ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA E SCIENZE INFORMATICHE

Sviluppo di nuovi algoritmi di pianificazione per sistemi non olonomici e con dinamica non lineare

TESI DI LAUREA IN
SISTEMI INTELLIGENTI ROBOTICI

RELATORE:
Chiar.mo Prof. Ing.
Andrea Roli

PRESENTATA DA:
Lorenzo Rocca

CORRELATORE:
Ing. Nicola Mimmo

SESSIONE II
ANNO ACCADEMICO 2015/2016

Ai miei genitori e a mio fratello

Indice

Elenco delle figure

Introduzione

Tra i molti problemi trattati nell'ambito dei sistemi robotici vi è quello della *pianificazione*: attraverso questo processo viene determinata la sequenza di attività da compiere per portare a termine un obiettivo. Se per un essere umano si tratta di un'attività spontanea e naturale che ne determina le capacità di problem solving, per una macchina la pianificazione è più complicata e molti sono stati gli sforzi fatti per comprendere come codificarla attraverso un algoritmo. I risultati sono stati validi e utilizzati non solo nella robotica: gli algoritmi di pianificazione si sono diffusi in ambiti industriali, manifatturieri, farmaceutici, e video-ludici. La rapida crescita in questi campi è indice che possono esservi molti altri ambiti di applicazione. Il termine pianificazione ha però un significato diverso in base al contesto in cui viene utilizzato. In **robotica** fa riferimento alla traduzione di un compito, definito ad alto livello da un essere umano, in una sequenza di mosse. Nella **teoria del controllo** il termine denota la costruzione di input per un sistema dinamico e non lineare; l'obiettivo è quello di guidare l'agente da uno stato iniziale ad uno specifico. Data la vastità dell'argomento è inevitabile che la pianificazione si specializzi in diversi aspetti del problema. Il *motion planning* è un caso particolare di pianificazione in cui sono ignorati i vincoli dinamici e differenziali e prevede traslazioni e rotazioni. Nella sua forma più semplice, l'unico vincolo previsto è l'assenza di collisioni. Tuttavia in molti sistemi l'inerzia e la potenza degli attuatori limitano le possibilità di decisioni arbitrarie sul robot e può essere necessario controllarne invece la velocità, la quale può essere anch'essa vincolata. Per questi motivi il *kinodynamic motion planning* estende gli aspetti di cinematica, includendo la dinamica del corpo e i vincoli sul controllo.

Qualunque sia il problema trattato, la capacità di pianificare permette di aumentare il grado di autonomia di un sistema. Con il tempo, questa caratteristica ha attirato l'attenzione di molti istituti di ricerca. Il progetto SHERPA[1] è un progetto europeo in cui un consor-

[1] www.sherpa-project.eu

zio di aziende, università e associazioni no-profit, collaborano per migliorare le attività di soccorso in ambiente alpino. Consiste nella realizzazione di un sistema collaborativo composto da diversi agenti sia umani sia robotici. Elemento importante per il sistema è proprio la capacità dei robot di operare in modo autonomo. Tra i diversi aspetti che il progetto tratta per il raggiungimento dell'obiettivo finale vi è anche quello della pianificazione di agenti eterogenei. In questo senso il problema della pianificazione deve essere trattato sia per rover di terra, veicoli a controllo differenziale per il trasporto di attrezzatura, sia per Unmanned Aerial Vehicle (UAV), droni multi-rotore per la perlustrazione dell'area.

Il presente lavoro di tesi nasce nel contesto del progetto SHERPA con l'obiettivo di fornire un pianificatore in grado di controllare il rover a disposizione dell'operatore. In questo contesto si prenderanno in esame diversi aspetti del sistema, come la sua dinamica e i vincoli; inoltre verrà realizzato un software utilizzabile e facilmente estendibile. Il pianificatore dovrà anche presentare performance appropriate per il soccorso alpino, come tempi di risposta accettabili e soluzioni realmente attuabili.

Il Capitolo 1 si occuperà di analizzare lo stato dell'arte relativo alla pianificazione. Inizialmente verrà presentata la formulazione dei due principali problemi, quello di motion planning e di kinodynamic motion planning. Successivamente verranno esposte le maggiori alternative per la loro risoluzione.

A seguito della comprensione delle tecniche disponibili, il Capitolo 2 verrà dedicato alla definizione dei nuovi algoritmi per la pianificazione. Verranno delineate le principali limitazioni e problematiche delle tecniche note. Sfruttando i vincoli individuati sarà possibile ideare un nuovo approccio al problema.

Il Capitolo 3 riguarderà lo sviluppo del software di pianificazione per la guida di un rover a controllo differenziale: verranno fornite dettagliate informazioni relative all'analisi e alla progettazione del software. Trattandosi di un sistema con dinamica non lineare e volendo rispettare i vincoli fisici del robot, in questo capitolo troveranno applicazione e definizione concreta gli algoritmi descritti nei capitoli precedenti.

Nel Capitolo 4 saranno analizzate le performance del sistema. Si confronteranno gli algoritmi sviluppati con quelli noti allo stato dell'arte in modo da fornire informazioni significative sia per l'utilizzo del pianificatore nel contesto del progetto SHERPA, sia per ulteriori analisi scientifiche.

Capitolo 1

Motion Planning: Stato dell'arte

La pianificazione consiste nel sintetizzare una sequenza di azioni che, eseguite da un agente a partire da uno stato iniziale del mondo, conducono al raggiungimento di uno stato desiderato. Questa definizione, seppur corretta, è molto generale. Il problema di motion planning può essere definito come segue: dati al robot la descrizione della sua dinamica, una descrizione dell'ambiente, uno stato iniziale, un insieme di stati goal e, opzionalmente, una rappresentazione esplicita degli ostacoli nello spazio delle configurazioni, il problema consiste nel determinare una sequenza di input che portino il robot da uno stato iniziale ad uno degli stati obiettivo, obbedendo ai vincoli imposti dall'ambiente.

La prima parte di questo capitolo verrà dedicata alla formulazione del problema di pianificazione in esame, noto come motion planning, e di una sua derivazione per un caso particolare, detto kinodynamic planning. Nella seconda parte verranno invece illustrate alcune tecniche risolutive per i due problemi descritti in precedenza.

1.1 Formulazione del problema

Quello della pianificazione è un problema trattato da diverse discipline e in diversi contesti. Alcuni elementi sono ricorrenti e la loro definizione può aiutare nella comprensione del problema. La pianificazione coinvolge uno **spazio**, detto degli stati o delle configurazioni, il quale rappresenta tutte le possibili situazioni che possono verificarsi. Uno stato o una configurazione dipende dallo specifico problema e possono essere sia continui sia discreti. Tutti i problemi di pianificazione riguardano una sequenza di azioni con una forte

connotazione temporale. Il **tempo** può essere modellato esplicitamente, ad esempio chiedendo che un robot agisca nel minor tempo possibile, o in modo implicito, semplicemente indicando che le azioni devono essere seguite in successione. Inoltre un piano genera delle **azioni** che manipolano lo stato. Spesso, ma non sempre, viene specificato come lo stato cambia quando un'azione viene applicata. Infine viene previsto un **criterio** che descrive il risultato di un piano. Esistono due criteri di pianificazione: quello di ammissibilità e quello di ottimalità, descritti nel seguito.

In questa sezione verranno presentate due formulazioni dei problemi di pianificazione: quello di motion planning e kinodynamic planning, descritti secondo gli elementi presentati.

1.1.1 Motion planning

La formulazione del problema di *motion planning* presentata in [5] è definita come *la ricerca di un percorso continuo in uno spazio continuo*; una rappresentazione a livello grafico è osservabile in Figura 1.1. Sia $\mathcal{C} \subseteq \mathbb{R}^n$ il modello del mondo, definito come **spazio delle configurazioni** del problema. Indichiamo con $\mathcal{C}_{obs} \subset \mathcal{C}$ le configurazioni in collisione con degli ostacoli e $\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}$ l'insieme di configurazioni ammissibili (o collision-free). La configurazione $x_I \in \mathcal{C}_{free}$ viene definita come configurazione iniziale, mentre $x_G \in \mathcal{C}_{free}$ come configurazione finale (goal); la coppia (x_I, x_G) viene anche spesso denominata *query-pairs*. Un algoritmo in grado di computare un percorso dalla configurazione iniziale a quella finale o di indicare che tale percorso non esiste, è detto pianificatore.

Un **percorso** viene definito come una funzione $\sigma : [0, 1] \to \mathcal{C}$ continua che crea un ordinamento per una successione di punti dello spazio delle configurazioni. Un path è detto *collision-free* se $\sigma(\tau) \in \mathcal{C}_{free}\ \forall \tau \in [0, 1]$. Convenzionalmente, si utilizzerà il termine «percorso» per indicare un percorso che gode della proprietà di collision-free, salvo esplicitamente indicato. Per alcuni problemi reali può anche essere interessante modellare la presenza di diversi goal; la formulazione precedente può quindi essere estesa considerando una regione di goal $\mathcal{C}_{goal}$ sottoinsieme aperto di $\mathcal{C}_{free}$.

Il problema di motion planning può essere definito seguendo due criteri, quello di ammissibilità e quello di ottimalità. Nel problema di pianificazione ammissibile si desidera determinare un percorso *ammissibile*, ovvero un percorso per cui vale $\sigma(0) = x_{init}$ e

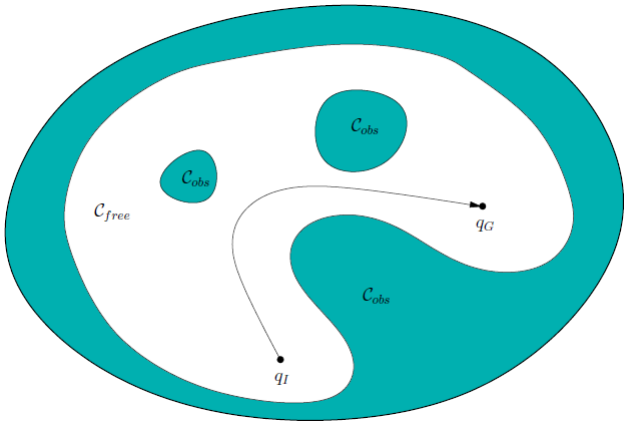


Figura 1.1: Rappresentazione concettuale del problema di Motion Planning

$\sigma(1) \in cl(\mathcal{C}_{goal})$, dove $cl(.)$ indica l'operazione di chiusura dell'insieme. Se il percorso esiste, deve essere restituito, altrimenti deve essere indicato un fallimento.

Sia Σ l'insieme di tutti i percorsi e Σ_{free} l'insieme di tutti i percorsi collision-free. Sia $\mathrm{c} : \Sigma \to \mathbb{R}_{\geq 0}$ una funzione in grado di restituire il costo di un percorso non triviale. Assumiamo essere questa monotòna e limitata; diremo allora che σ^* è un percorso *ottimo* se $\sigma^* = min\{c(\sigma) \ : \ \sigma \in \Sigma_{free}\}$. Il problema di pianificazione ottima richiede, sotto le ipotesi precedenti, di determinare il percorso ottimo o restituire un fallimento se non esiste.

1.1.2 Kinodynamic motion planning

Il termine *kinodynamic planning* è stato introdotto in [4] per indicare una classe di problemi di motion planning in cui sono posti dei vincoli alla velocità e all'accelerazione del robot. Più recentemente il termine è stato utilizzato per denotare i problemi di motion planning che coinvolgono anche la dinamica del corpo: in questo senso qualunque problema che coinvolga dei vincoli differenziali del secondo ordine o superiore può essere considerato come un problema di kinodynamic planning. Proprio per questo motivo, accade spesso che un problema di pianificazione simile, quello di *trajectory planning*, sia utilizzato come sinonimo. Sebbene questo non sia storicamente esatto, può essere corretto trattare le due classi di problemi con metodi di pianificazione sufficientemente generali da risolverle entrambe.

Nei problemi di motion planning, lo spazio delle configurazioni descrive solo la posizione e l'orientazione del robot. Per introdurre i vincoli relativi alla dinamica questo viene

esteso con le derivate nel tempo delle componenti. Questo nuovo spazio prende il nome di **spazio degli stati**. Siano quindi $X \subset \mathbb{R}^n$ e $U \subset \mathbb{R}^m$ gli insiemi compatti su cui viene definito il sistema dinamico

$$\dot{x} = f(x(t), u(t)) \qquad x(0) = x_0$$

dove $x(t) \in X$, $u(t) \in U$ per ogni t, $x_0 \in X$, ed f è una funzione continua e derivabile nelle sue variabili. Denotiamo l'insieme di tutte funzioni limitate e misurabili definite da $[0, T]$ ad X, per ogni $T \in \mathbb{R}_{>0}$ con $\mathcal{X}$ e definiamo $\mathcal{U}$ similmente. Le funzioni in $\mathcal{X}$ sono dette *traiettorie*, mentre quelle in $\mathcal{U}$ prendono il nome di *controlli*.

Dato il dominio X, la regione di ostacoli X_{obs}, la regione di goal X_{goal} e una funzione f continua e derivabile che descrive la dinamica del sistema, il problema di kinodynamic motion planning richiede di determinare un controllo $u \in \mathcal{U}$ tale che l'unica traiettoria corrispondente $x \in \mathcal{X}$, con $\dot{x} = f(x(t), u(t))$, eviti tutti gli ostacoli, ovvero $x(t) \in X_{free}$ per ogni t, e raggiunga la regione di goal, ovvero $x(T) \in X_{goal}$. In questo lavoro verrà trattata anche la versione ottima del problema il cui obiettivo è quello di risolvere il problema precedente, minimizzando al contempo una funzione di costo designata.

1.2 Algoritmi esistenti

Esistono diverse filosofie per risolvere i problemi di motion planning e kinodynamic motion planning proposti in precedenza: una di queste è quella *sampling-based*, ovvero basata sul campionamento. L'idea su cui si basa è evitare di costruire, ove non sia possibile, lo spazio degli ostacoli, sondando invece l'intero spazio con uno schema di campionamento. Questa famiglia di tecniche deve però essere affiancata da un modulo di collision detection che il pianificatore dovrà considerare come una black box, il cui compito sarà quello di verificare che nessun tratto del percorso giaccia su un'ostacolo.

Nel seguito verranno descritte brevemente alcune funzioni frequentemente utilizzate dagli algoritmi sampling-based. Saranno poi presentate le principali tecniche per la risoluzione del problema di motion planning, dette RRT e RRT*, nonché la versione per risolvere il problema di kinodynamic motion planning, noto come Kinodynamic-RRT*.

1.2.1 Funzioni primitive

Dato un grafo $G = (V, E)$ con $V \subset \mathcal{C}$ e un punto $x \in \mathcal{C}$, le funzioni Nearest e Near permettono di individuare dei punti adiacenti sulla base dei seguenti criteri:

- Nearest($G = (V, E), x$):= $argmin_{v \in V} ||x - v||$, ovvero la funzione restituisce il vertice v più vicino ad x che appartiene al grafo. In questo contesto viene considerata una metrica Euclidea, ma non è l'unica alternativa.

- Near($G = (V, E), x, r$):= $\{v \in V : v \in \mathcal{B}_{x,r}\}$, dove $\mathcal{B}_{x,r} = \{y \in \mathbb{R}_d : ||y - x|| \leq r\}$ denota una sfera di raggio r centrata in x. In altri termini la funzione restituisce tutti i punti appartenenti a V che si trovano entro una distanza r da x, secondo una certa metrica, in questo caso Euclidea.

Dati due punti $x, y \in \mathcal{C}$, le funzioni Steer e CollisionFree permettono di ottenere informazioni di carattere spaziale tra i due punti:

- Steer(x, y):= $argmin_{z \in \mathcal{B}_{x,\eta}} ||z - y||$, ovvero la funzione restituisce un punto z appartenente alla sfera di raggio η tale che la distanza da y è minima.

- CollisionFree(x, y) restituisce true se tutti i punti del segmento di retta avente x, y come estremi giacciono in $\mathcal{C}_{free}$.

