all'algoritmo NEAREST_NEIGHBOR.

In breve, secondo la Figura 4.10, detto $T(N, A)$ l'albero costituito da nodi e archi e premesso che RRT esplora solo le regioni di C utili per il singolo problema considerato, l'algoritmo svolge i seguenti passi:

- genera q_{rand} con distrubuzione di probabilità uniforme in C
- calcola q_{new} come $q_{near} + \delta q_{rand}$, con passo $\delta \leq 1$ prefissato o variabile

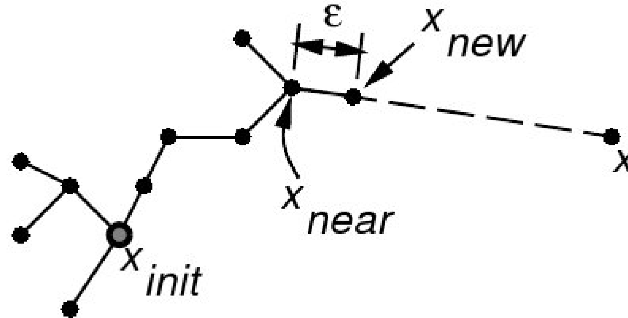


Figura 4.8: L'operazione EXTEND

```
CONNECT(T, x)
 1  repeat
 2       S ← EXTEND(T, x);
 3  until not (S = Advanced)
 4  Return S;
```

Figura 4.9: La funzione CONNECT

- test di collisione: su q_{new} e sul segmento aperto (q_{near}, q_{new}); se entrambi passano il test, T è espanso: $q_{new} \rightarrow N$, $(q_{near}, q_{new}) \rightarrow A$ altrimenti si ripete il loop.

Diversi tipi di pianificatori RRT

Come mostrato dalla Figura 4.11, esistono diverse tipologie di algoritmi basati su RRT; queste differiscono per il modo in cui viene generato l'albero, oppure dalla possibilità di generare due alberi per velocizzare la soluzione del problema.

Gli algoritmi che utilizzano questo meccanismo sono detti pianificatori bidirezionali e in MSL sono implementati a partire dalla classe *RRTDual*.

Ispirandosi alle classiche tecniche di ricerca bidirezionali, è ragionevole pensare che si possa ottenere un innalzamento delle performance costruendo due alberi RRT, uno da x_{init} e uno da x_{goal}; la soluzione è trovata quando i due alberi si incontrano.

La costruzione deve essere fatta comunque in modo da assicurare

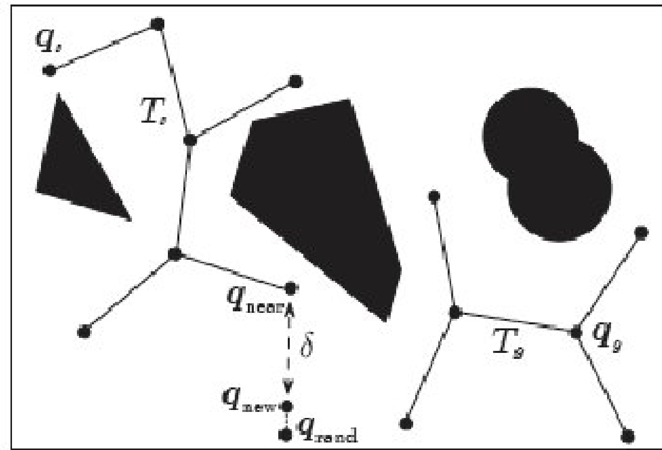


Figura 4.10: Algoritmo RRT

che i due alberi si incontrino prima di aver coperto tutta lo spazio degli stati e da permettere un'efficiente individuazione dell'incontro.

Proprio sull'individuazione dell'incontro si è scoperto un difetto abbastanza grave nel software: sul tratto di unione dei due alberi non viene effettuato il test di collisione. Poteva succedere che quindi venissero create pianificazioni assolutamente prive di senso. La connessione tra i due alberi viene effettuata se i 3 paramentri di stato (coordinate della posizione e inclinazione) soddisfano certi vincoli di vicinanza.

```
if (GapSatisfied(nn->State(), nn2->State()))
```

Questi vincoli (*GapError*) devono quindi essere scelti con molta attenzione, badando che siano inferiori alle dimensioni del robot, in modo da prevenire gli errori potenzialmenti generati dalla manacata esecuzione del test di collisione sull'arco che unisce i due alberi. Qui di seguito riportiamo il frammento di codice che esegue il test per stabilire se i due alberi possono essere uniti.

La Figura 4.12 mostra l'algoritmo bidirezionale RRT che va confrontato con l'algoritmo standard (Figura 4.7 nella pagina 35). L'algoritmo bidirezionale divide la computazione in due processi: 1) esplora lo spazio degli stati; 2) tenta di far crescere gli alberi l'uno verso l'altro. I due alberi T_a e T_b sono mantenti in memoria per tutto il tempo, finchè non si connettono e quindi una soluzione viene trovata. In ogni iterazione, un albero viene esteso e viene svolto il tentativo di connettere il nodo più dell'altro albero al nuovo vertice. Poi, i ruoli sono invertiti

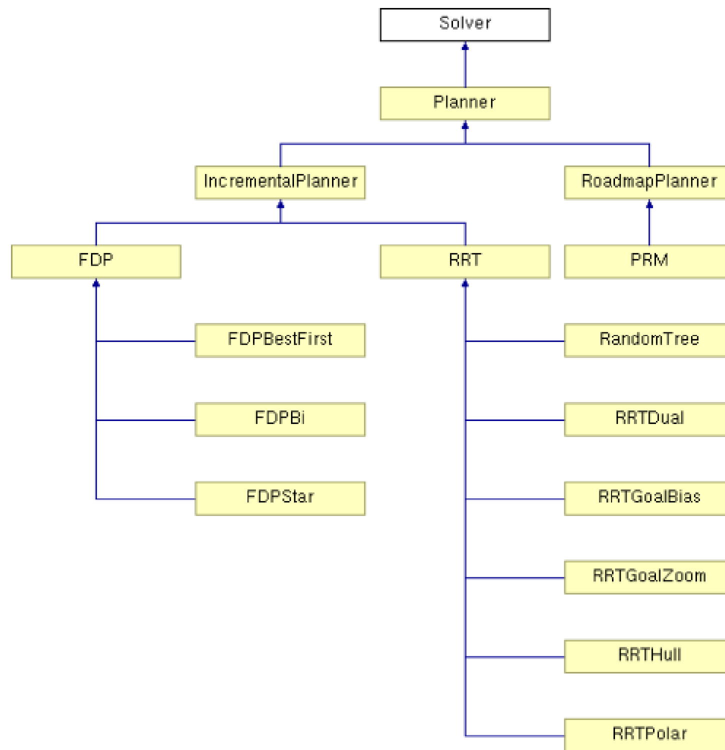


Figura 4.11: classe Solver e tipi di RRT

scambiando i due alberi.

In breve, secondo la Figura 4.13, con RRT bidirezionale si velocizza la ricerca espandendo due alberi in parallelo T_s dallo start e T_g dal goal; dopo un certo numero di iterazioni, si tenta di connettere i due alberi:

- test di collisione sul segmento tra q_{new} generato da T_s e q_{near} generato da T_g; se il test va a buon fine, l'espansione connette i due alberi; altrimenti si aggiunge a T_s solo l'eventuale tratto privo di collisioni.
- si ripete il tentativo scambiando i ruoli di T_s e T_g (bidirezionalità).
- se dopo un certo numero di tentativi non si ottiene connessione, cioè i due alberi sono ancora troppo lontani, si ritorna alla fase di espansione in parallelo dei due alberi .

Della famiglia degli algoritmi bidirezionali fa parte RRTExtExt (nella Figura 4.14 è mostrato il diagramma di ereditarietà della relativa classe); questo modello è sembrato in prima analisi il più adatto alla

```
bool Planner::GapSatisfied(const MSLVector &x1, const MSLVector
 &x2) {
  MSLVector x;
  int i;

  x = P->StateDifference(x1,x2);
  for (i = 0; i < P->StateDim; i++) {
    if (fabs(x[i]) > GapError[i])
      return false;
  }
```

RRT_BIDIRECTIONAL(x_{init}, x_{goal})
1 $\mathcal{T}_a$.init(x_{init}); $\mathcal{T}_b$.init(x_{goal});
2 **for** $k = 1$ **to** K **do**
3 $x_{rand} \leftarrow$ RANDOM_STATE();
4 **if not** (EXTEND($\mathcal{T}_a, x_{rand}$) = *Trapped*) **then**
5 **if** (EXTEND($\mathcal{T}_b, x_{new}$) = *Reached*) **then**
6 Return PATH($\mathcal{T}_a, \mathcal{T}_b$);
7 SWAP($\mathcal{T}_a, \mathcal{T}_b$);
8 Return *Failure*

Figura 4.12: Pianificatore bidirezionale RRT

pianificazione in ambienti domestici per la carrozina, ma un'analisi più approfondita ha permesso di trovare delle modifiche che hanno portato a un netto miglioramento delle *performance*. Questa analisi ha portato allo sviluppo di un nuovo algoritmo, chiamato *RRTBidirBalancedInpid*, che sarà illustrato nel paragrafo 5.3.

4.4 Prove del tool MSL

In questa sezione sono riportati le soluzioni di alcuni problemi presenti a titolo esemplificativo nella MSL. Mentre nel capitolo 6 nella pagina 58 saranno presentati le realizzazioni sperimentali e la valutazione delle soluzioni prodotte dopo aver applicato le modifiche al codice per renderlo aderente alle caratterstiche della carrozzina.

La Figura 4.15 mostra la costruzione di un albero secondo l'algoritmo RRT. La generazione random dei nodi porta, in prima istanza, a una soluzione non ottimale, in special modo se si usa un limitato

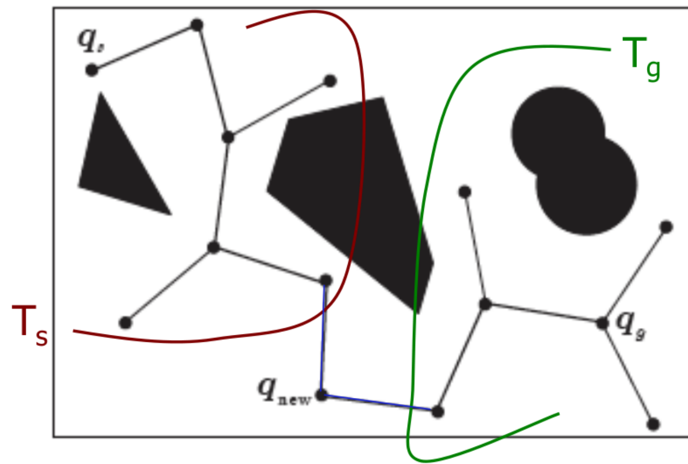


Figura 4.13: RRT bidirezionale

numero di nodi. Soluzioni a questo problema sono l'innalzamento del numero di nodi, oppure la creazione di più percorsi seguita dalla scelta del percorso migliore secondo diverse proprietà quali la lunghezza, il numero di nodi, o il tempo impiegato per la generazione (Figura 4.16).