1.2.2 Rapidly-exploring Random Tree

Rapidly exploring Random Tree (o RRT) è stato presentato in [6] ed è uno degli algoritmi sampling-based in grado di esplorare lo spazio delle configurazioni $\mathcal{C}$. Con l'algoritmo viene costruito un RRT, ovvero un albero, i cui nodi sono degli stati appartenenti a $\mathcal{C}_{free}$ e i cui archi, congiungenti i nodi, sono dei percorsi che giacciono in $\mathcal{C}_{free}$. Dato x_{init} lo stato iniziale, un RRT $T = (V, E)$ viene costruito come illustrato dall'Algoritmo 1.

Il primo elemento aggiunto ai vertici di T è x_{init}, per definizione appartenente a $\mathcal{C}_{free}$. Ad ogni iterazione un nuovo stato x_{rand} viene selezionato da $\mathcal{C}_{free}$ e viene individuato lo stato più vicino a x_{rand} appartenente a T, denotato come $x_{nearest}$. Attraverso la funzione Steer viene individuato un punto, x_{new}, il più possibile vicino a x_{rand}, ma che rimanga entro un raggio prefissato da $x_{nearest}$. Se è possibile collegare i due punti, ovvero non vi sono collisioni nel segmento che si crea, allora x_{new} può essere aggiunti all'albero $T = (V, E)$. Un esempio dell'algoritmo è mostrato in Figura 1.2.

Algoritmo 1 Algoritmo Rapidly-exploring Random Tree

```
1  RRT(C_free, x_init, C_obs)
2      V <- {x_init}; E <- empty;
3      for i = 1, ..., n do
4          x_rand <- SampleFree_i;
5          x_nearest<-Nearest(T =(V,E), x_rand);
6          x_new<-Steer(x_nearest, x_rand);
7          if ObtacleFree(x_nearest, x_new) then
8              V <- V U{x_new};
9              E <- E U{(x_nearest, x_new)};
10     return T =(V,E)
```

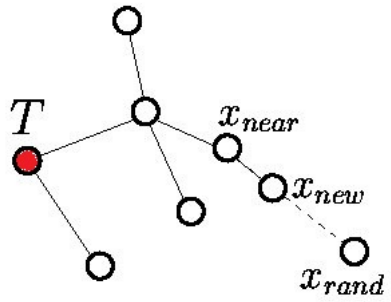


Figura 1.2: Costruzione di un Rapidly-exploring Random Tree

Algoritmo 2 Sampling dello spazio indirizzato verso il goal

```
1 Biased_SampleFree()
2     toss <- COIN_TOSS();
3     if toss = head then
4         return x_goal;
5     else
6         return SampleFree_i;
```

Si noti che l'algoritmo, in questa versione definita in [5], viene ripetuto n volte, costruendo un albero di dimensione minore o uguale ad n. Altre versioni del medesimo algoritmo permettono di costruire un albero di dimensione K fissata a priori ([6]) o proseguire nella costruzione fino a quando uno stato obiettivo non viene raggiunto ([2]). È inoltre possibile costruire due RRT, uno avente come radice lo stato iniziale, uno lo stato finale e introdurre una procedura di merge, come descritto in [7]. Tra tutte le versioni, la più interessante è forse quella che introduce un *bias* nell'esplorazione dello spazio: per fare fronte alla lentezza nella convergenza dell'algoritmo è possibile indirizzare la costruzione dell'albero verso il goal, modificando il campionamento come mostrato dall'Algoritmo 2. Questa tecnica, unita a quella di Steering, fornisce una convergenza più veloce, senza perdere generalità nell'esplorazione dello spazio delle configurazioni.

L'algoritmo, in tutte le sue varianti, gode di alcune interessanti proprietà, dimostrate formalmente in [5, 6].

- È intuitivo che l'albero costruito rimane sempre connesso, anche nel caso in cui si abbiamo pochi nodi o pochi campioni.
- RRT è utilizzabile come pianificatore per il problema del motion planning. Inoltre l'assenza di uno stato obiettivo x_{goal} definito a priori fa sì che l'algoritmo sia utilizzabile per molte query-pairs purché, per ogni coppia, valga $q_I = x_{init}$. In altre parole RRT può essere calcolato senza conoscere a priori lo stato obiettivo, ma solo lo stato iniziale, rendendolo in grado di risolvere query di natura one-to-many.

Inoltre RRT è probabilisticamente completo sotto alcune ipotesi molto generali. Un algoritmo è **probabilisticamente completo** se, per ogni problema di pianificazione $(\mathcal{C}_{free}, x_{init}, \mathcal{C}_{goal})$ con soluzione ammissibile, vale

$$\lim_{n\to\infty} \mathbb{P}(\{\exists x_{goal} \in V_n \cap \mathcal{C}_{goal} | x_{init}\, è\, connesso\, a\, x_{goal}\, in\, G_n\}) = 1$$

ovvero la probabilità di determinare un percorso ammissibile tende ad uno per un numero di campioni utilizzati tendente ad infinito.

1.2.3 Optimal Rapidly exploring Random Tree

L'algoritmo RRT* è stato presentato in [5] e si caratterizza per essere la versione ottima dell'algoritmo RRT. Costruire un RRT* può essere fatto semplicemente riscrivendo, anche dinamicamente, l'albero prodotto da RRT, garantendo che ogni vertice sia raggiungibile attraverso un cammino a costo minimo.

Prima di presentare l'algoritmo è necessario definire alcune funzioni:

- Line: dati due punti $x_1, x_2 \in \mathbb{R}^d$, la funzione $\mathrm{Line}(x_1, x_2) : [0, s] \to \mathcal{C}$ denota il percorso in linea d'aria tra x_1, x_2.
- Parent: dato un albero $T = (V, E)$, sia $\mathrm{Parent} : V \to V$ la funzione che per un qualunque vertice $v \in V$ restituisce l'unico vertice $u \in V$ tale che $(u, v) \in E$. Per convenzione se v_0 è la radice dell'albero allora $\mathrm{Parent}(v_0) = v_0$.
- Cost: dato un nodo in V, la funzione $\mathrm{Cost} : V \to \mathbb{R}_{\geq 0}$ restituisce il costo del percorso dalla radice dell'albero al nodo.

L'algoritmo RRT* (Algoritmo 3) aggiunge punti all'insieme dei vertici come RRT. Una volta determinato il punto candidato x_{new} vengono verificate le connessioni con i punti in X_{near}, ovvero i vertici posti entro la distanza $r(card(V))$ e appartenenti a V. Tuttavia non tutti i possibili vertici vengono aggiunti all'albero:

1. un arco può essere creato se la connessione a x_{new} avviene attraverso un percorso a costo minimo;
2. un arco da un nodo in X_{near}a x_{new} può essere creato solo se il percorso così stabilito ha costo minore di quello corrente. In questo caso l'arco che collega x_{near} al suo predecessore viene rimosso, per mantenere la struttura ad albero.

L'algoritmo si caratterizza per la presenza della procedura di *aggiornamento* (o update) evidenziata alla linea 18 dell'Algoritmo 3. Come si vedrà nel seguito, questa è molto importante e troverà applicazione in tutte le tecniche mirate a determinare la soluzione ottima al problema di pianificazione.

Algoritmo 3 Optimal Rapidly exploring Random Tree

```
1  RRT*(C_free, x_init, C_obs, n, c)
2      V <- {x_init}, E <- empty
3      for i=0 to n do
4          x_rnd <- SampleFree_i
5          x_nearest <- Nearest(G,x_rnd)
6          x_new <- Steer(x_nearest, x_rnd)
7          if CollisionFree(x_nearest, x_new) then
8              X_near <- Near(G,x_new,min(r,s))
9              V.add(x_new)
10             // --- non tutte le possibili connessioni sono aggiunte
11             x_min <- x_nearest;
12             c_min <- Cost(x_nearest)+c(Line(x_nearest,x_new))
13             foreach x_near in X_near do
14                 if CollisionFree(x_near,x_new) && Cost(x_near)+c(Line(
                       x_near,x_new) < c_min then
15                     x_min <- x_near
16                     c_min <- Cost(x_near)+c(Line(x_near,x_new))
17             E.add(x_min,x_new)
18             // Procedura di update: riscrive l'albero
19             foreach x_near in X_near do
20                 if CollisionFree(x_near,x_new) && Cost(x_new)+c(Line(
                       x_new,x_near) < Cost(x_near) then
21                     parent <- Parent(x_near)
22                     E.remove(parent,x_near)
23                     E.add(x_new,x_near)
24     return T=(V,E)
```

Il valore di $r(card(V))$ viene scelto in funzione della cardinalità dell'insieme V come:

$$r(card(V)) := min\left\{\gamma_{RRT} \cdot \left(\frac{log(card(V))}{card(V)}\right)^{1/d}, \eta\right\}$$

dove η è il parametro della funzione di steering e γ_{RRT} è un parametro euristico del problema soggetto al vincolo

$$\gamma_{RRT} > \gamma^*_{RRT} = 2\left(1+\frac{1}{d}\right)^{1/d}\left(\frac{\mu(\mathcal{C}_{free})}{\zeta_d}\right)$$

Può essere derivata un'altra versione dell'algoritmo, nota come k-nearest RRT*, se si limitano i tentativi di connessione ai soli k vicini. Il numero di vicini è una funzione della dimensione di V, definita come:

$$k(card(V)) := k_{RRT} log(card(V))$$

dove $k_{RRT} > k^*_{RRT} = e(1 + 1/d)$. Si noti che k^*_{RRT} è una costante dipendente solo da d e non dipende dalla dimensione dell'istanza del problema, come invece è per γ^*_{RRT}. Inoltre $k_{RRT} = 2e$ è una scelta valida per tutte le istanze del problema.

Utilizzando l'approccio di RRT*, basato su aggiornamento dei percorsi, è possibile determinare la soluzione ottima in modo asintotico. Sia c^* il costo della soluzione ottima e Y_n la variabile casuale corrispondente al costo della soluzione a costo minimo inclusa nel grafo restituita dopo n iterazioni (o campioni). Un algoritmo si dice **asintoticamente ottimo** se, per ogni problema di pianificazione $(\mathcal{C}_{free}, x_{init}, \mathcal{C}_{goal})$ e funzione di costo c che determina la soluzione ottima c^*, vale

$$\mathbb{P}\left(\left\{\limsup_{n\to\infty} Y_n = c^*\right\}\right) = 1$$

Dato che vale $Y_n \geq c^* \forall n \in \mathbb{N}$, la proprietà di ottimalità asintotica implica che $\lim_{n\to\infty} Y_n$ esiste ad è uguale a c^*. Ovviamente perché un algoritmo sia asintoticamente ottimo deve essere anche probabilisticamente completo. Inoltre la probabilità che un algoritmo converga alla soluzione ottima è uguale a zero oppure uno.

1.2.4 Kinodynamic RRT*

Negli ultimi anni l'approccio risolutivo di RRT* ha trovato applicazione anche nei problemi di kinodynamic motion planning. In [13] è stato proposto un metodo per determinare la traiettoria collision-free ottima, supponendo di avere a disposizione un metodo per calcolare la traiettoria ottima π^* che congiunge due punti e una funzione di costo c.

L'Algoritmo4 costruisce un albero di traiettorie nello spazio degli stati la cui radice è lo stato iniziale. Ad ogni iterazione viene campionato uno stato $\mathbf{x}_i \in X_{free}$ per essere aggiunto all'albero. Il predecessore del campione è selezionato tra tutti i nodi esistenti: la traiettoria scelta sarà quella il cui costo da x_{init} è minimo, purché rispetti il vincolo $c^*[\mathbf{x}, \mathbf{x}_i] < r$ e sia collision-free. Dopo aver effettuato l'espansione dell'albero, viene eseguito l'update, il cui obiettivo è quello di diminuire il costo dei percorsi esistenti utilizzando il nodo appena aggiunto. Per tutti gli stati $\mathbf{x}$ dell'albero per cui vale $c^*[\mathbf{x}_i, \mathbf{x}] < r$, se la traiettoria risultante è collision-free e permette di raggiungere il nodo con un costo minore, allora $\mathbf{x}_i$ è impostato come predecessore di $\mathbf{x}$. Dopo aver effettuato l'update, l'algoritmo può procedere con una nuova iterazione. Per un numero di iterazioni tendenti ad infinito, emergerà nell'albero una traiettoria ottima tra $\mathbf{x}_{init}$ e $\mathbf{x}_{goal}$ ottenuta come concatenazione di traiettorie ottime tra una successione di stati.

L'algoritmo presenta alcune motevoli differenze rispetto la versione standard di RRT*. La prima è l'introduzione esplicita dello stato goal durante l'operazione di update dell'albero: questa modifica permette di raggiungere $\mathbf{x}_{goal}$ considerando le connessioni da ogni campione e mantenendole sempre aggiornate. La seconda differenza risiede nell'assenza di metodo di steering che permette di far crescere l'albero *verso* il punto campione, realizzando solo una traiettoria parziale. Tuttavia l'aumento della dimensionalità dello spazio degli stati rende la definizione di una procedura di steering non triviale e quindi gli autori hanno preferito verificare una traiettoria diretta con il punto campione. Anche con questi cambiamenti l'algoritmo preserva la proprietà di ottimalità asintotica.

1.2.4.1 Calcolo della traiettoria ottima

Per risolvere il problema di kinodynamic motion planning ottimo, occorre essere in grado di calcolare la traiettoria ottima $\pi^*[\mathbf{x}_0, \mathbf{x}_1]$ (e il suo costo ottimo $c^*[\mathbf{x}_0, \mathbf{x}_1]$) tra due punti $\mathbf{x}_0 \in X_{free}$ e $\mathbf{x}_1 \in X_{free}$. L'approccio descritto in [13] richiede che la dinamica, rappresentata dal sistema **lineare** $\dot{\mathbf{x}}[t] = A\mathbf{x}[t] + B\mathbf{u}[t] + \mathbf{c}$, sia completamente controllabile e

Algoritmo 4 Algoritmo Kinodynamic-RRT*

```
1  Kinodynamic-RRT*(X_free, x_init, x_goal, X_obs, n, c)
2      V <- {x_init}, E <- empty
3      for i=0 to n do
4          x_i <- SampleFree_i
5          foreach x in V do
6              opt_pi = CalculateOptimalTrajectory(x, x_i)
7              if c(opt_pi)<r && CollisionFree(opt_pi) && Cost(x)+c(opt_pi
                   ) < c_min then
8                  x_min = x
9                  c_min = Cost(x)+c(opt_pi)
10         V.add(x_i)
11         E.add(x_min, x_i)
12
13         // Riscrive l'albero, dando la possibilità di usare x_i
14         foreach x in Union(V,{x_goal}) do
15             opt_pi = CalculateOptimalTrajectory(x_i, x)
16             if c(opt_pi)<r && CollisionFree(opt_pi) && Cost(x_i)+c(
                   opt_pi) < Cost(x) then
17                 parent <- Parent(x)
18                 E.remove(parent,x)
19                 E.add(x_i,x)
20     return T=(V,E)
```

raggiungibile. Una traiettoria è, in questo caso, definita dalla tupla $\pi = (\mathbf{x}[], \mathbf{u}[], \tau)$, in cui τ è il tempo di arrivo o durata della traiettoria, $\mathbf{x}[] \in \mathcal{X}$ gli stati lungo la traiettoria e $\mathbf{u}[] \in \mathcal{U}$ i controlli associati. La funzione di costo proposta è descritta dal seguente integrale

$$c[\pi] = \int (1 + \mathbf{u}[t]^T R \mathbf{u}[t]) dt,$$

in cui $R \in \mathbb{R}^{m \times m}$, positiva, costante e nota, è la matrice in grado di bilanciare il costo dei controlli e la durata della traiettoria. Questa funzione di costo permette di penalizzare sia traiettorie che impiegano troppo tempo sia quelle che richiedono un eccessivo sforzo di controllo.

Dato un tempo di arrivo *fissato* τ e due stati $\mathbf{x}_0, \mathbf{x}_1$, è possibile determinare la traiettoria ottima $\pi^* = (\mathbf{x}[], \mathbf{u}[], \tau)$ tale che $\mathbf{x}[0] = \mathbf{x}_0$, $\mathbf{x}[\tau] = \mathbf{x}_1$ e $\dot{\mathbf{x}}[t] = A\mathbf{x}[t] + B\mathbf{u}[t] + \mathbf{c}$ per ogni $0 \leq t \leq \tau$. Questo prende il nome di problema del controllo ottimo per uno stato e un tempo finale, ed è presentato formalmente in [9].

Sia $\bar{\mathbf{x}}[t]$ lo stato del sistema al tempo t in assenza di input di controllo; assumendo

$\bar{\mathbf{x}}[0] = \mathbf{x}_0$, tale stato è descritto dalla relazione $\dot{\bar{\mathbf{x}}}[t] = A\bar{\mathbf{x}}[t] + \mathbf{c}$. Sia inoltre $G\,[t]$ il Gramiano della controllabilità del sistema ottenuto dalla risoluzione dell'equazione di Lyapunov:

$$\dot{G} = AG[t] + G[t]A^T + BR^{-1}B^T, \qquad G[0] = 0.$$

Allora è possibile calcolare l'input di controllo come

$$\mathbf{u}[t] = R^{-1}B^T exp[A^T(\tau - t)]G[\tau]^{-1}(\mathbf{x}_1 - \bar{\mathbf{x}}[\tau]).$$

Per determinare il tempo di arrivo, il problema precedente viene esteso scegliendo *liberamente* τ: in questo caso viene risolto un problema di controllo ottimo per uno stato fissato e un tempo arbitrario. Sostituendo la legge di controllo determinata in precedenza nella funzione di costo e calcolando l'integrale, si ottiene

$$c[\tau] = \tau + (\mathbf{x}_1 - \bar{\mathbf{x}}[\tau])^T G[\tau]^{-1}(\mathbf{x}_1 - \bar{\mathbf{x}}[\tau]).$$

Il tempo di arrivo ottimo τ^* è il tempo per cui questa funzione assume valore minimo. Può essere facilmente determinato calcolando la derivata prima di $c[\tau]$ rispetto τ e ponendola uguale a zero.

Nota la durata ottima, è possibile ricavare la traiettoria ottima $\pi^* = (\mathbf{x}[], \mathbf{u}[], \tau^*)$ associata. Ricordando la legge di controllo definita in precedenza, è possibile riscriverla utilizzando

$$\mathbf{y}[t] = exp[A^T(\tau - t)]\mathbf{d}\,[t]\,,$$

dove $\mathbf{d}[t] = -G[\tau]^{-1}(\mathbf{x}_1 - \bar{\mathbf{x}}[\tau])$, ottenendo così$\mathbf{u}[t] = -R^{-1}B^T\mathbf{y}[t]$. Si osservi che $\mathbf{y}[t]$ è la soluzione dell'equazione differenziale

$$\mathbf{y}\,[t] = -A^T\mathbf{y}\,[t]\,, \qquad \mathbf{y}\,[\tau^*] = \mathbf{d}\,[\tau^*]\,.$$

Sostituendo $\mathbf{u}[t]$ nella dinamica del sistema, si ottiene $\dot{\mathbf{x}}[t] = A\mathbf{x}[t] - BR^{-1}B^T\mathbf{y}[t] + \mathbf{c}$ che, combinata con la precedente, determina

$$\begin{bmatrix} \dot{\mathbf{x}}[t] \\ \dot{\mathbf{y}}[t] \end{bmatrix} = \begin{bmatrix} A & -BRB^T \\ 0 & -A^T \end{bmatrix} \begin{bmatrix} \mathbf{x}\,[t] \\ \mathbf{y}\,[t] \end{bmatrix} + \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix}, \qquad \begin{bmatrix} \mathbf{x}\,[t] \\ \mathbf{y}\,[\tau^*] \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{d}\,[\tau^*] \end{bmatrix}.$$

Risolvendo l'equazione differenziale vengono determinati gli stati $\mathbf{x}[t]$ lungo la traiettoria e, sostituendo $\mathbf{y}[t]$ a $\mathbf{u}[t]$, i controlli. Questo descrive completamente la traiettoria $\pi^*\,[\mathbf{x}_0, \mathbf{x}_1]$ a cui è associato il tempo di arrivo ottimo τ^* e il costo ottimo $c^*\,[\mathbf{x}_0, \mathbf{x}_1]$.

1.2.4.1.1 Algoritmo di Calcolo della traiettoria ottima Quanto descritto confluisce nell'algoritmo di calcolo della traiettoria ottima tra due punti $\mathbf{x}_0 \in X_{free}$ e $\mathbf{x}_1 \in X_{free}$, riassunto nell'Algoritmo 5 e sintetizzato a partire dall'approccio definito in [13].

Inizialmente viene determinata la durata ottima della traiettoria utilizzando un metodo numerico per la risoluzione di equazioni differenziali come, ad esempio, quello di Eulero del primo ordine. Ad ogni iterazione viene calcolato il valore di $\bar{\mathbf{x}}[t]$ e il Gramiano di controllabilità $G\,[t]$. I valori ad ogni iterazione vengono usati per determinare esplicitamente il costo della legge di controllo e il valore di $\mathbf{d}[t]$, quest'ultimo utilizzato nella successiva procedura di calcolo della traiettoria ottima. Il ciclo viene ripetuto finché viene rispettata la condizione di terminazione: in questo caso si continua a iterare fintanto che il costo decresce.

Una volta che la durata è stata determinata si può procedere con il calcolo della traiettoria. Anche in questo caso, per risolvere le equazioni differenziali descritte in precedenza, ci si avvale di un metodo numerico. Sia dt il passo temporale utilizzato nel metodo e $n = \tau^*/dt$ il numero complessivo di iterazioni. Procedendo a ritroso, ovvero partendo dal punto di arrivo della traiettoria, si determinano numericamente $\mathbf{x}[t]$ e $\mathbf{y}[t]$, nonché $\mathbf{u}[t]$ utilizzando la formula esplicita. Questo permette di determinare completamente la traiettoria π^*.

L'algoritmo descritto non è però l'unica tecnica risolutiva al problema del controllo ottimo. Se la matrice $A \in \mathbb{R}^{n\times n}$ è **nilpotente**, ovvero esiste un $n \in \mathbb{N}^+$ per cui $A^n = 0$, allora $exp[At]$ ha come espressione in forma chiusa un polinomio di grado $(n-1)$ in t. Le equazioni differenziali di $\dot{G}$ e $\bar{\mathbf{x}}[t]$ possono quindi essere risolte in modo analitico e gli integrali calcolati esplicitamente. Questo permette di esprimere il costo $c[\tau]$ in forma chiusa come un polinomio di grado $2n^2$ in τ e determinare il minimo della funzione mediante la derivata prima. Infine, dato che A è nilpotente, lo è anche la matrice

Algoritmo 5 Pseudo-codice del calcolo della traiettoria ottima

```
1  CalculateTrajectory(x_0, x_1, out c, out pi)
2      t = CalculateOptimalArrivalTime(x_0, x_1, out c, out d)
3      pi = CalculateOptimalTrajectory(x_0, x_1, t, d)
4  end
5
6  CalculateOptimalArrivalTime(x_0, x_1, out c*, out d)
7      t[0] = 0
8      i = 0
9      x_bar[0] = x_0
10     G[0] = 0
11     do
12         i++
13         t[i]=t[i-1]+dt
14         x_bar[i] = x_bar[i-1] + (A*x_bar[i-1]+c)*dt
15         G[i] = G[i-1] + (A*G[i-1]+G[i-1]*A^T+B*R^(-1)*B^T)*dt
16         if(det(G[i])>0)
17             c[i] = t[i] + (x_0-x[i])^T*G[i]*(x_0-x[i])
18             d = -G^(-1)*(x_0-x[i])
19     while(c[i] < c[i-1])
20     c* = c[i]
21     return t[i]
22 end
23
24 CalculateOptimalTrajectory(x_0, x_1, tau, d)
25     x[n] = x_1
26     y[n] = d
27     t = tau
28     do
29         u[i] = -R^(-1)*B^T * y[i]
30         x[i-1] = x[i] - (A*x[i]+B*u[i]+c)*dt
31         y[i-1] = y[i] - (-A^T*y[i])*dt
32         i=i-1
33         t = t - dt
34     while (t >= 0)
35     return pi = (x[], u[], tau)
36 end
```

$\begin{bmatrix} A & -BRB^T \\ 0 & -A^T \end{bmatrix}$ dell'equazione differenziale $\begin{bmatrix} \dot{\mathbf{x}}[t] \\ \dot{\mathbf{y}}[t] \end{bmatrix}$, che può essere calcolata in modo esplicito per determinare i valori di $\mathbf{x}[t]$, $\mathbf{y}[t]$ e $\mathbf{u}[t]$.

1.2.4.2 Sistemi con dinamica non lineare

L'algoritmo presentato richiede che la dinamica del sistema sia lineare. Tuttavia è possibile utilizzare questo approccio anche nei sistemi la cui dinamica è non-lineare attraverso la tecnica di linearizzazione. Sia $\mathbf{f}$ la funzione che definisce la dinamica del sistema:

$$\dot{\mathbf{x}}\,[t] = \mathbf{f}\,[\mathbf{x}[t], \mathbf{u}[t]]\,.$$

Si può approssimare localmente la dinamica attorno allo stato $\hat{\mathbf{x}}$ e al controllo $\hat{\mathbf{u}} = \mathbf{0}$, ottenendo

$$A = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\,[\hat{\mathbf{x}}, \mathbf{0}]\,,\ B = \frac{\partial \mathbf{f}}{\partial \mathbf{u}}\,[\hat{\mathbf{x}}, \mathbf{0}]\,,\ \mathbf{c} = \partial \mathbf{f}\,[\hat{\mathbf{x}}, \mathbf{0}] - A\hat{\mathbf{x}}.$$

L'Algoritmo 4 viene modificato di conseguenza: dato che il punto $\mathbf{x}_i$ è sia inizio sia fine di tutte le traiettorie calcolate all'i-esima iterazione, il sistema viene linearizzato ripetutamente attorno a $\hat{\mathbf{x}} = \mathbf{x}_i$. Nell'utilizzo di questa tecnica è però fondamentale sapere che la linearizzazione è valida solamente se la traiettoria non si allontana troppo dal punto $\hat{\mathbf{x}}$ scelto. Sarà cura del capitolo successivo analizzare nel dettaglio in che modo questa condizione può essere definita formalmente e rispettata. È comunque noto che durante l'esecuzione le distanze tra i punti si riducono e le traiettorie diventano sempre migliori, riducendo gli errori introdotti dall'approssimazione. Tuttavia non è ancora stata data alcuna prova formale della convergenza dell'algoritmo per sistemi la cui dinamica non è lineare.

Capitolo 2

Sviluppo di algoritmi basati su RRT

Nel Capitolo 1 sono stati introdotti i problemi di motion planning e kinodynamic planning, nonché le principali tecniche per la loro risoluzione. Tali approcci godo di interessanti proprietà: la completezza e l'ottimalità asintotica, insieme alla possibilità di determinare una traiettoria ottima, fanno sì che quelli sampling-based siano tra i principali algoritmi utilizzati. Nonostante queste proprietà, la lentezza della convergenza ad una soluzione, unita al mancato utilizzo delle informazioni relative al goal, suggeriscono la possibilità di nuovi sviluppi per queste tecniche.

In questo capitolo verranno presentati degli algoritmi con l'obiettivo di superare le limitazioni di quelli noti allo stato dell'arte. Per iniziare verrà approfondita la tecnica di linearizzazione, in particolare la scelta del punto di riferimento da utilizzare. Il contributo algoritmico verrà esposto nella seconda parte, introducendo una nuova tecnica basata su espansione dell'albero a partire da un nodo. Nella terza parte verrà descritta una semplice modifica a Kinodynamic RRT* con l'intento di produrre una versione in grado di restituire una soluzione ammissibile. Infine nell'ultima sezione verranno descritte alcune modifiche che possono essere incluse in tutte le versioni di RRT.

2.1 Studio dell'approssimazione lineare e generazione del punto campione

Il comportamento dinamico di un sistema non lineare nell'intorno di un punto di equilibrio e' ben descritto dal comportamento dinamico del sistema ottenuto calcolando l'approssi-

mazione lineare nell'intorno del punto stesso. La scelta di tale punto è fondamentale per la correttezza dell'algoritmo. In questa parte verranno indicati i vincoli che il punto utilizzato deve rispettare.

Sia $\dot{x} = f(x, u)$ l'equazione che descrive la dinamica del sistema, con $x \in \mathbb{R}^n$, $u \in \mathbb{R}^m$, $f(x, u) : \Omega \subseteq \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ e $f(x, u)$ differenziabile su Ω. Scelto il punto di linearizzazione x_0, u_0 e definito $x = x_0 + dx$ e $u = u_0 + du$ rispetto le variazioni dx e du, $\dot{x}$ può essere scritto come

$$\dot{x} = \dot{x_0} + A(x_0, u_0) = \dot{dx} = f(x_0 + dx, u_0 + du)$$

Ipotesi fondamentale nella scelta della coppia (x_0, u_0) è che $\dot{x_0} = f(x_0, u_0) = 0$, ovvero sia un punto di equilibrio per il sistema.

Mediante lo sviluppo in serie di Taylor valutato in (x_0, u_0) è possibile approssimare $f(x_0 + dx, u_0 + du)$ come

$$f(x_0 + dx, u_0 + du) \approx f(x_0, u_0) + \left.\frac{\partial f}{\partial x}\right|_{(x_0,u_0)} dx + \left.\frac{\partial f}{\partial u}\right|_{(x_0,u_0)} du$$

Sostituendo questa in $\dot{x}$ e ricordando che $\dot{x_0} = f(x_0, u_0) = 0$ *per definizione*, si ottiene

$$\dot{x} = \dot{x_0} + \dot{dx} = \dot{dx} \approx f(x_0, u_0) + \left.\frac{\partial f}{\partial x}\right|_{(x_0,u_0)} dx + \left.\frac{\partial f}{\partial u}\right|_{(x_0,u_0)} du$$

ovvero

$$\dot{x} = \dot{dx} \approx A(x_0, u_0)dx + B(x_0, u_0)du$$

dove $A(x_0, u_0) = \left.\frac{\partial f}{\partial x}\right|_{(x_0,u_0)}$ e $B(x_0, u_0) = \left.\frac{\partial f}{\partial u}\right|_{(x_0,u_0)}$.

Il modello esaminato è un rover a controllo differenziale la cui dinamica è descritta da $\dot{\mathbf{x}} = f(\mathbf{x}) + g(\mathbf{x})\mathbf{u}$, dove $\mathbf{x} = (x, y, \theta, v, k)^T$ e $\mathbf{u} = (u_v, u_k)^T$. In particolare

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \\ \dot{k} \end{bmatrix} = \begin{bmatrix} v \cdot cos(\theta) \\ v \cdot sin(\theta) \\ v \cdot k \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_v \\ u_k \end{bmatrix}$$

Gli input di controllo $\mathbf{u}$ consistono nella derivata della velocità e della curvatura, rispettivamente. Inoltre i vincoli fisici di controllo del sistema impongono che $\theta \in [-\pi, \pi]$, $v \in [0, 0.8]$ e $k \in [-0.25, 0.25]$.

La linearizzazione avviene rispetto un **punto di equilibrio** (x_0, u_0) del sistema dinamico, ovvero un punto in cui l'evoluzione del sistema $\dot{x} = f(x, u)$ è stazionaria. Formalmente questo è indicato come $f(x_0, u_0) = 0$. Questo implica che $\dot{x_0} = 0$, cioè il sistema rimane immutato alle condizioni descritte da (x_0, u_0).