La serie di immagini in Figura 4.17 mostra il diagramma di Voronoi dei vertici dell'RRT durante più step della costruzione dell'albero. Si noti che le regioni di Voronoi più larghe appaiono alla frontiera, deviando in questo modo l'esplorazione verso porzioni inesplorate dello spazio.

Di seguito sono mostrati i risultati sperimentali nel caso di un passaggio attraverso una porta stretta (Figura 4.18), e nel caso di una stanza con due muri da superare (Figura 4.19). Nel primo esempio si può apprezzare anche la forma data al robot, di cui ovviamente la pianificazione tiene conto. Nel planner basato su RRT si può definire la forma e l'ingombro del robot. Nella Figura 4.18 si vede che il pianificatore tiene conto di queste informazioni per rendere sicuro e più agevole il passaggio tra gli ostacoli.

La costruzione dell'albero in modo random porta ad alcune soluzioni che potrebbero apparire poco sensate. In verità con alcuni accorgimenti si riesce ad avere pianificazioni molto buone. Per esempio nel caso di una stanza vuota (Figura 4.20), la pianificazione da un angolo al suo opposto appare come quella in figura. La pianificazione ottimale sarebbe quella di una semplice linea retta, ma anche la soluzione RRT è più che accettabile, soprattutto considerando la semplificazione che

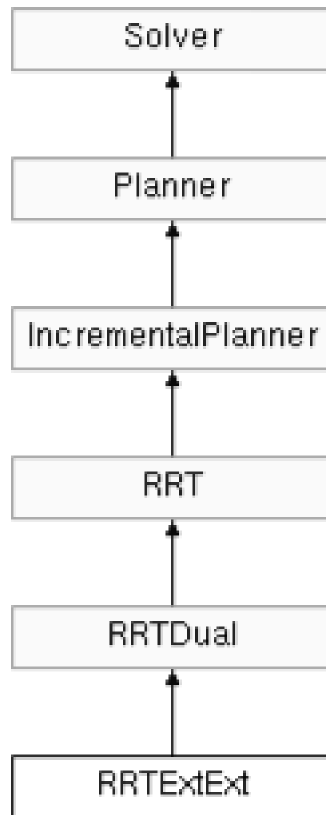


Figura 4.14: RRTExtExt

rettifica i via point.

L'algoritmo RRT è altamente personalizzabile e si possono imporre diversi tipi di vincoli. Per esempio la Figura 4.21 mostra l'albero che si otterrebbe se il robot fosse capace solo di andare avanti o indietro o di svolgere curve di 90°

Soluzioni bizzarre si possono avere imponendo che il robot abbia solo la capacità di girare a sinistra (Figura 4.22). RRT trova comunque un percorso che soddisfa le richieste anche in mappe mediamente complesse e con molti ostacoli (Figura 4.23)

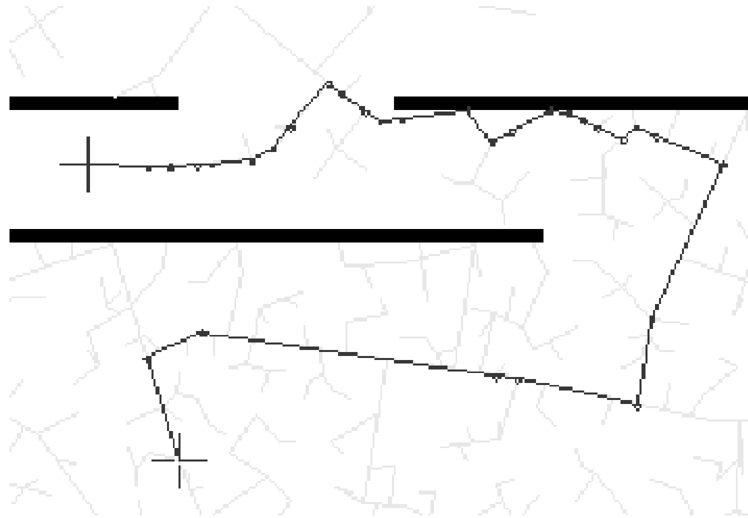


Figura 4.15: Rapidly-exploring Random Tree

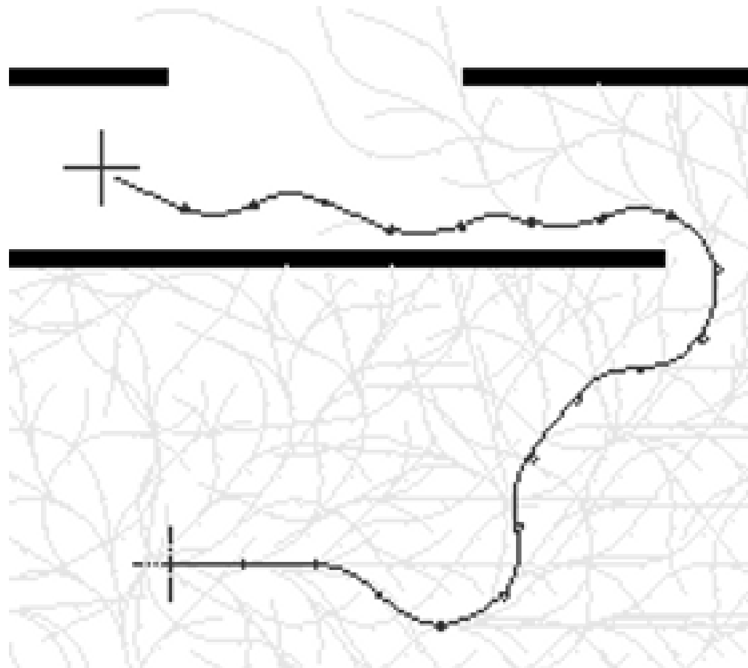
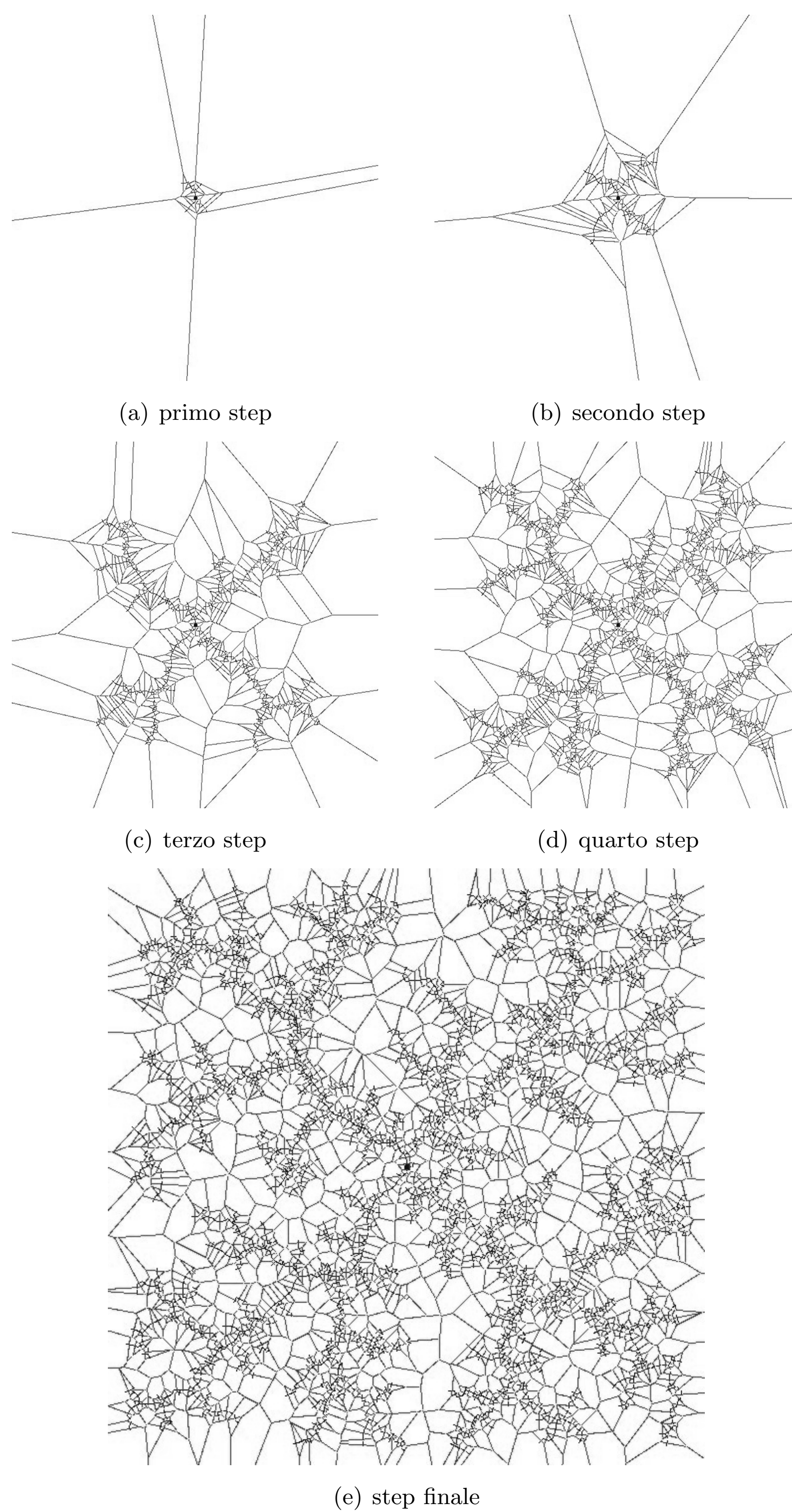