Nel il modello esaminato è opportuno definire lo stato x scegliendo solo alcune componenti significative, ovvero θ, v, k. Definito $x = (\theta, v, k)^T$, la dinamica di interesse è descritta da

$$\begin{bmatrix} \dot{\theta} \\ \dot{v} \\ \dot{k} \end{bmatrix} = \begin{bmatrix} v \cdot k \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_v \\ u_k \end{bmatrix} = f(x, u) = n(x) + g \cdot u.$$

Ricordando che le variazioni rispetto il punto di equilibrio(x_0, u_0) sono

$$\theta = \theta_0 + d\theta,\ v = v_0 + dv,\ k = k_0 + dk,\ u_v = u_{v_0} + du_v,\ u_k = u_{k_0} + du_k,$$

lo stato non lineare del sistema può essere scritto come

$$\begin{bmatrix} \dot{\theta_0} \\ \dot{v_0} \\ \dot{k_0} \end{bmatrix} + \begin{bmatrix} \dot{d\theta} \\ \dot{dv} \\ \dot{dk} \end{bmatrix} = \begin{bmatrix} (v_0 + dv) \cdot (k_0 + dk) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_{v_0} + du_v \\ u_{k_0} + du_k \end{bmatrix} =$$
$$= \begin{bmatrix} v_0 \cdot k_0 + v_0 \cdot dk + k_0 \cdot dv + dv \cdot dk \\ u_{v_0} + du_v \\ u_{k_0} + du_k \end{bmatrix}$$

Volendo riportare questa nella forma linearizzata $\dot{x} = \dot{dx} \approx A(x_0, u_0)dx + B(x_0, u_0)du$, riscriveremo questa come

$$\begin{bmatrix} \dot{\theta_0} \\ \dot{v_0} \\ \dot{k_0} \end{bmatrix} + \begin{bmatrix} \dot{d\theta} \\ \dot{dv} \\ \dot{dk} \end{bmatrix} = \begin{bmatrix} v_0 \cdot k_0 \\ u_{v_0} \\ u_{k_0} \end{bmatrix} + \begin{bmatrix} 0 & k_0 & v_0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} d\theta \\ dv \\ dk \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} du_v \\ du_k \end{bmatrix}$$

così da indicare $A(x_0, u_0) = \begin{bmatrix} 0 & k_0 & v_0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = A(x_0)$ e $B(x_0, u_0) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = B$.

Ricordiamo che *per definizione* nel punto (x_0, u_0) scelto deve valere

$$\begin{bmatrix} \dot{\theta_0} \\ \dot{v_0} \\ \dot{k_0} \end{bmatrix} = \begin{bmatrix} v_0 \cdot k_0 \\ u_{v_0} \\ u_{k_0} \end{bmatrix} = 0$$

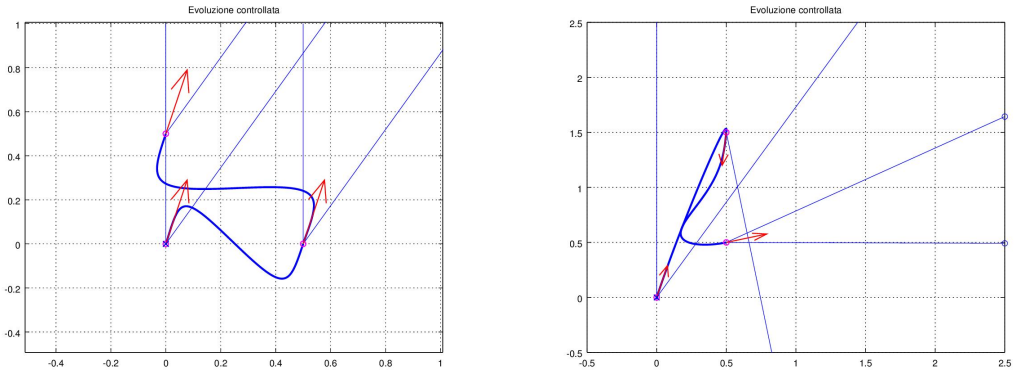
ovvero che le accelerazioni u_{v_0} e u_{k_0} devono essere nulle. Inoltre **anche k_0 deve essere nulla**: se così non fosse, si registrerebbe un'accelerazione centripeta.

Rivedendo la scelta del punto di linearizzazione nell'algoritmo Kinodynamic RRT* proposto in [13] risulta quindi chiaro che l'utilizzo del punto campione è **fondamentalmente errata**. Non vi è infatti alcuna garanzia che il punto campionato sia anche di equilibrio, ovvero il valore di k sia nullo. Dunque non si avrebbe più $\dot{x} = \dot{dx}$, ma $\dot{x} = \dot{x_0} + \dot{dx}$ ed proprio è questa la causa dal comportamento riscontrato:

> "[...] nel caso del [sistema non lineare] gli effetti della linearizzazione sono chiaramente visibili; il robot sembra slittare lateralmente per alcuni tratti, come se stesse percorrendo una curva."

Nel sistema trattato un altro elemento che può generare problemi durante il calcolo della traiettoria è l'orientazione. È intuitivo infatti che il modo in cui è orientato il robot discrimina fortemente le traiettorie che possono essere create a partire dallo stato iniziale: cambi di orientazione repentini impongono al robot movimenti bruschi, non sempre eseguibili dagli attuatori in dotazione.

Inoltre la linearizzazione richiede, come indicato in precedenza, che la traiettoria non si allontani troppo dal punto di equilibrio scelto. Per evitare le situazioni descritte in precedenza e per garantire la creazione di traiettorie ammissibili, occorre scegliere in mo-



(a) Punti fuori dal cono

(b) Orientazioni fuori dal limiti

Figura 2.1: Esempi di traiettorie errate

do opportuno il punto campione, imponendo dei vincoli sulla sua posizione spaziale e l'orientazione.

Consideriamo nuovamente il sistema non lineare in esame. Sia $\mathbf{x}_n$ lo stato associata all'n-esimo nodo dell'albero RRT e θ_n l'orientazione del robot nel suddetto stato. È possibile descrivere una regione di spazio $\mathcal{A}_{x_n}$ entro cui il punto finale $\mathbf{x}_1$ della traiettoria da calcolare deve trovarsi:

$$\mathcal{A}_{x_n} = \{(x_n + \rho \cdot cos(\theta), y_n + \rho \cdot sin(\theta)) : \rho \in [\rho_{min}, \rho_{max}], \theta \in [\theta_n - \Delta\theta, \theta_n + \Delta\theta]\}.$$

Nella definizione di $\mathcal{A}_{x_n}$ il parametro $\Delta\theta$ descrive l'angolo entro il quale è possibile linearizzare il sistema, mentre ρ_{min} e ρ_{max} identificano la distanza minima e massima, rispettivamente, entro la quale deve trovarsi il punto da raggiungere. Al fine di garantire una corretta linearizzazione è opportuno imporre $\mathbf{x}_1 \in \mathcal{A}_{x_n}$, così da evitare la creazione di traiettorie simili a quelle mostrate in Figura 2.1a.

Un approccio simile deve essere anche applicato anche all'orientazione del punto finale: connessioni incompatibili sono anche quelle che impongono al robot un cambio repentino di orientazione durante il tragitto, come mostrato in Figura 2.1b. Per escludere queste traiettorie è sufficiente richiedere che $\theta_1 \in [\theta_n - \Delta\theta, \theta_n + \Delta\theta]$, dove θ_1 è l'orientazione del robot nel punto finale della traiettoria.

Infine anche la velocità del punto campione deve essere limitata. Si consideri l'evolu-

zione del sistema nello spazio, descritta dalla seguente equazione:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} v \cdot cos(\theta) \\ v \cdot sin(\theta) \end{bmatrix}$$

Secondo la linearizzazione rispetto il punto di equilibrio, questa è approssimabile come

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} v \cdot cos(\theta) \\ v \cdot sin(\theta) \end{bmatrix} \approx \begin{bmatrix} cos(\theta_0) & -v_0 \cdot sin(\theta_0) \\ sin(\theta_0) & v_0 \cdot cos(\theta_0) \end{bmatrix} \begin{bmatrix} dv \\ d\theta \end{bmatrix}$$

Ricordando che nella computazione della traiettoria ottima sarà necessario calcolare il Gramiano della controllabilità del sistema e invertirlo, è necessario che il determinante della matrice $\begin{bmatrix} cos(\theta_0) & -v_0 \cdot sin(\theta_0) \\ sin(\theta_0) & v_0 \cdot cos(\theta_0) \end{bmatrix}$ sia diverso da 0:

$$det \begin{bmatrix} cos(\theta_0) & -v_0 \cdot sin(\theta_0) \\ sin(\theta_0) & v_0 \cdot cos(\theta_0) \end{bmatrix} = v_0 \neq 0$$

Inoltre la linearizzazione è valida solo se non si avventura troppo lontano da (x_0, u_0). Per questo motivo è necessario che v_1, ovvero la velocità nel punto finale della traiettoria, rimanga entro l'intervallo $[v_0 - \Delta v, v_0 + \Delta v]$.

Le considerazioni fatte definiscono in modo completo le caratteristiche di un punto per cui è possibile determinare la traiettoria di un sistema con dinamica non lineare e nonolonomico.

Ricordiamo che l'obiettivo della definizione della regione $\mathcal{A}_{x_n}$ è quello descrivere l'insieme di tutti i punti che generano traiettorie senza curve strette e cuspidi (non percorribili dal robot). Questa regione si traduce nella definizione di vincoli che i punti devono rispettare. Uniti alla condizione di linearizzazione per cui si richiede $k = 0$, un punto $\mathbf{x}_1$ è ammissibile per la traiettoria con $\mathbf{x}_0$ se e solo se

$$\mathbf{x}_1 \in \mathcal{A}_{x_0} \wedge v_1 \in [v_0 - \Delta v, v_0 + \Delta v] \wedge v_1 \neq 0 \wedge k_1 = 0$$

Quando viene scelto un punto campione $\mathbf{x}_{random}$ è quindi necessario verificare le tre condizioni: (1) deve esiste un $\mathbf{x}_i$ per cui vale $\mathbf{x}_{random} \in \mathcal{A}_{x_i}$, (2) inoltre il valore $v_{random} \in [v_0 - \Delta v, v_0 + \Delta v] \neq 0$, ed infine (3) $k_{random} = 0$. Se le condizioni sono rispettate il nodo può essere usato per ampliare l'albero.

2.1.1 Linearizzazione rispetto il punto di inizio traiettoria

La linearizzazione suggerita avviene nell'intorno del punto iniziale di ogni traiettoria, così da garantire una maggiore continuità nella traiettoria.

Si ricorda quindi che linearizzare nell'intorno di x_0 e u_0 equivale a scrivere $f(x,u)$ come

$$f(x,u) \approx f(x_0,u_0) + \frac{\partial f}{\partial x}(x_0,u_0)dx + \frac{\partial f}{\partial u}(x_0,u_0)du,$$

e lo stato del sistema come

$$\dot{x} = A(x_0)dx + Bdu.$$

Il punto finale della traiettoria dovrà essere governato dalla medesima dinamica linearizzata. Questo vuole dire che i valori di dv e $d\theta$ devono essere riscritti in accordo con il nuovo sistema. A tale fine è possibile calcolare la velocità e angolo non lineari come

$$\begin{array}{c} v = \sqrt{v_x^2 + v_y^2} \\ \theta = \arctan\left(\frac{v_y}{v_x}\right) \end{array}.$$

Dove v_x e v_y sono rispettivamente le velocità lungo l'asse x e l'asse y calcolate rispetto la dinamica del sistema linearizzata, ovvero

$$\begin{aligned} v_x &= -v_0 \cdot sin(\theta_0) \cdot (d\theta - \theta_0) + dv \cdot cos(\theta_0) = -a \cdot (d\theta - \theta_0) + dv \cdot cos(\theta_0) \\ v_y &= v_0 \cdot cos(\theta_0) \cdot (d\theta - \theta_0) + dv \cdot sin(\theta_0) = b \cdot (d\theta - \theta_0) + dv \cdot sin(\theta_0) \end{aligned}$$

dove $a = v_0 \cdot sin(\theta_0)$ e $b = v_0 \cdot cos(\theta_0)$.

Sostituendo v_x e v_y nella formulazione di θ, è possibile scrivere dv come funzione di θ

$$dv = \frac{-v_0\left[cos(\theta_0) + tg(\theta) \cdot sin(\theta_0)\right](d\theta - \theta_0)}{sin(\theta_0) \cdot tg(\theta) \cdot cos(\theta_0)} = -c \cdot (d\theta - \theta_0)$$

e sostituirla in v_x e v_y. Sostituendo quindi le due velocità in $v = \sqrt{v_x^2 + v_y^2}$ e raccogliendo $(d\theta - \theta_0) = \Delta\theta$ si ottiene l'equazione di secondo grado

$$\left((-a - c \cdot cos(\theta_0))^2 + (b - c \cdot sin(\theta_0))^2\right) \Delta\theta^2 - v^2 = 0$$
$$\left(\bar{a} + \bar{b}\right) \Delta\theta^2 - v^2 = 0$$

in cui l'incognita $\Delta\theta$ ha come soluzione $\Delta\theta = \pm\sqrt{\frac{v^2}{\bar{a}^2+\bar{b}^2}}$. Ricordando che $\Delta\theta = (d\theta - \theta_0)$, è possibile calcolare l'orientazione e la velocità nel punto finale della traiettoria, in accordo con la nuova dinamica linearizzata, come

$$d\theta_{1,2} = \pm\sqrt{\frac{v^2}{\bar{a}^2+\bar{b}^2}} + \theta_0$$
$$dv_{1,2} = \mp c \cdot \sqrt{\frac{v^2}{\bar{a}^2+\bar{b}^2}}$$

Infine, la scelta di quale coppia di soluzioni utilizzare per il calcolo della traiettoria deve essere basata sulla corrispondenza del segno tra il seno e il coseno di θ e quelli di $d\theta_{1,2}$.

2.2 Kinodynamic RRT basato su espansione

Come si evince dall'analisi precedente, determinare i limiti entro cui è accettabile linearizzare il sistema è un problema fondamentale. Garantire il rispetto della condizione dei linearizzazione determina la corretta riuscita del calcolo della traiettoria. Tuttavia è stato possibile sfruttare al meglio questi vincoli per realizzare un approccio innovativo per la risoluzione del problema di kinodynamic motion planning.

Nel seguito verranno descritte le tecniche di pianificazione basate su campionamento sviluppate. Seguendo l'approccio storicamente proposto, verrà prima presentato l'algoritmo in grado di restituire una soluzione ammissibile, quindi verrà raffinata la procedure per restituire la soluzione ottima.

2.2.1 Expansion KinoRRT

L'approccio suggerito fino ad ora prevede l'estrazione di un punto campione e i tentativi di connessione con tutti i nodi dell'albero, fino a quando non ne viene individuato uno rispettante i vincoli descritti in precedenza. Questa costante verifica risulta spesso in uno

Algoritmo 6 Algoritmo Kinodynamic RRT basato su espansione

```
1  ExpansionKinoRRT(X_free, x_init, x_goal, X_obs, n, c)
2      V <- {x_init}, E <- empty
3      for i=0 to n do
4          x_i <- ChooseNode(T=(V,E))
5          x_rand <- SampleFree_A(x_i)
6          opt_pi = CalculateOptimalTrajectory(x_i, x_rand)
7              if CollisionFree(opt_pi) then
8                  V.add(x_i)
9                  E.add(x_min, x_i)
10     return T=(V,E)
```

spreco di risorse computazionali per la generazione di punti non utilizzabili dall'algoritmo di calcolo della traiettoria.

Le condizioni poste sul punto campione possono essere usate per definire un metodo innovativo per la costruzione di RRT, modificando il campionamento utilizzato. Inizialmente viene scelto casualmente un nodo $\mathbf{x}_n$ dell'albero da espandere e viene determinata esplicitamente la regione di spazio $\mathcal{A}_{x_n}$. Le coordinate spaziali del nodo possono essere determinate campionando direttamente $\mathcal{A}$, piuttosto che $\mathcal{C}$, mentre l'orientazione del sistema nel punto può essere estratta entro l'intervallo $[\theta_n - \Delta\theta, \theta_n + \Delta\theta]$; infine k verrà imposto nullo e il valore della velocità scelto liberamente. Attraverso questo tipo di campionamento le condizioni per il calcolo della traiettoria sono rispettate. Il nodo può essere aggiunto all'albero, previa verifica di aver ottenuto una traiettoria collision-free. L'approccio basato su espansione appena descritto è riportato nell'Algoritmo 6 e prende il nome di *Expansion KinoRRT*.

Si noti che, applicando l'algoritmo al sistema non lineare in esame, durante il calcolo di ogni traiettoria dovrà essere attuata la linearizzazione rispetto $\mathbf{x}_0$ e $\mathbf{u}_0 = 0$, come descritta in precedenza.