Figura 4.16: Pianificazione

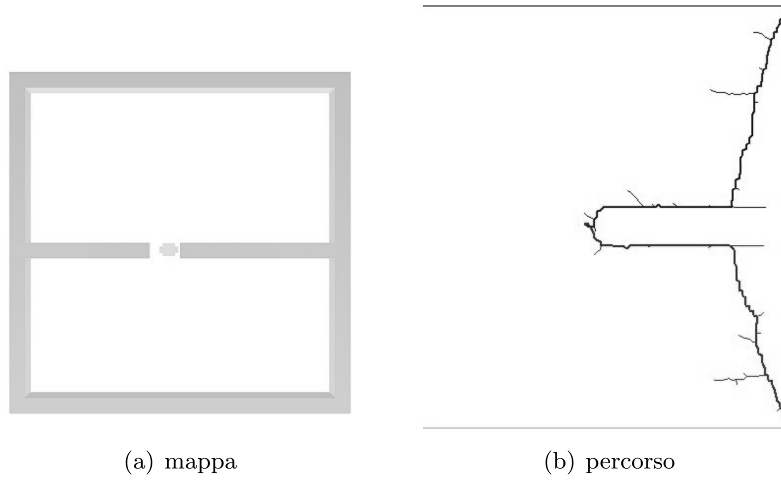


(a) primo step (b) secondo step

(c) terzo step (d) quarto step

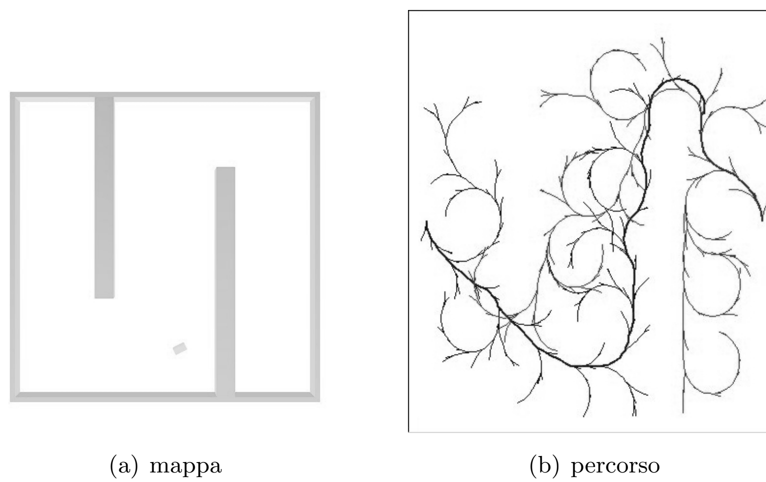
(e) step finale

Figura 4.17: diagrammi di Voronoi nella costruzione dell'albero



(a) mappa (b) percorso

Figura 4.18: passaggio stretto e forma del robot



(a) mappa (b) percorso

Figura 4.19: stanza con due muri

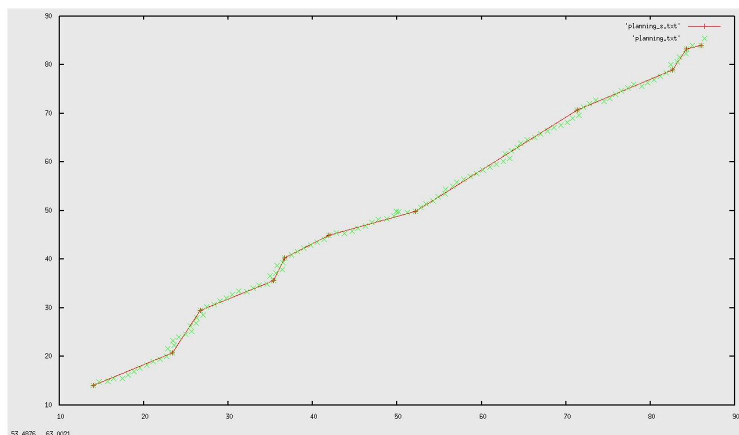


Figura 4.20: pianidicazione in una stanza vuota

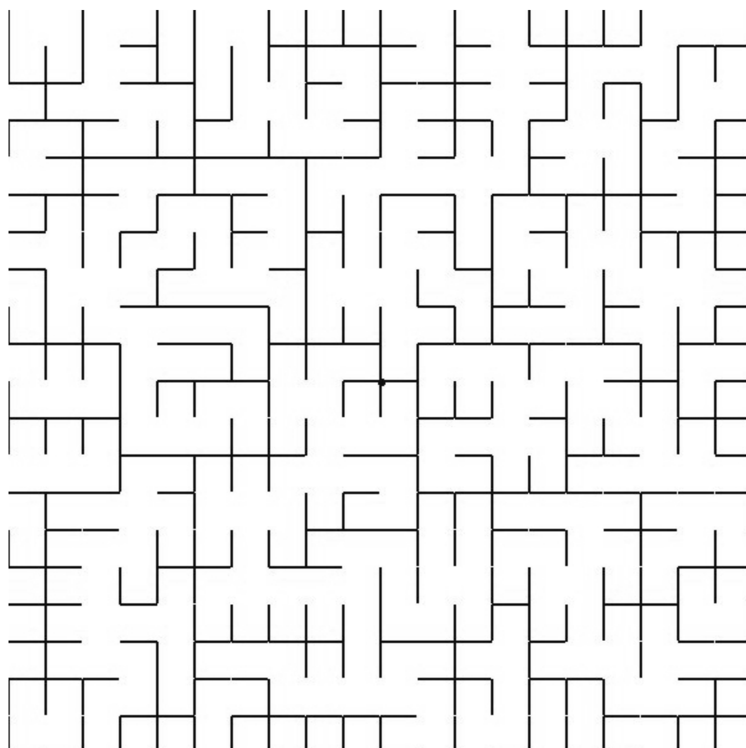


Figura 4.21: Albero generato nel caso in cui il robot abbia il vincolo di sterzo a 90°

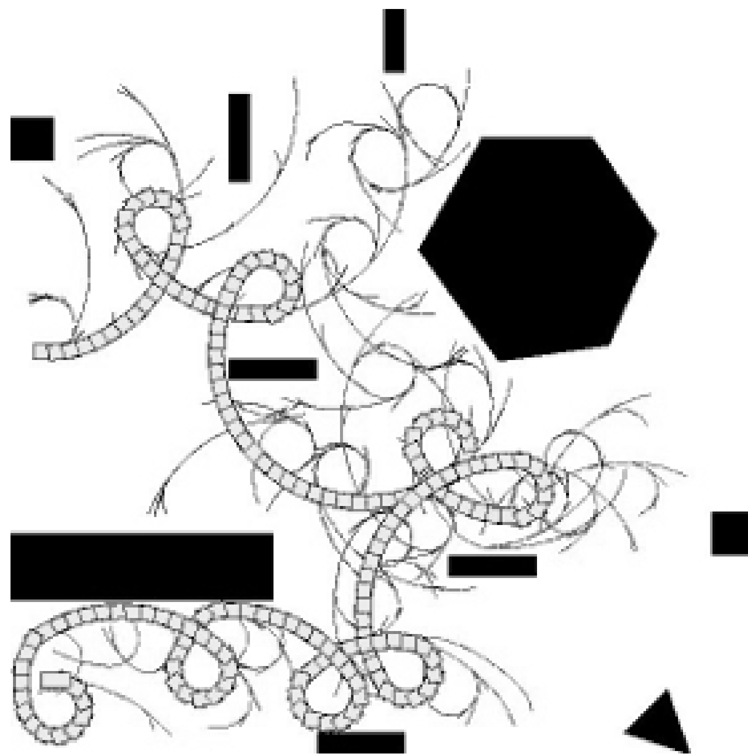


Figura 4.22: Pianificazione per un robot capace di sterzare solo a sinistra / 1

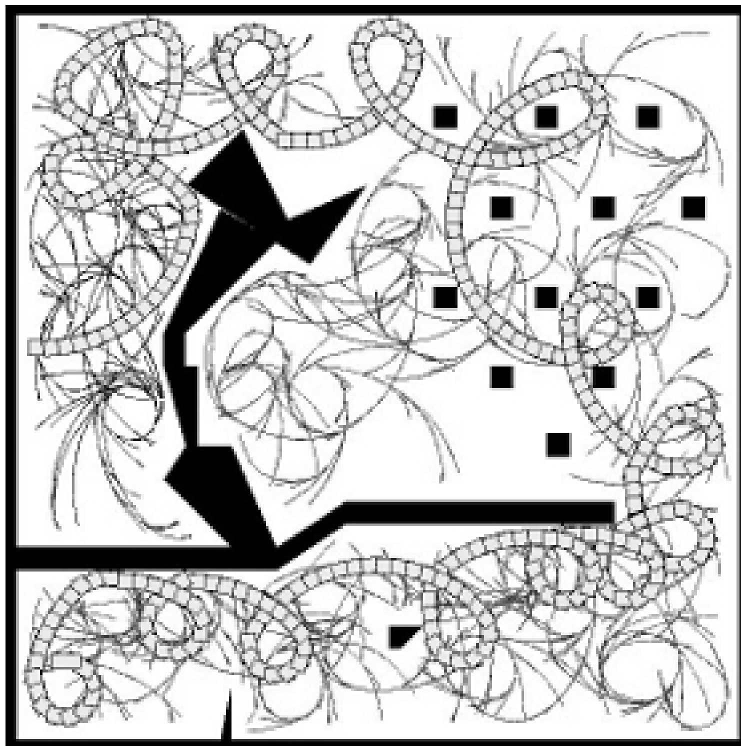


Figura 4.23: Pianificazione per un robot capace di sterzare solo a sinistra / 2

Capitolo 5

Pianificatore basato su RRT e relativo software

"Non posso cambiare la direzione del vento, ma posso spiegare le mie vele in modo tale da giungere sempre a destinazione."

Anonimo

5.1 Dal codice sorgente originario

Il primo passo per giungere a un software utilizzabile sulla carrozzina è stato quello di perfezionare il codice sorgente della Motion Strategy Library. Questo infatti dava molti errori in fase di configurazione e di compilazione su macchina Linux/Ubuntu tramite l'utility *configure* e *make*: per esempio molte dipendenze non venivano risolte e l'invocazione di alcuni programmi esterni falliva. Il codice sorgende della MSL non veniva quindi compilato in codice oggetto, visto che falliva l'unione e il link del codice oggetto in eseguibili o in librerie.

Nel software della MSL sono presenti anche i file *configure* e *makefile* per determinare il grafo delle dipendenze per un particolare output, e gli script necessari per la compilazione da passare alla shell. È stato necessario installare le librerie *freeglut*, *inventor*, *libFox*, il cui scopo è spiegato nel paragrafo 4.1 nella pagina 24. Sono state necessarie anche alcune modiche ad alcuni FLAGS usati nel makefile, il perfezionamento dei *namespaces* utilizzati e l'inclusione esplicita di alcune librerie in file *.c* e *.h*. Con queste modifiche la libreria è stata resa compatiile con la versione 4.3.3 (Ubuntu 4.3.3-5ubuntu4) di GCC.

Per adattare la libreria all'uso specifico sulla carrozzina sono state apportate numerose modifiche e notevoli semplificazioni, che riguardano in special modo l'esclusione di tutti gli aspetti grafici e relativi al render. Inoltre si è reso necessario definire un modello *ad hoc* e un algoritmo pensato appositamente per risolvere il problema della pianificazione locale per la carrozzina.