La costruzione dell'albero scegliendo preventivamente il nodo da espandere presenta tuttavia una forte differenza rispetto i classici approcci basati su campionamento. È noto da [8] che l'algoritmo RRT indirizza l'esplorazione verso le regioni non ancora visitate dello spazio di ricerca. Scegliendo casualmente il nodo da espandere tra tutti quelli dell'albero, questa proprietà non è garantita per Expansion KinoRRT ed è quindi richiesto modificare in modo opportuno la scelta del nodo.

Come mostrato in Figura 2.2, è possibile associare ad ogni nodo dell'albero una regione

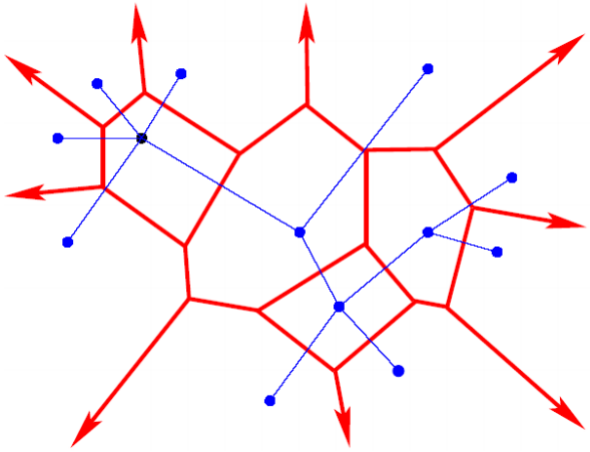


Figura 2.2: Il diagramma di Voronoi associato ad un RRT bidimensionale

Algoritmo 7 Scelta del nodo da espandere basato sulle dimensioni della regione di Voronoi

```
1 ChooseNode(T=(V,E))
2     x_rand <- SampleFree
3     x_expand <- Nearest(T =(V,E), x_rand)
4     return x_expand
```

di Voronoi. Ricordiamo che una partizione di Voronoi è definita su un insieme di punti. La regione R_k associata ad un punto p_k è l'insieme di tutti i punti la cui distanza da p_k è minore o uguale della distanza da tutti gli altri punti dell'insieme. La probabilità di scegliere un nodo associato ad una data regione è proporzionale al volume della regione stessa. Dato che i vertici di frontiera hanno associata una regione più grande, sarà più facile che vi venga collegato un nuovo nodo. Questo garantisce l'espansione di RRT verso le regioni di spazio inesplorate.

In modo analogo la scelta del nodo da espandere nell'algoritmo Expansion KinoRRT deve essere guidata dalla regione di Voronoi più grande. Tuttavia la computazione esplicita della regione non è ammissibile, né necessaria: scegliere il nodo associato alla regione più grande equivale a scegliere casualmente un punto dello spazio degli stati e scegliere come nodo da espandere il più vicino. Proprio come nel caso di RRT, questo garantisce che l'algoritmo esplori in modo uniforme lo spazio. Si osservi che questa tecnica, riassunta nell'Algoritmo 7, è facilmente implementabile con le primitive definite per gli algoritmi sampling-based.

Il principale vantaggio di Kinodynamic RRT basato su espansione è la riduzione del

Algoritmo 8 Algoritmo Kinodynamic RRT* basato su espansione

```
1  ExpansionKinoRRT*(X_free, x_init, x_goal, X_obs, n, c)
2      V <- {x_init}, E <- empty
3      for i=0 to n do
4          x_i <- ChooseNode(T=(V,E))
5          x_rand <- SampleFree_A(x_i)
6          opt_pi = CalculateOptimalTrajectory(x_i, x_rand)
7              if CollisionFree(opt_pi) then
8                  V.add(x_i)
9                  E.add(x_min,x_i)
10         // Aggiornamento dell'albero
11         X_A = Admissible(T=(V,E),x_i)
12         foreach x_a in X_A do
13             opt_pi = CalculateOptimalTrajectory(x_i, x_near)
14             if CollisionFree(opt_pi) && Cost(x_i)+c(opt_pi) < Cost(x_a)
                   then
15                 parent <- Parent(x)
16                 E.remove(parent,x)
17                 E.add(x_i,x_a)
18     return T=(V,E)
```

costo computazionale. Sia n il numero di nodi dell'albero e t il tempo (costante) per calcolare una traiettoria e determinare se è collision free. Ad ogni iterazione dell'algoritmo viene calcolato uno ed un solo percorso, che potrebbe o meno sovrapporsi a degli ostacoli. Nel caso migliore, se tutte le espansioni danno esito positivo (ovvero non sono presenti ostacoli), il costo computazionale è lineare rispetto il numero di nodi dell'albero, ovvero $t \cdot n$.

2.2.2 Optimal Expansion KinoRRT

A conclusione dell'approccio innovativo proposto viene introdotto anche l'algoritmo Expansion KinoRRT* che gode della proprietà di ottimalità asintotica. Questa versione è ottenuta associando ad ogni nodo il costo minimo della traiettoria e riscrivendo l'albero ogni volta che viene espanso.

Si assuma di avere a disposizioni la funzione $\mathrm{Cost} : V \to \mathbb{R}_{\geq 0}$, in grado di restituire il costo del percorso dalla radice dell'albero ad un nodo, e $\mathrm{Parent} : V \to V$, per determinare il predecessore di un nodo. Inoltre la funzione $Admissible(T = (V, E), x_i \in V)$ restituisce tutti i nodi dell'albero $T = (V, E)$ entro la regione $\mathcal{A}_{x_i}$.

Attraverso queste funzioni è possibile definire l'Algoritmo 8, che prende il nome di Expansion KinoRRT*. Ad ogni iterazione, dopo che il campione è stato aggiunto, vengono determinati i nodi $x_a \in \mathcal{A}_{x_{new}}$ entro la regione e, per ognuno di questi, viene calcolata la traiettoria a partire da x_{new}. Se il costo $c^*[\mathbf{x}_{new}, \mathbf{x}_a]$ della traiettoria sommato al costo per raggiungere il nodo x_{new} è minore del costo $Cost(x_a)$ associato ad x_a, allora è stato determinato un nuovo cammino minimo e l'albero può essere aggiornato, riscrivendo il predecessore di x_a.

Anche in questo caso è possibile valutare il costo computazionale, così da avere a disposizione un criterio di confronto con la versione non ottima ed altri algoritmi di pianificazione. Assumiamo di valutare il costo in un ambiente privo di ostacoli, costruendo un albero con n nodi e impiegando t per calcolare una traiettoria e determinare se è collision free. Inoltre, indichiamo con $|\mathcal{A}_i|$ la cardinalità dell'insieme $\mathcal{A}$ determinato durante l'espansione all'iterazione $\boldsymbol{i}$. Il costo computazionale dipende dalla cardinalità dell'insieme $\mathcal{A}$ di ogni iterazione:

$$t + (1 + |\mathcal{A}_2|)t + (1 + |\mathcal{A}_3|)t + ... + (1 + |\mathcal{A}_n|)t = t \sum_{i=1}^{n} (1 + |\mathcal{A}_i|) = n \cdot t + t \cdot \sum_{i=1}^{n} |\mathcal{A}_i|$$

Il caso peggiore è quello in cui $|\mathcal{A}_i| = |V|$, ovvero la cardinalità di $\mathcal{A}$ cresce con il crescere dell'albero. In questo caso particolare il costo risulta essere

$$t + (1 + 1)t + (1 + 2)t + (1 + 3)t... + (1 + n|)t = t \sum_{i=1}^{n} (1 + i) = n \cdot t + t \cdot \frac{n^2 + n}{2}$$

Nel caso peggiore quindi l'aggiornamento dell'albero può avere un costo quadratico.

2.3 Revisione di Kinodynamic RRT*

L'algoritmo Kinodynamic RRT* è uno dei principali elementi di interesse per la risoluzione del problema di kinodynamic motion planning. Per questo la sua analisi formale ha un forte rilievo, sia per gli aspetti pratici, sia per quelli teorici. Inoltre alcuni concetti propri di RRT e RRT* possono essere facilmente applicati anche nella versione kinodynamic.

In questa sezione verrà rivisto l'algoritmo Kinodynamic RRT*, presentando il costo computazionale e la versione in grado di restituire una soluzione ammissibile ma non necessariamente ottima. Infine, questa versione verrà ulteriormente raffinata, descrivendo la possibilità di utilizzare un numero ristretto di nodi durante la creazione dell'albero.

2.3.1 Costo computazionale di Kinodynamic RRT*

In [13] gli autori di Kinodynamic RRT* non hanno fornito alcuna indicazione relativa al costo computazionale dell'algoritmo, sebbene questo sia facilmente determinabile. Sia quindi n il numero di nodi di cui deve essere composto l'albero e t il tempo richiesto per calcolare una traiettoria e determinare se è collision free. Consideriamo il caso migliore in cui non sono presenti ostacoli, e quindi ogni campionamento produce una traiettoria che estende l'albero. Il costo computazione di Kinodynamic RRT* sotto queste ipotesi allora è

$$(t+t)+(2t+2t)+(3t+3t)+...+(nt+nt) = \sum_{i=1}^{n} 2 \cdot i \cdot t = 2t \sum_{i=1}^{n} i = t \cdot (n^2+n).$$

Il costo quadratico rispetto il numero di nodi rende l'algoritmo difficilmente utilizzabile in contesti pratici in cui è richiesta una soluzione in tempo breve. Inoltre non è da dimenticare che alcuni dei nodi campione potrebbero essere scartati durante la verifica dei vincoli di linearizzazione.

2.3.2 Kinodynamic RRT

Non in tutti i contesti applicativi si ha la necessità di avere una soluzione ottima per la risoluzione di un problema di pianificazione. A volte lo sforzo e le risorse richieste per mettere in movimento il robot è talmente basso da non giustificare l'impiego di un criterio di ottimalità nella definizione del problema. Altre volte invece il tempo a disposizione per la risoluzione non permette l'utilizzo di tecniche ottime. Per questi e altri motivi gli algoritmi basati su criterio di ammissibilità non sono solo una versione minore di quelli ottimi, ma delle vere e proprie possibilità e punti di sviluppo per la ricerca.

Algoritmo 9 Algoritmo Kinodynamic RRT

```
1  Kinodynamic-RRT(X_free, x_init, x_goal, X_obs, n, c)
2      V <- {x_init}, E <- empty
3      for i=0 to n do
4          x_i <- SampleFree_i
5          foreach x in V do
6              opt_pi = CalculateOptimalTrajectory(x, x_i)
7              if c(opt_pi)<r && CollisionFree(opt_pi) && Cost(x)+c(opt_pi
                   ) < c_min then
8                  x_min = x
9                  c_min = Cost(x)+c(opt_pi)
10         V.add(x_i)
11         E.add(x_min, x_i)
12     return T=(V,E)
```

La definizione di Kinodynamic RRT nasce con un processo inverso di quello che è il passaggio da RRT a RRT*: se la versione ottima viene prodotta includendo una procedura di aggiornamento dell'albero, la semplificazione di Kinodynamic RRT* si ha rimuovendo proprio questa procedura. La modifica è visibile nell'Algoritmo 9: dell'approccio originale viene preservata la scansione di tutti i nodi dell'albero per determinare la traiettoria a costo minore.

È inoltre interessante valutare il costo di Kinodynamic RRT, così da poterlo confrontare con la sua versione ottima. Si consideri quindi la costruzione di un albero di n nodi e un tempo t per determinare la traiettoria per connettere due nodi e verificare se è collisione. Allora il costo computazionale, in assenza di ostacoli, è pari a

$$t + 2t + 3t + ... + nt = \sum_{i=1}^{n} i \cdot t = t \sum_{i=1}^{n} i = t \cdot \frac{n^2 + n}{2}$$

L'esclusione dell'aggiornamento dell'albero suggerisce quindi un miglioramento teorico: sebbene i due costi computazionali siano entrambi quadratici, il vantaggio principale che si ottiene utilizzando la versione non ottima è un dimezzamento del tempo di esecuzione.

2.3.3 Variante k-Neighbours

Il problema principale insito in Kinodynamic RRT* e nella sua versione non ottima è la scansione totale di tutti i nodi che compongono l'albero. Questa problematica sebbene

Algoritmo 10 Algoritmo k-Kinodynamic RRT

```
1  Kinodynamic-RRT(X_free, x_init, x_goal, X_obs, n, c)
2      V <- {x_init}, E <- empty
3      for i=0 to n do
4          x_i <- SampleFree_i
5          X_Near <- Near(G,x_new,min(r,s))
6          foreach x in X_Near do
7              opt_pi = CalculateOptimalTrajectory(x, x_i)
8              if c(opt_pi)<r && CollisionFree(opt_pi) && Cost(x)+c(opt_pi
                   ) < c_min then
9                  x_min = x
10                 c_min = Cost(x)+c(opt_pi)
11         V.add(x_i)
12         E.add(x_min,x_i)
13     return T=(V,E)
```

sia stata trattata in [13] non ha trovato una risoluzione esaustiva: l'idea proposta dagli autori dell'algoritmo è quella di definire una metrica non Euclidea **basata sul costo della traiettoria** per escludere i nodi che non possono contribuire all'espansione dell'albero. L'approccio è formalmente corretto, ma non è certamente mirato a ridurre il costo computazionale dell'algoritmo: è intuitivo che la metrica proposta richiede inevitabilmente di calcolare tutte le traiettorie.

Quella proposta è invece un'*euristica*, simile a quanto indicato in [5] per RRT*. Utilizzando la funzione $Near\,(G = (V, E), x, r)$ vengono selezionati a priori tutti i nodi entro un raggio r da x, utilizzando come metrica la distanza Euclidea.

Come già mostrato in [5], il valore di r può essere scelto come costante o come funzione della cardinalità dell'insieme V. In entrambi i casi, il raggio è un parametro euristico del problema. Per questo motivo la versione proposta limita i tentativi di connessione ai soli k vicini, da cui il nome *k-Kinodynamic RRT* (Algoritmo 10).

In questo algoritmo la scelta del numero di vicini da considerare ricopre un ruolo chiave. Considerare pochi vicini permette di ridurre il tempo di calcolo, ma può diminuire la qualità della soluzione trovata. Viceversa, considerarne un numero elevato fornisce una più ampia scelta a scapito della velocità di computazione. In questo senso evidenziamo tre interessanti possibilità:

- $k = 1$: in questo caso viene considerato solo il nodo più vicino al campione. Verrà calcolata la traiettoria e verificata essere collision-free; qualora non lo sia, non ver-

ranno provati altri nodi, ma verrà estratto un nuovo campione. Si noti che questa versione è quella quanto più semplice possibile e, per l'approccio, molto simile alla versione di RRT proposta in [6].

- k variabile: in questo caso il numero di vicini da considerare viene scelto come funzione del numero di elementi dell'albero o del numero totale di campioni. La funzione proposta è proprio $k(card(V)) := k_{RRT} log(card(V))$, con $k_{RRT} = 2e$. Il principale punto di forza di questa scelta è la sua invarianza rispetto la dimensione dello spazio degli stati. Nulla vieta però di scegliere una funzione diversa, in grado di modellare meglio il problema in esame.

- $k = n$: utilizzare il numero complessivo di nodi non produce alcun miglioramento nei tempi di esecuzione, ma fornisce la visione completa dello spazio esplorato. Inoltre è interessante considerare come utilizzare questo valore di k sia equivalente e non effettuare nessuna selezione a priori sui nodi e quindi ad utilizzare Kinodynamic RRT.

Il costo computazione permette di descrivere in modo riassuntivo uno dei due effetti dell'introduzione della selezione a priori dei nodi, ovvero il tempo di esecuzione ipotetico. L'ottimalità della soluzione invece può essere provata solo attraverso prove sperimentali. Sia ancora una volta n il numero di nodi dell'albero e t il tempo per calcolare una traiettoria e determinare se è collision free. Il costo computazione di k-Kinodynamic RRT, assumendo un k costante e ipotizzando l'assenza di ostacoli, allora è

$$t + 2t + ... + kt + kt + ... + kt = \sum_{i=1}^{k} t \cdot i + \sum_{i=k+1}^{n} t \cdot k = \left[\frac{(k-1)\,k}{2} + (n-k)\,k \right] t.$$

La prima sommatoria fa riferimento alle iterazioni in cui il numero di nodi dell'albero è inferiore a k, la seconda a quelle in cui è superiore e quindi k limita il numero di nodi considerati. Nel caso in cui venga utilizzato $k(card(V)) := 2e \cdot log(card(V))$ il costo computazionale risulta invece

$$t\,(2e \cdot log(1)) + t\,(2e \cdot log(2)) + ... + t\,(2e \cdot log(n)) = t \cdot 2e \cdot \sum_{i=1}^{n} log\,(i) = \Theta t(n \cdot log(n))$$

È però fondamentale ricordare che questo approccio è basato sull'ipotesi che il tempo di calcolo della distanza Euclidea tra due punti sia minore del tempo di calcolo della traiettoria.

2.4 Ulteriori miglioramenti agli algoritmi

La definizione del problema di linearizzazione ha permesso di definire un algoritmo completamente nuovo. Sono ancora presenti delle problematiche, comuni ai diversi algoritmi, che possono essere risolte in modo indipendente dall'algoritmo esaminato.

In questa parte verranno presentate brevemente le problematiche, suggerendo un approccio che può essere seguito per la loro risoluzione.

2.4.1 Terminazione preventiva

Come in tutti gli algoritmi iterativi, la condizione di terminazione di RRT e delle sue versioni alternative svolge un ruolo molto importante per la costruzione dell'albero. Due sono le principali modalità per cui è possibile terminare l'algoritmo:

- Terminazione per numero di nodi: in questo caso la costruzione di RRT termina non appena vengono aggiunti n nodi all'albero. Un valore di n molto grande permette di coprire una regione maggiore dello spazio di ricerca, ma implica un maggiore tempo di esecuzione, in accordo con i costi computazionali riportati in precedenza.

- Terminazione per numero di cicli: in questo caso la costruzione termina appena vengono eseguiti n cicli, senza fare distinzione tra quelli che hanno permesso di espandere l'albero e quelli che hanno prodotto percorsi in collisione con ostacoli. In generale infatti non tutti i campionamenti espandono l'albero: se un percorso non è collision-free, il campione che lo ha generato viene scartato.

Queste due condizioni presentano però un problema concettuale quando utilizzate in uno degli algoritmi che forniscono come risultato unicamente una *soluzione ammissibile*. Si supponga, ad esempio, che il pianificatore riesca a raggiungere il goal dopo k iterazioni: per come sono stati definiti gli algoritmi, le restanti $(n - k)$ iterazioni non possono modificare la qualità della soluzione, poiché non aggiornano i percorsi dalla sorgente, e quindi non sono utili al fine di risolvere il problema.

Algoritmo 11 Condizione di terminazione basata sul raggiungimento del goal

```
1  SamplingBasedPlanner(X_free, x_init, x_goal, X_obs, n, c)
2      V <- {x_init}, E <- empty
3      do
4          x_new = ExpandTree(V,E)
5          if x_new != nil then
6              reached = ConnectViaCollisionFree(x_new,x_goal)
7      while(!reached)
8      V.add(x_goal)
9      E.add(x_new,x_goal)
10     return T=(V,E)
```

Questa osservazione suggerisce la possibilità di utilizzare una nuova condizione di terminazione non basata sul numero di nodi, ma piuttosto sul raggiungimento del goal. L'idea è presentata nell'Algoritmo 11 e consiste nel provare a connettere il goal ad ogni iterazione. Se l'espansione dell'albero ha avuto successo, ovvero è stato aggiunto un nuovo nodo x_{new} all'insieme V, viene verificato se è possibile connettere x_{goal} e x_{new} attraverso un percorso collision free. Se il riscontro è positivo, l'algoritmo può terminare e x_{goal} può essere aggiunto all'albero.

Si osservi che l'Algoritmo 11 è stato mantenuto volutamente generico, con l'intenzione di sottolineare la possibilità di utilizzare questa condizione in uno qualunque delle versioni definite in precedenza. Inoltre è importante evidenziare le diverse condizioni di terminazione possono essere combinate al fine di garantire una maggiore stabilità e robustezza del metodo di risoluzione.

2.4.2 Campionamento indirizzato al goal

Uno dei principali problemi degli algoritmi RRT risiede nell'esplorazione uniforme dello spazio. Infatti se da un lato questo garantisce la proprietà di completezza probabilistica, dall'altro fa si che la convergenza alla soluzione sia *lenta*. Il problema risiede nell'assenza di un *bias* verso la regione di interesse, che può essere tuttavia introdotto sfruttando informazione data dal goal, se presente.

Questa parte integra i concetti relativi al campionamento indirizzato al goal già accennati nel capitolo 1. Inizialmente verrà rivista la tecnica proposta, introducendo i vantaggi e gli svantaggi di questo approccio. Quindi verranno presentate nuove possibilità per l'utilizzo del goal come bias.

Indirizzare l'espansione verso il goal equivale a modificare la distribuzione di probabilità, dando un peso maggiore al nodo goal. Tipicamente, prima di campionare lo spazio, vi è una probabilità p di restituire il goal e $(1 - p)$ di estrarre uno qualunque dei punti dello spazio. Ad esempio, se $p = 1/2$ questo equivale a lanciare una moneta e restituire il goal se esce "testa" o un punto casuale nel caso in cui esca "croce".

In base a quale sia l'algoritmo utilizzato il bias può essere applicato in modo diverso:

- per RRT e Kinodynamic RRT il bias modifica il campionamento dello spazio, come descritto nell'esempio precedente;
- per Expansion KinoRRT il bias altera la scelta del nodo da espandere modificando l'estrazione del campione nell'Algoritmo 7. Con la stessa idea precedente, è possibile scegliere il goal o un punto campione; il nodo dell'albero da espandere sarà quello più vicino al punto scelto.

Attraverso un campionamento guidato verso il goal è possibile migliorare le performance degli algoritmi. Sfortunatamente utilizzando un approccio di questo tipo si incorre in quelli che sono i problemi tipici dei problemi di ricerca locale: se la costruzione dell'albero è troppo influenzata dal bias l'esplorazione dello spazio potrebbe rimanere intrappolata in un minimo locale, come un ostacolo vicino al goal. Fortunatamente, dato che l'algoritmo continua ad esplorare le regioni in modo casuale, è ancora garantita la proprietà di completezza probabilistica e, con un numero sufficiente di campioni, verrà determinato un percorso in grado di connettere il goal. Quanto si osserva è un aumento del tempo di esecuzione, dovuto alle iterazioni "sprecate" in cui il bias ha portato l'espansione verso un minimo locale.

È possibile scegliere altre strategia di bias, con l'obiettivo di ridurre la possibilità che l'algoritmo rimanga bloccato in un minimo locale.

La prima, e più semplice, consiste nell'utilizzare il goal con un periodo T fissato: una volta ogni T campionamenti effettuati viene restituito il goal, anziché un punto casuale. Maggiore è T, minore è il bias verso il goal e la possibilità che l'algoritmo rimanga intrappolato in un minimo locale, ma più lenta è la convergenza verso la soluzione.

Un'altra strategia prevede di utilizzare una regione favorita in cui indirizzare l'espansione. Anziché restituire il goal, viene estratto un punto a caso da una sfera di raggio s_{goal} centrata sul goal. Se la sfera ha un diametro ampio, la convergenza sarà più lenta, ma l'espansione verso ostacoli prossimi al goal sarà meno probabile.

Capitolo 3

Sviluppo del software

Nel Capitolo 2 sono stati presentati degli algoritmi che sfruttano un approccio innovativo alla risoluzione del problema di kinodynamic motion planning. L'algoritmo Expansion KinoRRT (e la corrispondente versione ottima) è stato ideato per risolvere il problema della linearizzazione intrinseco in Kinodynamic RRT*. L'algoritmo trova la sua principale applicazione come pianificatore per la navigazione di un robot governato da una dinamica non lineare.

La principale forza promotrice delle ricerche fatte nell'ambito della pianificazione mediante algoritmi sampling-based è il progetto SHERPA[1]. Nel contesto di questo progetto, uno degli obiettivi è quello di avere a disposizione un software in grado di pianificare una traiettoria per gli agenti autonomi robotici, nel rispetto della loro dinamica.

In questo capitolo verrà descritto lo sviluppo del software realizzato per il progetto SHERPA. L'obiettivo è quello di fornire un modulo software facilmente estendibile in grado di computare una traiettoria per un problema di kinodynamic motion planning. Ai fini di un'accurata analisi verranno inizialmente descritti i requisiti del problema. Nella seconda parte verrà proposta un'architettura logica di riferimento, descrivendola in un'ottica strutturale, di interazione e di comportamento. Infine nella terza e ultima parte verranno trattati i dettagli di progettazione, descrivendo gli algoritmi implementati.

[1] `www.sherpa-project.eu`

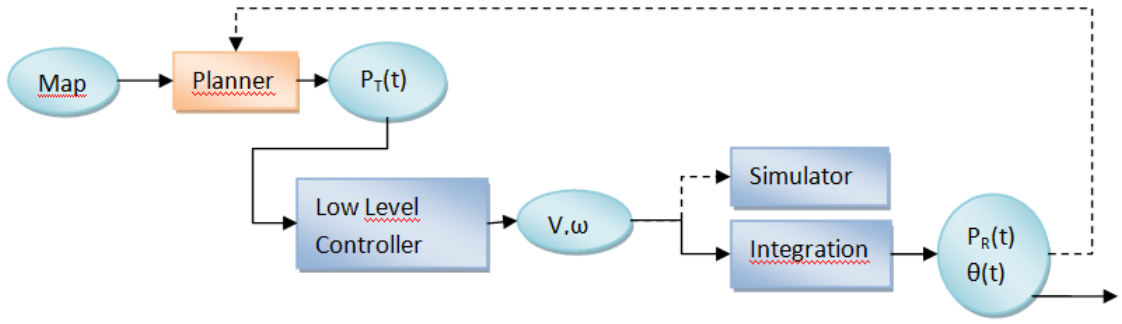


Figura 3.1: Astrazione del robot dal punto di vista logico

3.1 Requisiti del problema

Si desidera realizzare un modulo software che permetta ad un robot a controllo differenziale di muoversi in maniera autonoma da un punto prefissato A_1 ad un punto prefissato A_2, dato un insieme di nozioni relative all'ambiente. Il robot, durante la fase di pianificazione, deve tenere conto della presenza di ostacoli nell'ambiente e produrre un risultato libero da collisioni con gli ostacoli noti. Inoltre la traiettoria calcolata deve tenere conto e rispettare le leggi di dinamica del sistema. Non sono quindi ammesse traiettorie con cuspidi e spigoli evidenti. Infine, come vincolo tecnologico, si richiede che il software sia compatibile con il sistema operativo ROS (Linux); pertanto è suggerito l'utilizzo di un linguaggio di programmazione orientato ad oggetti e nativamente supportato, come C++ o Python.

3.1.1 Analisi dei requisiti e Modello del Dominio

Nel contesto della progettazione del software per il robot SHERPA, *l'agente* viene astratto come riportato in Figura 3.1. Il software a disposizione è già dotato di un controllore a basso livello in grado di trasformare la posizione del target da inseguire (P_T) in termini di velocità del corpo e velocità angolare (V, ω). Queste due informazioni verranno fornite ad un integratore in grado di tradurre i comandi ed eseguirli. Il risultato dell'integratore sono la posizione corrente del robot (P_R) e la sua orientazione (θ); eventualmente è possibile utilizzare queste informazioni in retroazione. Il sistema è anche dotato di un simulatore grafico.

Nei requisiti è richiesto in modo esplicito il rispetto della *dinamica* del sistema. Il modello del robot utilizzato è il medesimo del sistema esaminato nel capitolo 2.

Quanto descritto nei requisiti è un *pianificatore*, ovvero un sistema, umano o macchina, che semplicemente costruisce un piano. Se il pianificatore è una macchina, verrà considerato come un algoritmo di pianificazione. Diversa è l'accezione che viene data al termine pianificazione in base all'ambito in cui utilizzato: in generale, possiamo considerarlo come il processo che costruisce una sequenza di mosse che portano da uno stato iniziale ad uno stato desiderato. In questo caso i requisiti indicano espressamente la risoluzione di un problema di navigazione e pertanto gli algoritmi potranno essere studiati e realizzati con l'obiettivo di risolvere questa specifica classe di problemi di motion planning.

Il concetto di *stato* deve essere ulteriormente approfondito ed è relativo al contesto in esame. Lo stato infatti dipende dal problema, dal sistema, dai sensori, dagli attuatori e dalle informazioni manipolabili. Data la dinamica del sistema descritta, lo spazio degli stati $\mathbf{x} = (x, y, \theta, v, k)^T$ consiste nella posizione (x, y) sul piano, l'orientazione θ rispetto il sistema di riferimento inerziale, la velocità v (m/s) e la curvatura k (m^{-1}).

Lo stato manipolato dal pianificatore deve trovare una forma di corrispondenza con la *mappa* a disposizione del sistema. Una mappa è una griglia di occupazione (2D o 3D) rappresentante l'ambiente in termini di celle. Ogni cella ha associato un valore che indica se essa è occupata o no. Il valore permette di descrivere lo stato del robot quando si trova in una data posizione, utilizzando diversi range di valori. Per la realizzazione di questo software sarà presa in esame una mappa binaria: il valore 100 indica che il robot è certamente in collisione mentre un valore 0 indica che si trova in una posizione sicura.

Infine i termini traiettoria e percorso non sono sinonimi. In accordo con quanto descritto anche nel Capitolo 1, un *percorso* crea un ordinamento per una successione di punti dello spazio delle configurazioni; in altri termini, un percorso è un luogo geometrico di punti. Una *traiettoria* è invece prodotta da una legge oraria che esprime non solo la posizione nello spazio, ma anche la velocità e l'orientazione che il robot deve avere. Questo suggerisce che il percorso è un caso particolare della traiettoria in cui i valori di velocità e orientazione non sono specificati.

3.2 Architettura Logica

Una prima fase dello sviluppo del software è stata dedicata alla definizione dei confini del sistema e dell'architettura logica. Tre sono gli elementi fondamentali emersi dai requisiti:

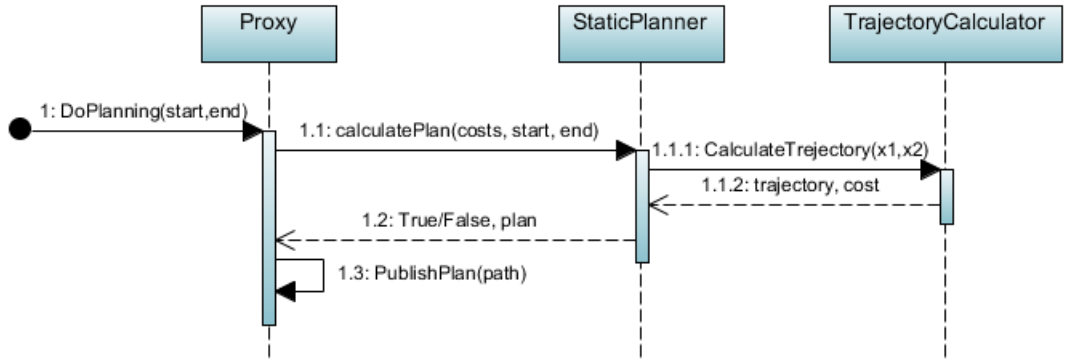


Figura 3.2: Diagramma di interazione relativo all'architettura logica

- **Proxy** - Lo scopo di questa classe è quello di ricevere e servire le richieste derivanti dall'esterno del sistema. Interagisce con il vero e proprio planner per la risoluzione del problema di pianificazione posto. Inoltre si occupa di convalidare le informazioni ricevute e recuperare le informazioni mancanti, come la mappa.

- **StaticPlanner** - È la classe provider dei servizi di pianificazione. Ricevute le informazioni relative al problema si limita a risolverlo utilizzando gli algoritmi a disposizione. I controlli sulla conformità degli input sono delegati alla classe client, sebbene siano possibili ulteriori verifiche.