5.2 Il modello di moto della carrozzina: stato e ingressi

Nessuno dei modelli proposti dalla MSL rispecchia completamente la cinematica della carrozzina. Nei modelli forniti dalla MSL, l'inesattezza più evidente consiste nella cinematica della sterzata, che è eseguita secondo il paradigma di una automobile, cioè con le ruote anteriori sterzanti. Per quanto riguarda la carrozzina, invece, le sterzate vengono imposte dai due motori posteriori, attraverso una diversa velocità imposta alla ruota destra o sinistra. Una evidente conseguenza di questa caratteristica è che l'angolo di sterzata non è costante, ma dipende dalla velocità, e che quindi con velocità nulla, è possibile svolgere rotazioni sul posto. Questo modello è noto come *differential drive.*

Lo stato della carrozzina è descritto dalle seguenti equazioni:

$$\begin{aligned} x_t &= x_{t-1} + v\cdot \cos(\vartheta_{t-1})\cdot \Delta t, \\ y_t &= y_{t-1} + v\cdot \sin(\vartheta_{t-1})\cdot \Delta t, \\ \vartheta_t &= \omega\cdot \Delta t \end{aligned} \tag{5.1}$$

dove x e y rappresentano la posizione in coordinate cartesiane, ϑ rappresenta l'angolo di inclinazione e v e ω sono definite nel modo seguente

$$\begin{aligned} v &= \frac{v_l + v_r}{2}, \\ \omega &= \frac{v_r - v_l}{L}, \end{aligned} \tag{5.2}$$

dove L è la larghezza nominale del robot, ovvero la distanza tra le due ruote.

Da queste equazioni si ricava facilmente il legame tra x_t e x_{t-1}, y_t e y_{t-1}, ϑ_t e ϑ_{t-1}:

$$\begin{aligned} dx &= v \cdot \cos(\vartheta_{t-1}), \\ dy &= v \cdot \sin(\vartheta_{t-1}), \\ d\vartheta &= \omega \end{aligned} \tag{5.3}$$

Le suddette equazioni sono state tradotte nel seguente frammento di codice:

```
MSLVector Model2DWheelchair::StateTransitionEquation(const
 MSLVector &x, const MSLVector &u) {

  MSLVector dx(StateDim);
  double v,w;
  v=(u[0]+u[1])/2.0;
  w=(u[1]-u[0])/L;

  dx[0] = v*cos(x[2]);
  dx[1] = v*sin(x[2]);
  dx[2] = w;
  return dx;
}
```

Dalle equazioni della velocità si ricavano altre considerazioni sul raggio si sterzata della carrozzina (Figura 5.1). Per convertire la velocità del robot espressa come v e ω in velocità delle singole ruote si seguono le seguenti espressioni

$$\begin{aligned} v_L &= \omega(R - \frac{L}{2}), \\ v_R &= \omega(R + \frac{L}{2}), \end{aligned} \tag{5.4}$$

dove R è il raggio di curvatura calcolabile come

$$R = \frac{L}{2} \cdot \frac{v_R + v_L}{v_R - v_L}, \tag{5.5}$$

che quindi non è costante.

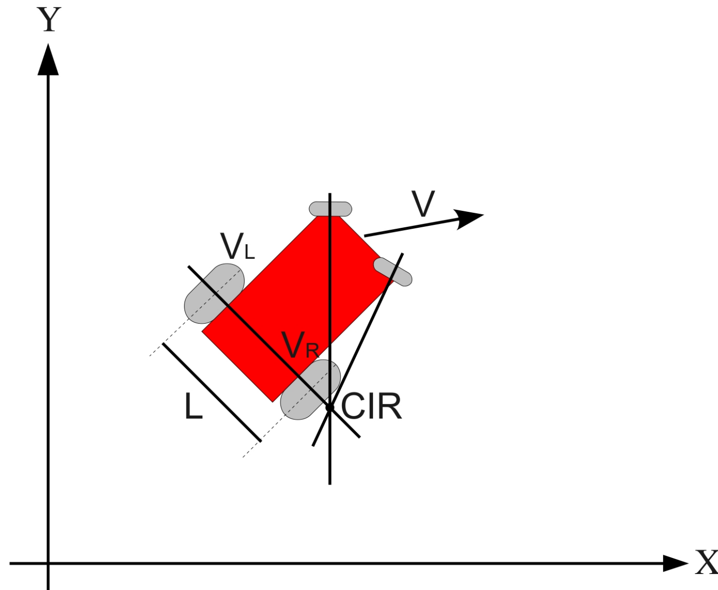


Figura 5.1: Centro di Istantanea Rotazione (CIR)

Le variabili di ingresso al modello di moto (Input) rappresentano le azioni da applicare alle equazioni di stato per trovare lo stato successivo. Definendo gli Input come variazioni sulla posizione si creano molti nodi che corrispondono a stati non validi. In altre parole, in questo caso si creerebbero molti nodi, anche raggiungendo il valore limite che determina il fallimento della pianificazione, ma senza far salire significativamente il numero di nodi validi, cioè quelli che soddisfano i vincoli cinetici e i vincoli dovuti alla presenza di ostacoli. Si è quindi deciso di definire gli ingressi come velocità da imprimere ai motori, ovvero sono stati derivati alcuni valori della coppia v_l e v_r.

Sono stati modellizzati tre insiemi di possibili Input per la carrozzina:

- spostamento in avanti con possibilità di sterzate. Detta max_v_motor la velocità massima dei motori, questo Input è descritto imponendo max_v_motor a entrambi i motori, oppure max_v_motor a un motore e $\frac{max_v_motor}{2}$ all'altro e viceversa. Attualmente max_v_motor è impostata a 0.5 metri per unità di tempo, anche se questa non è la reale velocità massima dei motori, ma una velocità ritenuta adeguata per un movimento autonomo.

- rotazioni sul posto, ottenute imponendo $\frac{max_v_motor}{2}$ a un motore e $-\frac{max_v_motor}{2}$ all'altro o viceversa.
- retromarcia con sterzate, imponendo $-max_v_motor$ a entrambi i motori, oppure $-max_v_motor$ a un motore e $-\frac{max_v_motor}{2}$ all'altro.
- altre tipologie di movimento come le rotazioni "larghe" (ovvero con una ruota ferma a fare da perno), sia in avanti che in retromarcia, non sono state prese in considerazione in quanto avrebbero inutilmente complicato il processo di pianificazione. Tali tipi di ingressi corrispondono all'imposizione di una certa velocità a una ruota (velocità positiva o negativa) mantenendo ferma l'altra.

Seguendo il paradigma della MSL, queste azioni vengono applicate ad ogni nodo dell'albero. Così facendo, però, la pianificazione includeva retromarce o rotazioni sul posto anche quando non erano strettamente necessarie. È stata quindi apportata una semplice ma importante modifica: dividere le tre azioni in gruppi e dare la possibilità di passare da un gruppo di Input all'altro solo in determinate circostanze.

Per questo approccio si è preso spunto da un articolo di Xiaoshan Pan, di cui al riferimento bibliografico [11]. Nel caso della pianificazione del percorso della carrozzina, prima vengono generati tutti i nodi costruiti con azioni in avanti e solo quando non è più possibile muoversi con questo tipo di Input, si passa alla creazione di nodi mediante azioni di rotazione. Dopodichè si cercano ancora nodi in avanti e solo quando non è più possibile muoversi né avanti né tramite rotazioni, si generano nodi mediante retromarcia.

Questo comportamento è schematizzato tramite l'automa a stati finiti descritto in Figura 5.2.

La soglia alla quale effettuare il cambio del metodo di generazione dei nodi è modificabile; attualmente è impostata a 0, cioè viene effettuato lo switch quando la precedente modalità non ha più possibilità di creare nuovi nodi. Questo approccio garantisce che si giunga allo stato di fallimento solo quando effettivamente sono state esaurite tutte le possibilità di movimento per la carrozzina.

5.3 RRTBidirBalancedIndip

Oltre al modello di moto aderente alle caratteristiche della carrozzina (descritto nel Paragrafo 5.2 nella pagina 49), è stato creato anche un

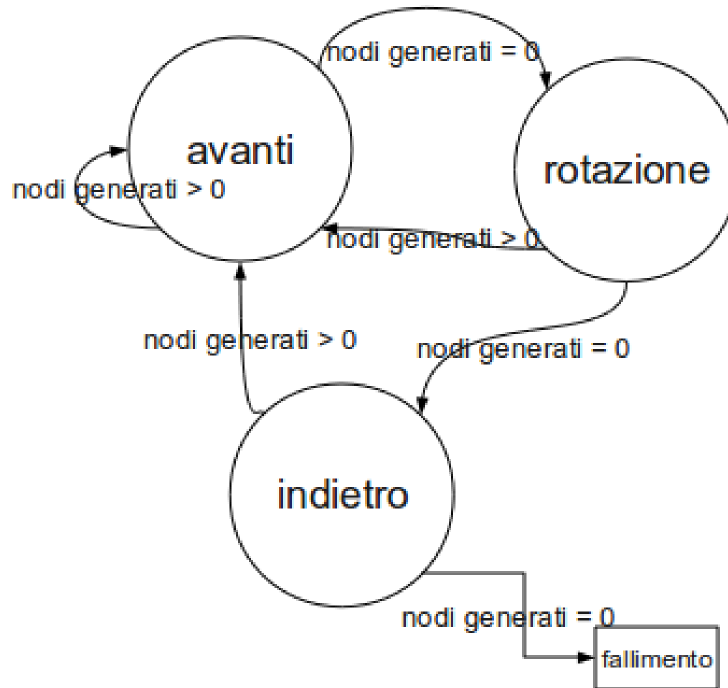


Figura 5.2: Switch del tipo di Input con soglia = 0

algoritmo *ad hoc* per la risoluzione del problema di pianificazione.

I pianificatori già implementati nella MSL si sono infatti rivelati inadeguati. Nella classe di base (*RRT*), un unico albero viene generato senza tenere in considerazione il *GoalState*. Nella famiglia dei pianificatori RRT a singolo albero sono stati presi in considerazione

- RRTGoalBias: con un certo valore di probabilità, si cerca di connettere lo stato finale invece che nodi casuali ai nodi già esistenti nell'albero. La probabilità con cui si sceglie il GoalState al posto che uno stato a caso è *GoalProb*. In mappe che si rifanno ad un ambiente domestico, questo approccio non porta significativi miglioramenti

- RRTGoalZoom: cerca di far propendere i nodi casuali generati verso la regione che contiene la meta. La regione si stringe intorno alla meta man mano che l'albero si avvicina. Questo approccio cade facilmente in minimi locali.

Approcci ad albero doppio (*RRTDual*) sono più efficienti e in particolare

- RRTConCon: cerca di bilanciare la computazione tra la crescita dell'albero verso nodi casuali e la crescita verso l'altro albero. Detto G l'albero che parte dallo stato iniziale e $G2$ l'albero che parte dalo stato obiettivo, in ogni iterazione vengono svolte quattro fasi: cresce G verso un nodo casuale, cresce $G2$ verso il nuovo nodo di G, cresce $G2$ verso un nodo casuale, cresce G verso il nuovo nodo di $G2$.

- RRTBidirBalanced: si comporta in modo simile a RRTConCon, eccetto per il fatto che viene introdotto un criterio di cardinalità per mantenere l'equilibrio tra il numero di nodi in ogni albero. Ad ogni iterazione, l'albero con il minor numero di nodi è selezionato come l'albero attivo.

L'algoritmo creato è stato chiamato *RRTBidirBalancedIndip*; il suo nome deriva dalle sue caratteristiche:

- *Bidir* perché appartiene alla classe degli algoritmi RRTDual, cioè RRT potenziato dal fatto che due pianificazioni vengono eseguite contemporaneamente: una dallo start e una in modo *reverse* dal goal, in modo da diminuire il tempo di pianificazione (si veda il paragrafo 4.3 nella pagina 36)

- *Balanced* perché gli alberi di start e goal sono fatti crescere in modo bilanciato in termini di numero di nodi. A differenza dell'algoritmo *RRTBidirBalanced* proposto dalla MSL è stata introdotta una soglia entro la quale un albero può crescere con più nodi rispetto all'altro, in modo tale da garantire una esplorazione più veloce in spazi aperti. Il numero di nodi che fungono da soglia è stato scelto comunque in modo da non penalizzare eccessivamente nessun albero.

- *Indip* perché il set di Input utilizzato da un albero non è legato a quello dell'altro albero ma è indipendente. L'indipendenza è stata introdotta per evitare che la pianificazione risultasse simmetrica, ovvero, ad esempio, che l'inizio di un percorso in retromarcia costringa anche il raggiungimento del goal in retromarcia.

5.4 Struttura e funzionamento del software

Il paragrafo 4.1.2 nella pagina 26 spiega come viene creato un problema nella Motion Strategy Library. In questa sezione invece si illustra la struttura e il funzionamento del software adattato all'utilizzo per la carrozzina.

L'esecuzione del programma inizia con l'impostazione del modello e della geometria a cui il robot appartiene. Come modello si utilizza quello descritto nel Paragrafo 5.2 nella pagina 49; come *Geom* si è lasciato quello di *default* ovvero *GeomPQP*.

A questo punto, il programma legge da file la mappa con gli ostacoli contenuta in un file. Per la lettura del file con la mappa, il codice è contenuto nel file *GeomPQP.c* ed è il seguente:

```
READ_OPTIONAL_PARAMETER(Obst);
```

con riferimento alla *macro*

```
#define READ_OPTIONAL_PARAMETER(F)
_msl_fin.clear();
_msl_fin.open((FilePath+""#F"").c_str());
if (_msl_fin) {_msl_fin >> F;}
_msl_fin.close();
```

La struttura del file *Obst* è mostrata nel riquadro

```
(0,0) (90,0)
(90,0)(90,100)
(90,100)(0,100)
(0,100)(0,0)

(30,100)(30,70)
(60,70)(60,100)
(30,70)(40,70)
(50,70)(70,70)
(80,70)(90,70)

(90,50)(25,50)(25,40)(90,40)
(70,30)(25,30)(25,20)(70,20)
```

cioè è semplicemente una lista di ostacoli, descritti come una lista di poligoni di cui sono specificati i vertici.

A differenza di quanto accadeva nel codice originario, l'InitialState e il GoalState vengono impostati da codice, ovvero sono stati implementati i metodi *setInitialState* e *setGoalState*, che permettono di specificare la posizione di start e goal nei loro tre parametri x, y, ϑ (Figura 5.3).

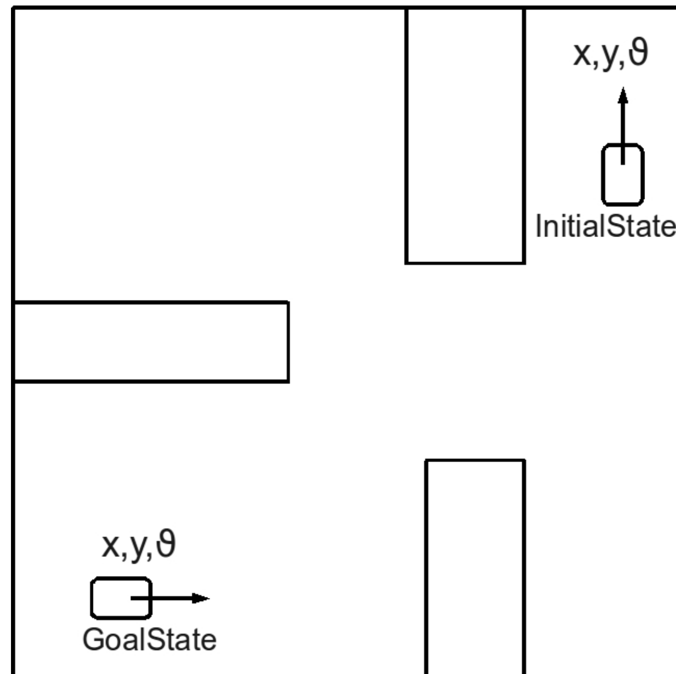


Figura 5.3: Parametri di stato

Il terzo parametro passato come *InitialState* e *GoalState*, oltre ovviamente le coordinate x e y del punto di partenza e di arrivo, è il coefficiente angolare che descrive l'orientamento del robot alla partenza e che impone la direzione del robot all'arrivo. Questo parametro è espresso in radianti, quindi può assumere valori che vanno da 0 a 2π.

L'orientamento all'arrivo, che può essere utile ai fini pratici d'uso, è scritto nella pianificazione risultante, ma può essere anche facilmente verificato dopo l'esecuzione dell'algoritmo RRT, usando gli ultimi punti del planning; successivamente è quindi possibile far ruotare la carrozzina per raggiungere con ulteriore precisione l'orientamento desiderato.

Dopo aver settato il Model e la Geom e aver letto da file la mappa con gli ostacoli, il programma provvede a cancellare eventuali pianificazioni precedenti presenti nel sistema, predisponendosi di fatto alla crazione di una nuova pianificazione. Il problema viene creato leggendo l'Initial State e il GoalState e successivamente si invoca l'algoritmo per la soluzione.

Il primo passo da compiere riguarda la scelta dell'algoritmo di pianificazione, che di dafault è RRTBidirBalancedIndip.

Con l'istruzione *Plan* si invoca la pianificazione vera e propria; questa viene ripetuta finchè non va a buon fine. Il fallimento della connessione tra start e goal e quindi la necessità dell'invocazione di una nuova pianificazione viene stabilito basandosi su alcuni parametri, quali in numero di nodi generati, le collisioni rilevate o il tempo di pianificazione.

Quando RRT riesce a connettere lo *start* con il *goal*, viene fornito il percorso sottoforma di una lista di via point. Successivamente questa lista molto fitta di pose (x, y, ϑ) viene semplificata per poter essere eseguita dal controllore della traiettoria.

Per quanto riguarda i tempi di pianificazone, questi dipendono dall'unità di tempo ($deltaT$) impostata. Sperimentalmente si è visto che un $deltaT$ che consenta di creare alberi abbastanza fitti, ma che al contempo garantisca una veloce soluzione del problema è di $0.5s$.

Con questo valore di $deltaT$ e una velocità massima dei motori di $0,5m/s$, si impone una distanza di $0,25m$ tra due nodi.

In rotazione, imponendo $\frac{max_v_motor}{2}$ $(0,25m/s)$ a un motore e $-\frac{max_v_motor}{2}$ $(-0,25m/s)$ all'altro e con larghezza del robot pari a $60cm$, si ricava una variazione di ω di circa $0,27rad/s$ $(15,5°/s)$. Quindi con $deltaT = 0,5s$, l'orientamento tra due nodi differisce di $0,133rad$, cioè circa $7,62°$.

In ambiente domestico il tempo di pianificazione va dai pochi secondi per percorsi non intricati, alla decina di secondi per percorsi più ardui e complicati, o intorno ai $30s$ per pianificazioni in mappe caotiche.

Capitolo 6

Realizzazioni sperimentali e valutazione

"Una bussola non dispensa dal remare."

Anonimo

In questo capitolo si mostrano alcuni risultati ottenuti tramite l'algoritmo di pianificazione applicato a situazioni reali. In particolare si mostra la crescita dell'alberto RRT secondo il modello della carrozzina in ambienti noti e la pianificazione risultante, utilizzando i plot dell'ambiente MATLAB.

La prima mappa presa in considerazione è stata creata appositamente per testare se l'algoritmo RRTBidirBalancedIndip riuscisse a trovare pianificazioni sensate in ambienti complessi, in una sorta di labirinto. La mappa è mostrata in Figura 6.1 e riporta la pianificazione in un caso molto semplice. Le dimensioni della mappa sono in questo caso di $20m$ x $20m$, mentre il corridoio dove viene richiesta la pianificazione è largo circa $2,5m$. Si nota come la pianificazione venga generata già col primo set in Input cioè con movimento solo in avanti. I due alberi (in rosso l'albero che cresce dal Goal, in verde l'albero che cresce dallo Start) si incontrano dopo aver generato circa 80 nodi per albero e in termini di tempo la pianificazione è trovata in un intervallo che va dai $0,3s$ ai $0,7s$.

In Figura 6.2, il problema richiede di trovare una pianificazione con Start e Goal posti in modo più distante. In questo caso il tempo necessario per trovare la pianificazione è di circa $20s$. L'algoritmo ha bisogno di generare molti nodi prima di connettere i due alberi. La crescita

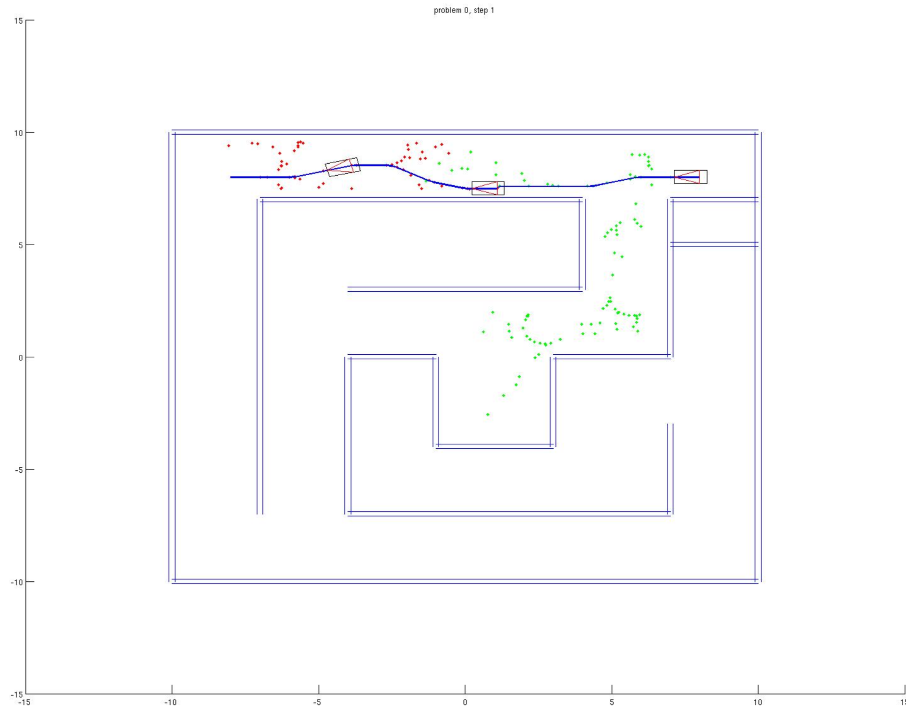


Figura 6.1: Pianificazione con Start e Goal posti in modo semplice

random ha come conseguenza l'esplorazione di parti di mappa che non risulteranno utili alla pianificazione finale. Con Input corrispondenti a movimementi in avanti si riescono a creare nuovi nodi ad ogni iterazione, raggiungendo così la costruzione del percorso senza ricorrere a manovre di retromarcia o rotazione.