- **TrajectoryCalculator** - Questa classe si occupa della risoluzione del problema di controllo, ovvero del calcolo della traiettoria.

Il legame tra i tre componenti descritti è evidenziato dal diagramma di interazione in Figura 3.2. Dato che il software dovrà essere compatibile con il sistema operativo ROS, il Proxy dovrà aderire al modello progettuale del framework. Dopo aver verificato la correttezza della richiesta, delegherà alla classe StaticPlanner la risoluzione del problema. Questa astrae tutti gli algoritmi di pianificazione, senza eccezione; sarà la classe Proxy a definire quale pianificatore istanziare e utilizzare.

In modo generale è corretto affermare che StaticPlanner non ha bisogno di utilizzare il modulo di calcolo della traiettoria. Infatti, da un punto di vista strettamente semantico, un pianificatore statico non risolve unicamente il problema di kinodynamic motion planning, ma più generalmente quello di pianificazione. Al fine di risolvere i requisiti posti, non è però possibile considerare il modo secondario TrajectoryCalculator. Questo spin-

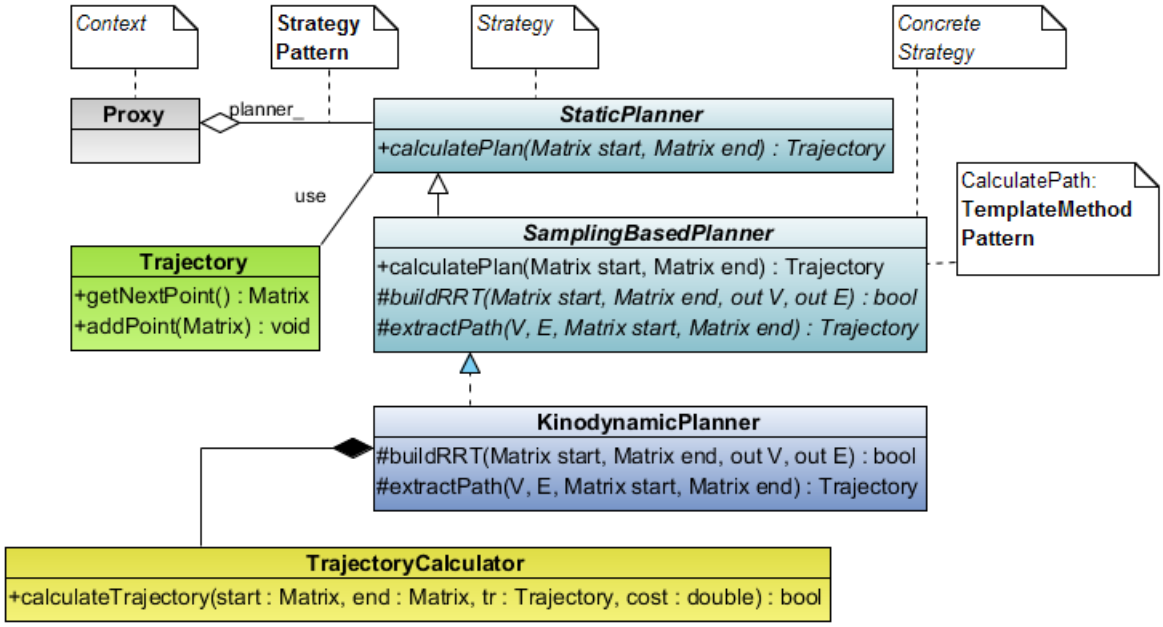


Figura 3.3: Diagramma strutturale relativo all'architettura logica

ge alla necessità di analizzare l'architettura logica secondo un punto di vista strutturale, considerando come parte integrande anche una classe per il calcolo della traiettoria.

Il diagramma delle classi in Figura 3.3 fornisce maggiori informazioni relative ai pattern utilizzati per dare corpo al sistema di pianificazione.

Il pattern **Strategy** permette di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Ciò permette agli algoritmi di variare indipendentemente dal client che li utilizza. Questo pattern favorisce l'estendibilità del codice, dando la possibilità di implementare nuovi algoritmi di pianificazione (Kinodynamic RRT, Expansion KinoRRT, e così via) senza richiedere modifiche strutturali all'architettura.

Il pattern **Template Method** completa Strategy, definendo lo scheletro di un algoritmo in termini di macro operazioni e lasciando la loro implementazione alle classi figlie. In questo modo è possibile definire il comportamento dei singoli algoritmi, senza replicare codice o cambiare la struttura. Ovviamente l'applicazione di questo pattern ha richiesto la definizione di una nuova classe, *SamplingBasedPlanner*, che rappresenti tutti gli algoritmi basati su campionamento.

Il diagramma strutturale indica anche la codifica utilizzata per lo stato del sistema, ovvero la classe *Matrix*. L'utilizzo delle matrici come entità del primo ordine per modellare

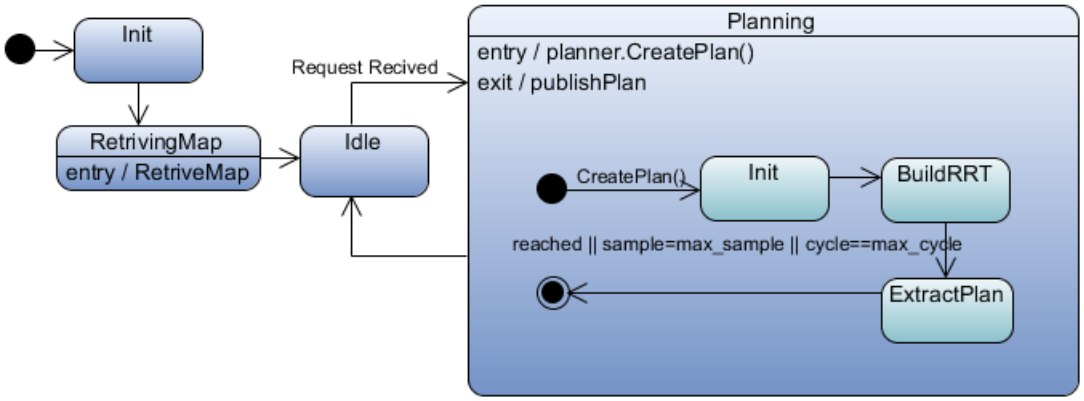


Figura 3.4: Diagramma di comportamento relativo all'architettura logica

gli elementi dello spazio degli stati è suggerito dal modello del dominio. Infatti tutti gli algoritmi fin qui presentati sono stati generalizzati per $\mathbb{R}^n$ dimensioni. Inoltre sono presenti concetti di algebra lineare necessari per la risoluzione del calcolo della traiettoria.

Infine *Trajectory* è una struttura dati sequenziale (unidirezionale o bidirezionale) il cui scopo è quello di memorizzare gli stati che compongono la traiettoria.

Il diagramma degli stati riportato in Figura 3.4 completa l'analisi dell'architettura logica. Con questo si vogliono sottolineare le fasi eseguite dagli algoritmi di pianificazione basati su campionamento, in particolare la divisione tra costruzione dell'albero ed estrazione del piano. Inoltre viene evidenziata la condizione di terminazione degli algoritmi: se durante la costruzione viene raggiunto il goal, o vengono aggiunti un numero sufficiente di nodi o sono completati un numero massimo di cicli, l'algoritmo di pianificazione termina. L'estrazione del piano può avere successo o meno, ma in entrambi i casi l'informazione viene fornita al client (ovvero la classe Proxy).

3.2.1 Robotic Operating System e Abstraction Gap

Le attività di SHERPA si concentrano sulla combinazione di agenti robotici eterogenei; layer software comune su cui questi si basano è il Robot Operating System. Questo framework ha come obiettivo quello di semplificare lo sviluppo delle applicazioni di robotica, fornendo semplici interfacce di accesso all'hardware e codice software riutilizzabile.

ROS viene definito[2] come un **meta-sistema operativo** open source per la realizzazione

[2]http://www.ros.org/about-ros/

di robot. Fornisce tutti i servizi attesi da un sistema operativo, incluse le astrazioni hardware, il controllo di device di basso livello, le funzionalità di utilizzo comune e la gestione dei package. Fornisce anche gli strumenti e le librerie per lo sviluppo software. Questa descrizione pone particolare attenzione al fatto che ROS non rimpiazza il sistema operativo, ma piuttosto si costituisce come uno strato che ne potenzia e amplia le funzionalità, come un framework.

L'obiettivo primario di ROS è quello di permettere agli sviluppatori di progettare e sviluppare il software come una collezione di elementi computazionali essenziali ed indipendenti, chiamati **nodi**. Un nodo è l'istanza in esecuzione di un programma ROS. I nodi sono entità computazionali indipendenti, ognuno con un proprio flusso di controllo, che comunicano attraverso un meccanismo Event-Driven. Un nodo che desidera comunicare pubblica (*publish*) dei messaggi su uno o più **topic**; nodi che desiderano conoscere i messaggi pubblicati si mettono in ascolto (*subscribe*) di uno o più topic. I **messaggi** sono i dati scambiati tra i nodi attraverso i topic, ognuno con un proprio tipo, definiti da file con estensione *.msg*.

Il tratto distintivo dei sistemi ROS è che i nodi sono debolmente accoppiati **(loosely coupled**): nessuno dei nodi conosce il riferimento agli altri nodi, né chi sarà in ascolto sul canale di comunicazione; il modo principale di interazione è questo meccanismo di comunicazione indiretta.

L'utilizzo di ROS, come richiesto nei requisiti, non implica nessun abstraction gap nella progettazione del software. Tuttavia deve essere predisposto un nodo da inserire nel sistema esistente. Questo ruolo può essere ricoperto in modo appropriato dalla classe Proxy: essa infatti riceve e serve le richieste dall'esterno del sistema, nascondendo il funzionamento interno del pianificatore. Aderendo alle specifiche di ROS, comunicherà con gli altri nodi attraverso il meccanismo di message passing, utilizzando messaggi di tipo RoverTrajectory.msg:

RoverTrajectory.msg

```
RoverPosition[] positions
```

Dove RoverPosition fa riferimento ad un tipo di dato specificato dal file RoverPosition.msg:

RoverPosition.msg

```
float32 x
float32 y
float32 velocity_x
float32 velocity_y
```

3.3 Progettazione

Durante la progettazione viene posto come scopo quello di raffinare l'architettura logica del sistema, considerando gli aspetti vincolanti che sono stati trascurati nelle fasi precedenti. In particolare verrà posta l'attenzione sugli algoritmi implementati, sia di pianificazione sia di calcolo della traiettoria e sulla modalità di inizializzazione del sistema.

3.3.1 Estensione degli algoritmi

L'architettura logica proposta agevola e favorisce la definizione di nuovi algoritmi di pianificazione. A titolo esplicativo e per la risoluzione del problema di navigazione, verranno introdotti quattro algoritmi: Kinodynamic RRT, Expansion KinoRRT e le corrispettive versioni ottime. Per una maggiore chiarezza questi verranno prima descritti secondo una visione strutturale, così da raffinare l'architettura proposta, e in un secondo momento ne verrà definito il comportamento.

3.3.1.1 Struttura delle classi

Per introdurre nuovi algoritmi ed estendere l'architettura è sufficiente creare una nuova classe che implementi la classe astratta SamplingBasedPlanner e definisca i metodi *buildRRT* ed *extractPath*. La classe SamplingBasedPlanner deve mettere a disposizione tutti i metodi propri degli algoritmi basati su campionamento. Per questo motivo, durante la progettazione, sono state introdotte le funzioni primitive descritte in [5]. Qualunque altra operazione fondamentale per la manipolazione degli alberi RRT dovrà, in futuro, essere inserita in questa classe.

Infine, in questa fase progettuale, occorre definire in modo formale le strutture dati da utilizzare, in particolare per quanto riguarda la memorizzazione degli alberi. Gli algoritmi descritti eseguono alcune operazioni:

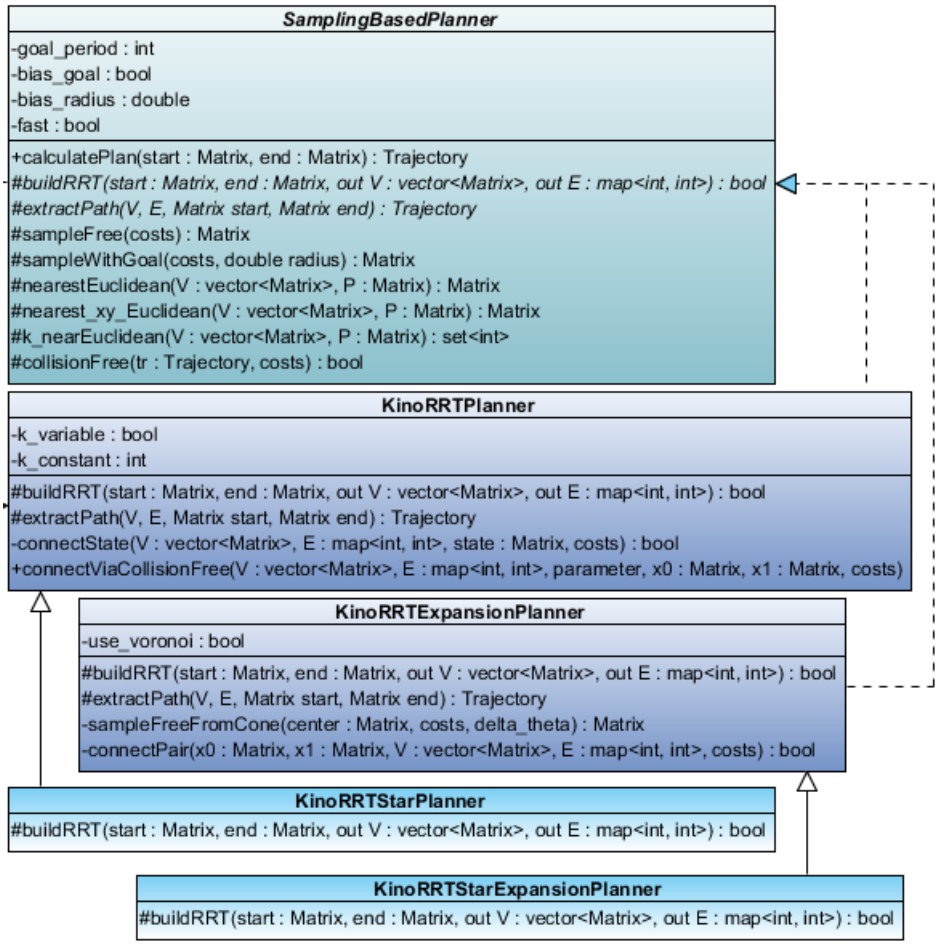


Figura 3.5: Progettazione strutturale dei nuovi algoritmi

- accesso diretto ai nodi dell'albero durante la fase di costruzione dell'albero,
- recupero del predecessore di un nodo durante l'estrazione della traiettoria finale,
- calcolo di metriche di vicinanza tra i nodi dell'albero.

Com'è noto esistono diverse possibilità in grado di gestire in modo efficiente un albero, a partire da una semplice lista fino ad arrivare alla definizione di un tipo di dato astratto. In questo caso però si è preferito favorire l'accesso diretto, memorizzando i nodi dell'albero con una lista indicizzata e gli archi con un hash map: in questo modo si favoriscono l'accesso diretto ai nodi e il recupero dei predecessori, accettando di impiegare più tempo nel calcolo dei vicini di un nodo.

La progettazione degli algoritmi di pianificazione viene riassunta in Figura 3.5: la classe SamplingBasedPlanner è stata raffinata con l'introduzione di nuovi metodi e nuovi attributi per gestire i diversi comportamenti; tra questi rientrano la possibilità di utilizzare il *bias* verso il goal e la terminazione appena il goal viene raggiunto.

Le due classi KinoRRTPlanner e KinoRRTExpansionPlanner rappresentano gli algoritmi Kinodynamic RRT ed Expansion KinoRRT, rispettivamente. Da queste sono definite le classi KinoRRTStarPlanner e KinoRRTStarExpansionPlanner in cui, attraverso la tecnica dell'override, viene ridefinito il metodo buildRRT. Grazie al pattern Template Method il metodo di costruzione dell'albero permetterà di costruire un RRT aggiornato in modo completamente trasparente al client del servizio.