Nelle Figure in 6.3 si richiede una pianificazione con Start e Goal posti in modo più complicato, sia come posizione sia come orientamento di origine e arrivo. Anche in questo caso, l'algortimo ha bisogno di creare molti punti per riuscire a connettere i due alberi. Il tempo di pianificazione di questo problema è di circa 30s. Anche in questo caso non c'è stato bisogno di ricorrere a manovre di retromarcia, mentre viene usata la manovra di rotazione per raggiungere l'inclinzazione richiesta all'arrivo.

La seconda mappa utilizzata per la sperimentazione è una mappa

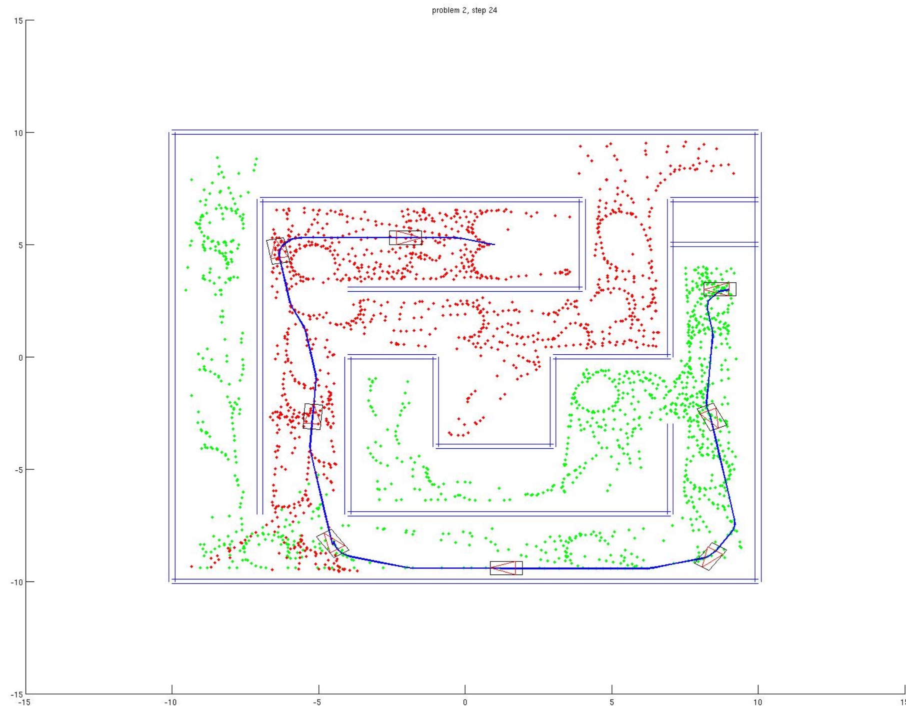
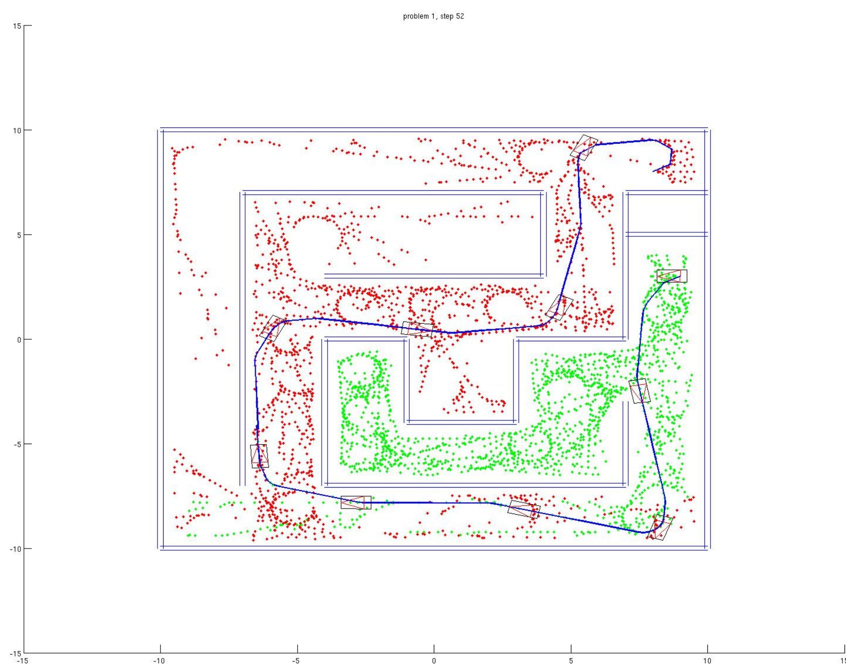


Figura 6.2: Pianificazione in mappa a labirinto

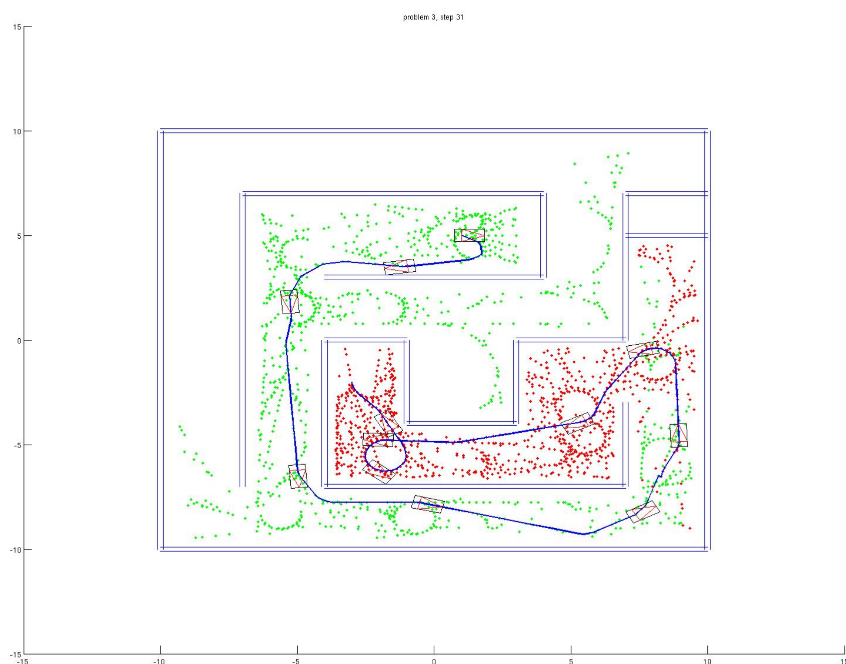
reale e rispecchia la zona dei Servizi Igienici in un edificio del complesso dell'università Bicocca. Gli ambienti sono dunque relativamente stretti; alcuni ambienti sono progettati per poter essere frequentati anche da persone disabili in carrozzina (tipicamente le stanze più grandi in Figura 6.4), ma la nostra sperimentazione non si è limitata alla verifica del movimento in tali ambienti, ma ha anche coinvolto le stanze non progettate specificatamente per la fruizione da parte di persone disabili. Precisamente tutta la zona presa in considerazione è di $19m$ x $13m$; la stanza di una singola toilette è $1m$ x $2m$. In questo caso il punto di Start è stato posizionato in una zona priva di ostagoli, mentre la destinazione è una stanza stretta, in cui l'inclinzazione richiesta all'arrivo non prevede rotazioni. Il percorso viene generato solo con movimenti in avanti; il tempo di pianificazione è di crica $15s$. La serie di immagini nelle Figure 6.5 e 6.6 mostra la crescita dell'albero durante l'esecuzione dell'algoritmo di pianificazione. Si vede come la crescita dei due alberi

è asimmentrica, infatti l'albero che parte dal Goal inizialmente cresce lentamente perché ha bisogno di rotazioni, ma ciò non interferisce con la crescita dell'albero di Start che al contrario è libero di generare nodi con movimenti in avanti.

La Figura 6.7 mostra il risultato della pianificazione in un caso dove la direzione di movimento in avanti non è suffciente per giungere alla soluzione del problema. In accordo con il metodo di generazione degli Input utilizzabili per la costruzione dell'albero, quando movimenti in avanti non portano alla generazione di nuovi nodi, si passa a movimenti di retromarcia e rotazione. È questo il caso della posizione di partenza, nella quale la carrozzina è obbligata a ruotare su se stessa per uscire dalla prima stanza. Il tempo di pianificazione in questo caso è di circa $10s$. Nel grafico in Figura 6.8 si mostrano le azioni intraprese per ogni ciclo di iterazione dell'algoritmo. Sull'asse verticale c'è il numero di nodi nuovi generati, sull'asse orizzontale ci sono i cicli dell'algoritmo; ciascuna barra in ogni ciclo è etichettata col nome del set di Input attivo in quel ciclo (forward, backward o rotation). Quando i movimenti in avanti non sono in grado di generare nuovi nodi, si passa alle rotazioni; dopo aver generato i nodi derivanti dalle rotazioni, nuovi movimenti in avanti diventano disponibili e in questo modo si giunge alla soluzione del problema di pianificazione.



(a)



(b)

Figura 6.3: Pianificazione in mappa a labirinto con rotazioni

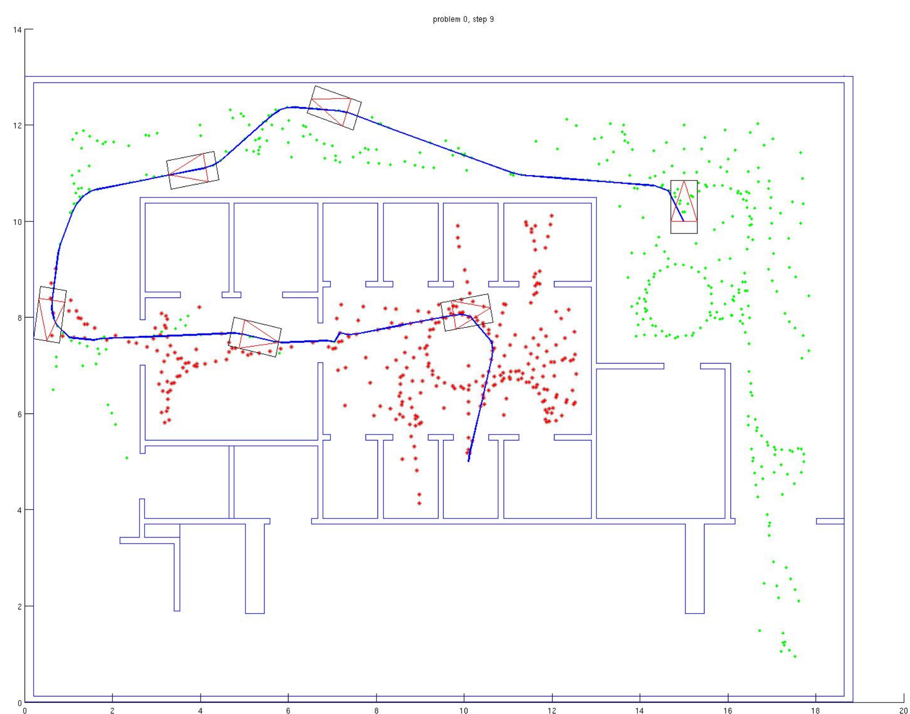
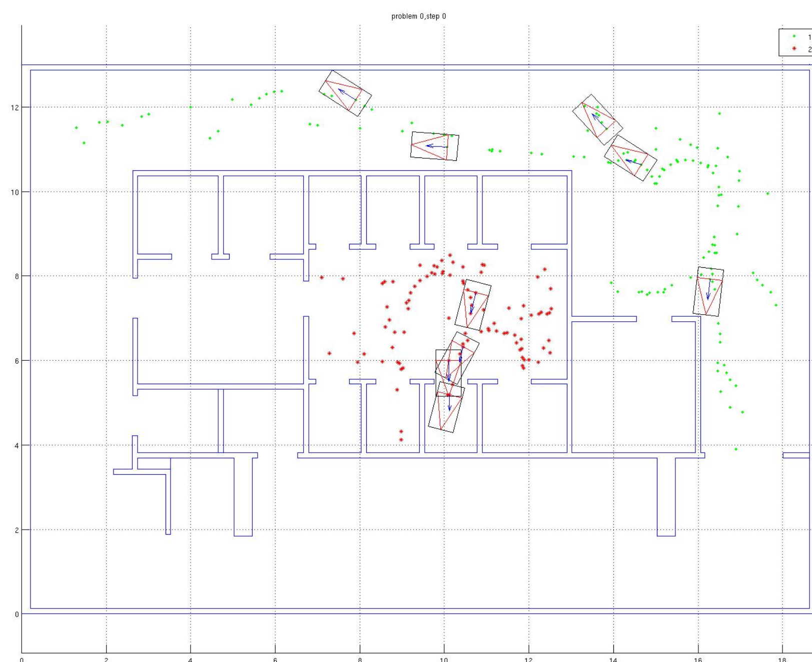
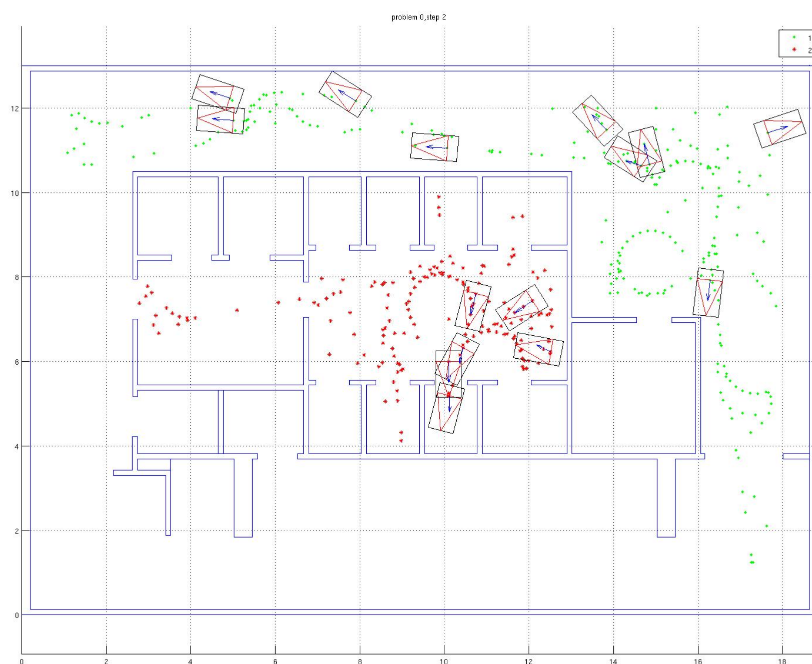


Figura 6.4: Mappa della zona Servizi Igienici Bicocca

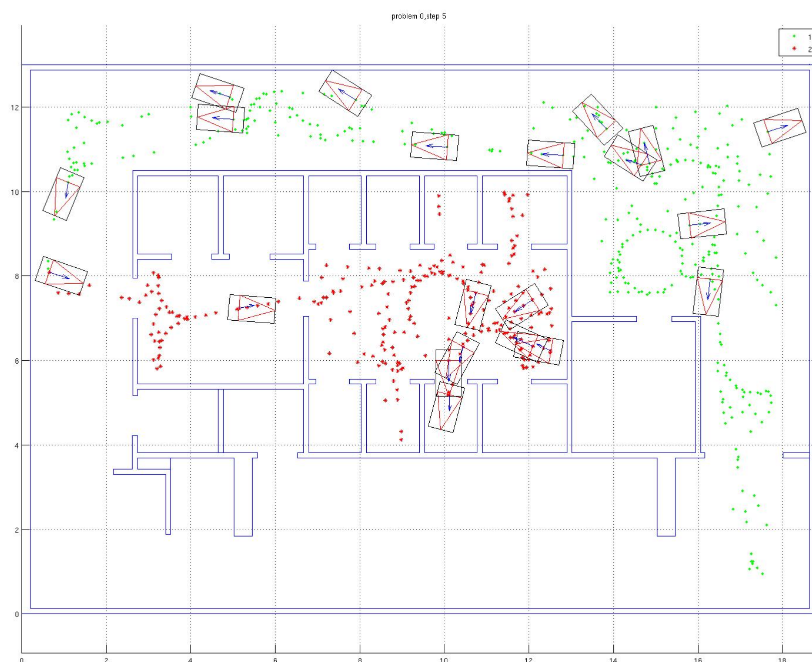


(a)

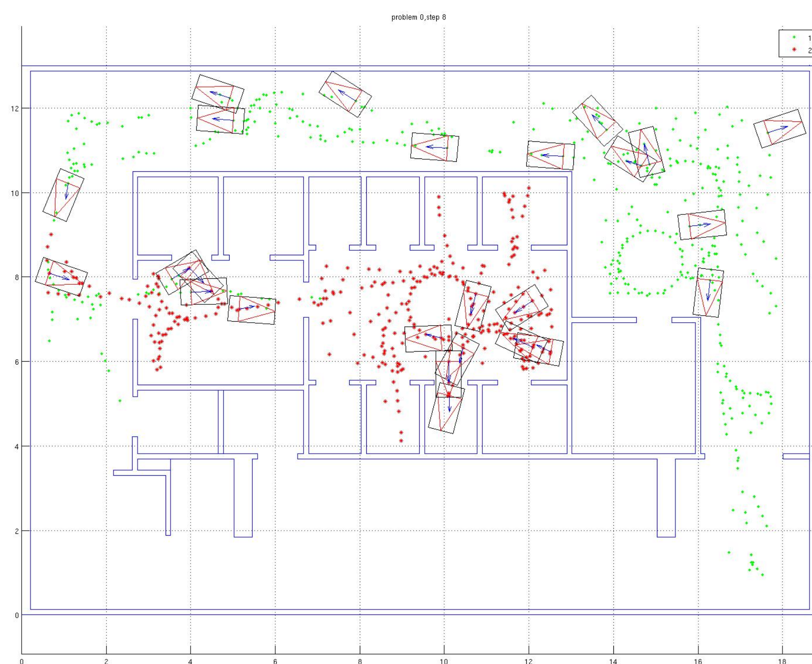


(b)

Figura 6.5: Crescita degli alberi durante vari step dell'esecuzione dell'algoritmo (1 di 2)



(a)



(b)

Figura 6.6: Crescita degli alberi durante vari step dell'esecuzione dell'algoritmo (2 di 2)

problem 3, step 20

Figura 6.7: Pianificazione con rotazione necessaria

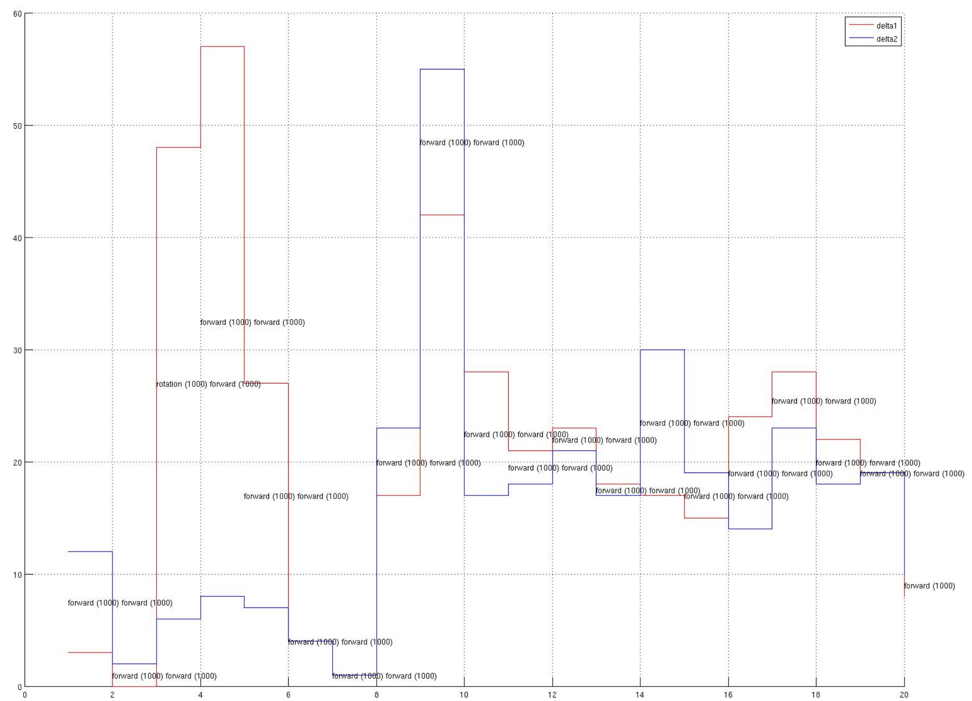


Figura 6.8: Set di Input utilizzati nella generazione dell'albero

Capitolo 7

Conclusioni e sviluppi futuri

"La pianificazione strategica va in crisi quando il futuro si rifiuta di assumere il ruolo assegnatoli dai pianificatori"

Edward De Bono, scrittore e psicologo maltese

L'implementazione di un planner globale (cioè la ricerca di quali azioni svolgere per muoversi tra due punti che non sono sulla stessa mappa) esula dagli obiettivi del pianificatore qui realizzato.

Questa funzione può comunque facilmente essere aggiunta partendo dalla definizione della mappa in cui sono definiti i punti di uscita e entrata per passare da una mappa all'altra. Questi punti sono inclusi nella struttura XML della mappa (Figura 7.1) e sono chiamati *portal*.

Formalizzando questa *mappa delle mappe* come un grafo, si riesce a integrare il pianificatore locale e il pianificatore globale attraverso semplici algoritmi come $A*$. La costruzione del grafo è pressochè immediata, considerando che a ciascun *portal* che collega due mappe è già associato un attributo *cost*, che funge da euristica per l'algoritmo di pianificazione globale.

Per esempio per andare dalla posizione corrente S sulla mappa A alla destinazione G sulla mappa B è sufficiente andare dalla posizione corrente all'uscita della mappa A con RRT, calcolare tramite A* come andare dall'uscita della mappa A all'ingresso della mappa B, e infine calcolare con RRT il percorso dall'ingresso della mappa B alla destinazione desiderata G, così come illustrato nella Figura 7.2.

Attualmente l'algoritmo di ricerca della pianificazione non tiene conto degli ostacoli mobili che possono entrare e interferire nella traiet-

toria. Questo compito è affidato a un agente che si occupa di gestire i comportamenti reattivi per l'evitamento ostacoli ed effettua l'*obstacle avoidance*. In futuro si potrà pensare di far interagire questi due agenti in modo da arrivare a un unico algoritmo di pianificazione con la capacità di aggiornare la mappa con ostacoli rilevati dinamicamente. A questo proposito si veda il paragrafo 3.4 nella pagina 18 sull'algoritmo *Vector Field Histogram*, che svolge questa funzione.

Attualmente per il rilevamento delle collisioni è utilizzata la libreria PQP (Proximity Query Package).

Questa svolge i tre seguenti compiti

- Rilevamento delle collisioni
- Calcolo distanza
- Verifica della distanza di tolleranza

Lo stesso compito potrà essere svolto da un altra libreria, ANN (Approximate Nearest Neighbor Searching), che promette maggiori performance, ma il cui sviluppo è per ora a un punto fermo e i risultati attuali non sono considerati stabili.

Ann è una libreria scritta in C++, che supporta le strutture dati e algoritmi per la ricerca del nodo più vicino in spazi di dimensione arbitrarie.

Nel problema della ricerca del nodo più vicino è data una serie di punti in uno spazio n-dimensionale. Questi punti sono preelaborati in una struttura dati, in modo che per ogni dato punto q, venga trovato in modo efficiente il più vicino, o in generale l'insieme dei punti p più vicini ai punti di Q. La distanza tra due punti può essere definita in molti modi: ANN presuppone che le distanze siano misurate con una classe di funzioni di distanza basate sulle metriche di Minkowski. Queste includono la distanza euclidea, la distanza di Manhattan, e la distanza massima.

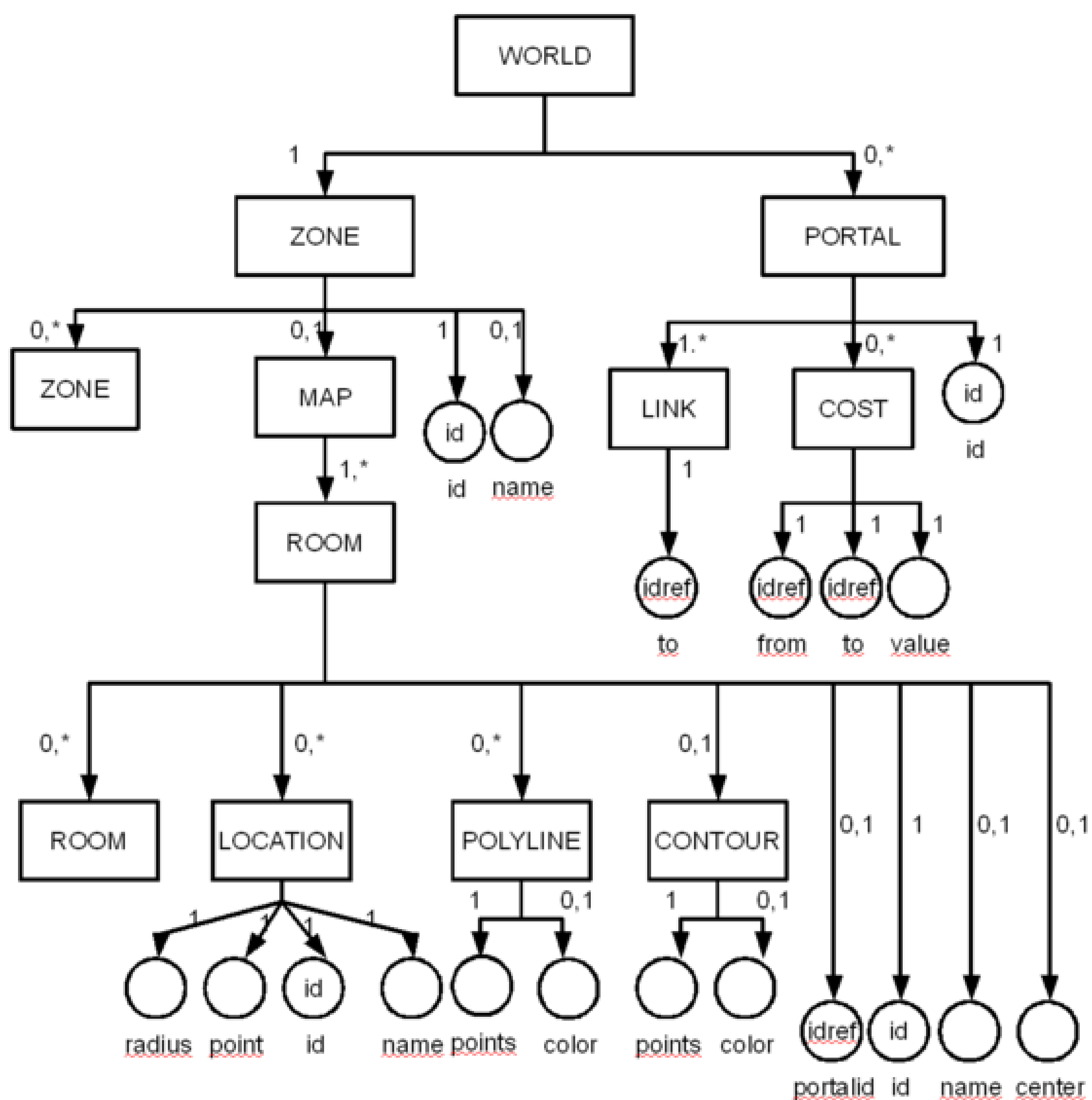


Figura 7.1: Struttura XML della mappa

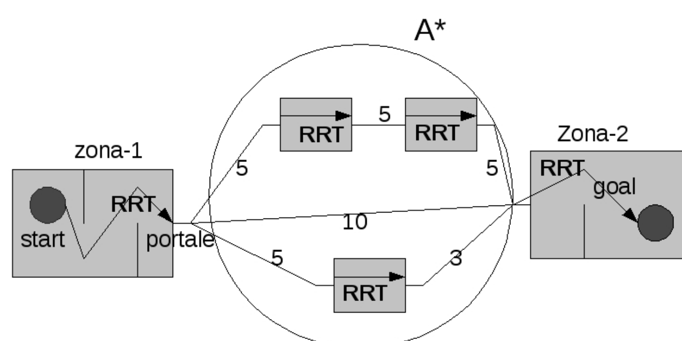


Figura 7.2: Pianificatore globale

Bibliografia

[1] http://en.wikipedia.org/wiki/Motion_planning.

[2] http://msl.cs.uiuc.edu/rrt/about.html.

[3] Simone Ceriani. Sviluppo di una carrozzina autonoma d'ausilio ai disabili motori. Master's thesis, Politecnico di Milano, 2008.

[4] Simone Ceriani, Bernardo DalSeno, Giulio Fontana, Matteo Matteucci, Davide Migliore, and Rossella Blatt. Brain control of a smart wheelchair. *Intelligent autonomous systems 10 : IAS-10*, page 221, 2008.

[5] M. Habib. *Can plannig and reactive systems realize an autonomus navigation*. International Symposium on Robotics (ISR99), 11(2), 1999.

[6] L. Iliadis, I. Maglogiannis, G. Tsoumakas, I. Vlahavas, M. Bramer. *Artificial Intelligence Applications and Innovations*. Proceedings of the 5th IFIP Conference on Artificial Intelligence Applications and Innovations, Thessaloniki, Greece, 2009.

[7] Jean-Claude Latombe. *Robot motion planning*. Hardcover, 1990.

[8] Jean-Paul P. Laumond. *Robot motion planning and control*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1998.

[9] S.M Lavalle. *Rapidly-exploring Random Trees: a new tool for path planning*. Computer Science Dept, Iowa State University, Tech. Rep. TR: 98 11. Retrieved on 2008.06.30, 1998.

[10] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[11] Xiaoshan Pan. Wheelchair robotic motion planning. *Civil and Environmental Engineering, Stanford University*.

[12] Peter Norvig Stuart J. Russell. *Intelligenza artificiale. Vol. 2: Un approccio moderno.* Pearson Education Italia, 2005.

[13] Salvatore Nicosia, Francesco Martinelli. *Dispense del corso di robotica industriale (pianificazione del moto dei robot).* Technical report, Università degli Studio di Roma Tor Vergata, 2000.

[14] Fabio Sora. *FollowMe: composizione "informata" di reattività e pianificazione in un robot guida*. PhD thesis, Politecnico di Milano, 2005.

[15] Steven M. Lavalle, James J. Kuffner Jr. *Rapidly-exploring Random Trees: progress and prospects*. Algorithmic and Computational Robotics: New Directions, 2000.

[16] J. Borenstein I. Ulrich. Vfh+: Reliable obstacle avoidance for fast mobile robots. *Proceeding of the 1998 IEEE Intenational Conference on Robotics and Automation, Leuven, Belgium*, pages 1572–1577, 1998.

[17] J. Borenstein I. Ulrich. Vfh*: Local obstacle avoidance with look-ahead verification. *Proceeding of the 2000 IEEE International Conference on Robotics and Automation, San Francisco, CA*, pages 2505–2510, 2000.

[18] J. Borenstein Y Koren. *Vector Field Histogram: Fast Obstacle Avoidance for Mobile Robots*. IEEE Trans. Robot. Automat, Vol 7, No. 3, pp. 278 - 288, June 1991.