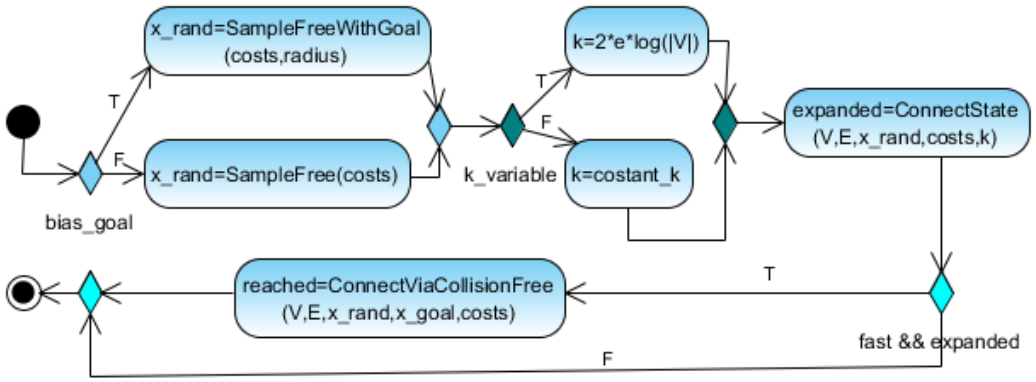
3.3.1.2 Comportamento delle classi

L'introduzione delle classi richiede la definizione degli algoritmi buildRRT ed extractPath; questi caratterizzano il comportamento del pianificatore e completano la loro progettazione. Al fine di definire in modo formale questi aspetti verranno utilizzati dei diagrammi di attività.

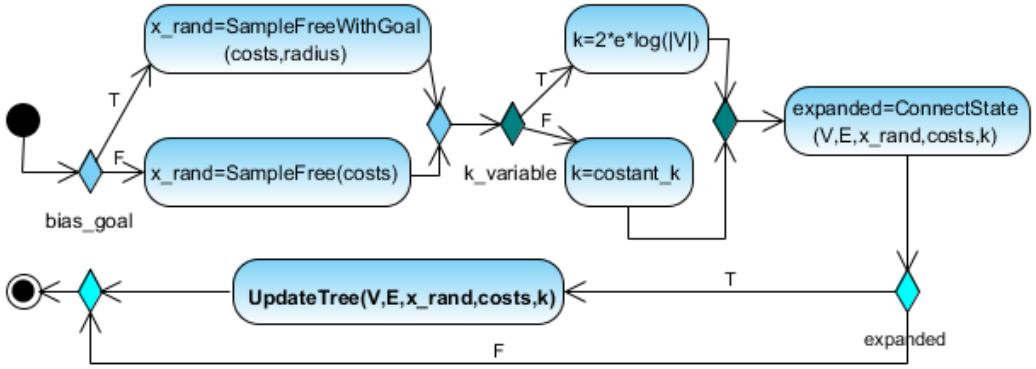
Il metodo buildRRT permette di costruire l'albero esplorativo dello spazio degli stati. La Figura 3.6a ne descrive il comportamento per la classe KinoRRTPlanner, indicando le attività eseguite *ad ogni iterazione*. Tre parametri regolano il flusso di operazioni: *bias_goal*, *k_variable* e *fast*. Il primo indica se l'espansione dell'albero deve essere indirizzata verso il goal, il secondo se si vuole considerare un numero di vicini variabile quando un nuovo stato viene connesso, il terzo se si desidera terminare preventivamente l'algoritmo appena il goal è raggiungibile.

In maniera simile opera la versione ottima, riportata in Figura 3.6b, in cui è stato introdotta la procedura di aggiornamento. Questa verrà invocata ogni volta che un nuovo è stato aggiunto all'albero e utilizzerà il valore k per determinare quanti vicini considerare durante l'aggiornamento. Si osservi che, in questo caso, non è presente la terminazione preventiva, nel rispetto della proprietà di ottimalità asintotica.

La descrizione riportata per buildRRT differisce, ovviamente, per la classe KinoRRTExpansionPlanner. In questo caso infatti non vengono individuati dei vicini con cui provare a connettere il nodo campionato. Al contrario, viene estratto casualmente un nodo da espandere e il campionamento avviene all'interno della regione stabilita. In questa situa-

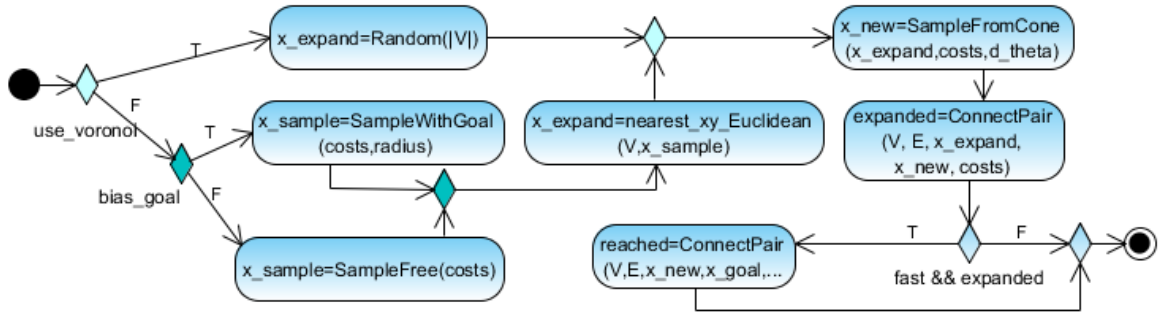


(a)

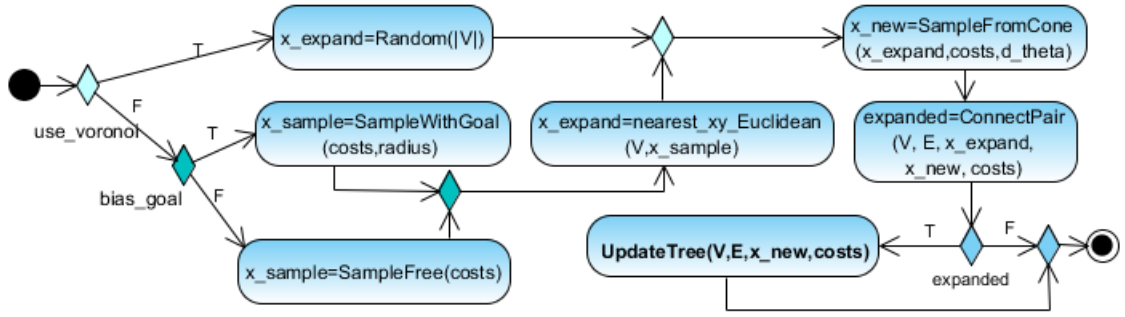


(b)

Figura 3.6: Comportamento della procedura buildRRT di Kinodynamic RRT e Kinodynamic RRT*



(a)



(b)

Figura 3.7: Comportamento della procedura buildRRT per le versioni basate su espansione

zione, oltre i parametri bias_goal e fast, è presente anche *use_voronoi* che permette all'algoritmo di esplorare le regioni non ancora visitate dello spazio. La Figura 3.7a riassume il comportamento dell'algoritmo ripetuto ad ogni iterazione di espansione dell'albero.

Nel caso di ExpansionKinoRRT*, il comportamento viene esteso eseguendo l'aggiornamento dell'albero (Figura 3.7b). Data la struttura di memorizzazione dei nodi che compongono RRT, per determinare i nodi che rientrano nell'area ammissibile di x_new occorre un'analisi sequenziale: per tutti quelli che rispettano i vincoli definiti nel Capitolo 2 sarà possibile determinare la traiettoria ed, eventualmente, aggiornare il percorso e il costo associato.

Contrariamente alle precedenti, la procedura di estrazione del percorso finale è identica per tutti gli algoritmi. In Figura 3.8 è riportato il flusso di attività del metodo *extractPath*. Si tratta di un algoritmo iterativo che, a partire dal goal, risale l'albero fino alla radice per determinare tutte le traiettorie.

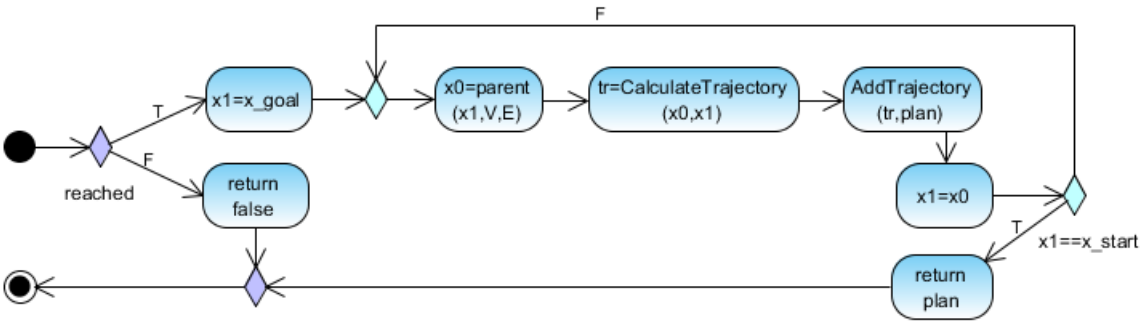


Figura 3.8: Comportamento della procedura extractPath

3.3.2 Algoritmi di calcolo della traiettoria

Tutti gli algoritmi introdotti fino ad ora fanno ampio uso del calcolo della traiettoria. Risulta quindi di fondamentale importanza progettare una classe che implementi l'algoritmo di calcolo della traiettoria ottima descritto in [13].

A tale scopo si definisce quindi una classe OptimalTrajectoryCalculator che eredita da quella astratta TrajectoryCalculator e implementa il metodo calculateTrajectory (Figura 3.9). Oltre questo, sono previste le procedure per la risoluzione del problema di controllo a tempo minimo (calculateOptimalTau) e di controllo a minima energia (calculateOptimalTrajectory).

L'approccio seguito è quello suggerito dal pattern Strategy, in cui la classe per il calcolo della traiettoria ottima implementa la specifica strategia di risoluzione del problema.

Il comportamento della classe OptimalTrajectoryCalculator è essenziale per la risoluzione del problema di controllo ottimo, ed è descritto in Figura 3.10. Inizialmente la classe si trova, all'invocazione del costruttore, in uno stato di inizializzazione durante il quale possono essere eseguite tutte le eventuali attività di pre-calcolo dei valori costanti. Lo stato di idle rappresenta l'attesa della richiesta di calcolo della traiettoria, effettuata attraverso la chiamata del metodo calculateTrajectory. A fronte di questo, la classe entra in uno stato di verifica dei dati inseriti: se risultano tutti conformi si può procedere all'elaborazione o, in caso contrario, informare del fallimento e ritornare allo stato di idle. Durante il calcolo della traiettoria il punto finale verrà convertito rispetto la dinamica non lineare, come indicato nel Capitolo 2, quindi verrà calcolata la traiettoria ottima, come riportato nel Capitolo 1.

Per i metodi *calculateOptimalTau* e *calculateOptimalTrajectory* è proposto l'utilizzo

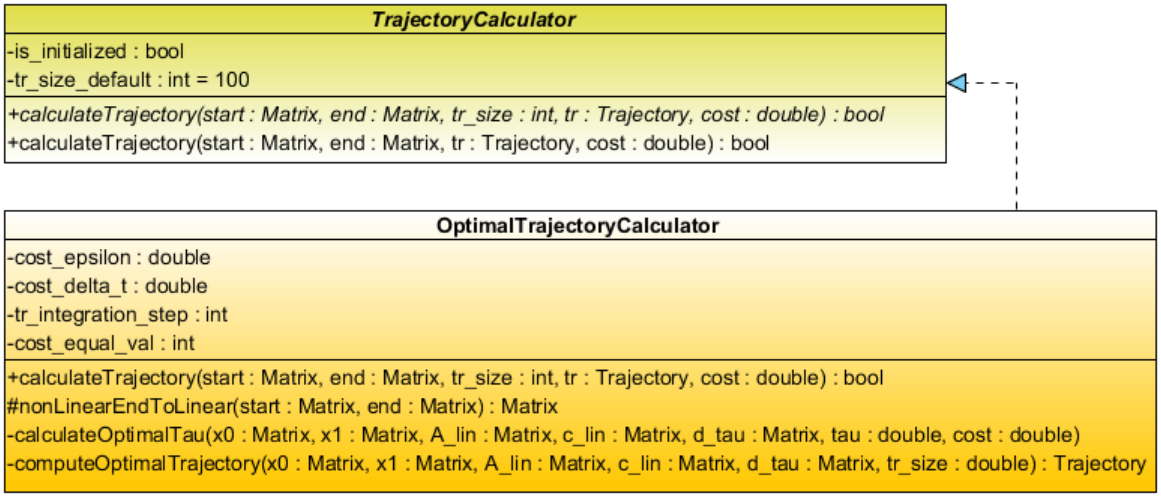


Figura 3.9: Diagramma delle classi degli algoritmi di calcolo della traiettoria

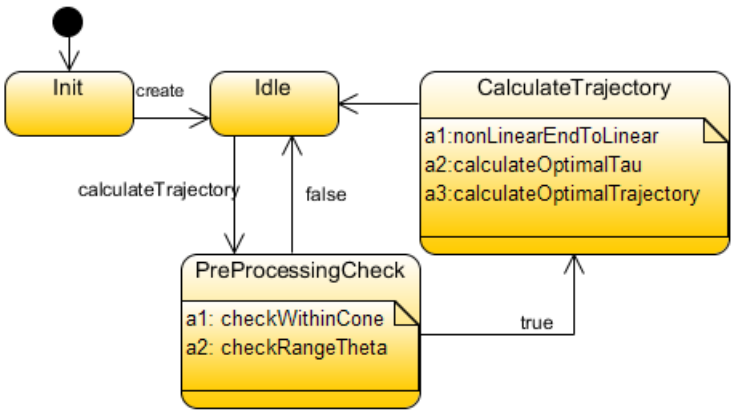


Figura 3.10: Diagramma degli stati della classe OptimalTrajectoryCalculator

del metodo di Eulero del primo ordine. È molto importante sottolineare che l'utilizzo di metodi numerici introduce un errore di calcolo. Dato che la traiettoria viene determinata con un'integrazione all'indietro, i valori dello stato iniziale per cui questa è calcolata potranno essere soggetti ad errori che dovranno essere gestiti a livello di algoritmo di pianificazione. Per una maggiore precisione, a scapito del tempo di esecuzione, può essere implementato il metodo di Runge-Kutta del quarto ordine. Attraverso il pattern Strategy, questa estensione risulta molto semplice.

Infine sono possibili alcune osservazioni a livello generale che permettono di delineare meglio i parametri che regolano gli algoritmi proposti:

- Nel calcolo del tempo di arrivo ottimo, l'obiettivo è quello di determinare il minimo di una funzione. Per questo motivo la condizione di terminazione del metodo di Eulero verifica se il costo ha subito variazioni rispetto l'iterazione precedente. Se non sono state registrate variazioni dopo un numero fissato di iterazioni (definito da *cost_equal_val*) il metodo può terminare.

- Per il calcolo del tempo di arrivo ottimo, viene scelto un incremento temporale pari a *cost_delta_t*.

- Per il calcolo della traiettoria ottima invece il metodo di Eulero viene iterato un numero di volte pari a *tr_integration_step*, utilizzando un delta temporale pari al rapporto tra τ^* e tr_integration_step.

3.3.3 Planner builder

A conclusione della progettazione del sistema si prevede la definizione di un insieme di classi per la creazione del pianificatore. La progettazione del costruttore ha come obiettivi il favorire la trasparenza nell'utilizzo degli algoritmi, promuovere la corretta costruzione del pianificatore e agevolare l'estensione del sistema con l'introduzione di nuove tecniche per la soluzione del problema di navigation.

Per inizializzare il sistema in modo appropriato occorre progettare l'insieme delle classi utilizzando il pattern **Abstract Factory**. Si assuma di avere una classe base e un numero non prefissato di classi derivate; il pattern Factory permette di creare un oggetto di un tipo derivato che dipende dai parametri o dai valori forniti in ingresso al metodo di costruzione.

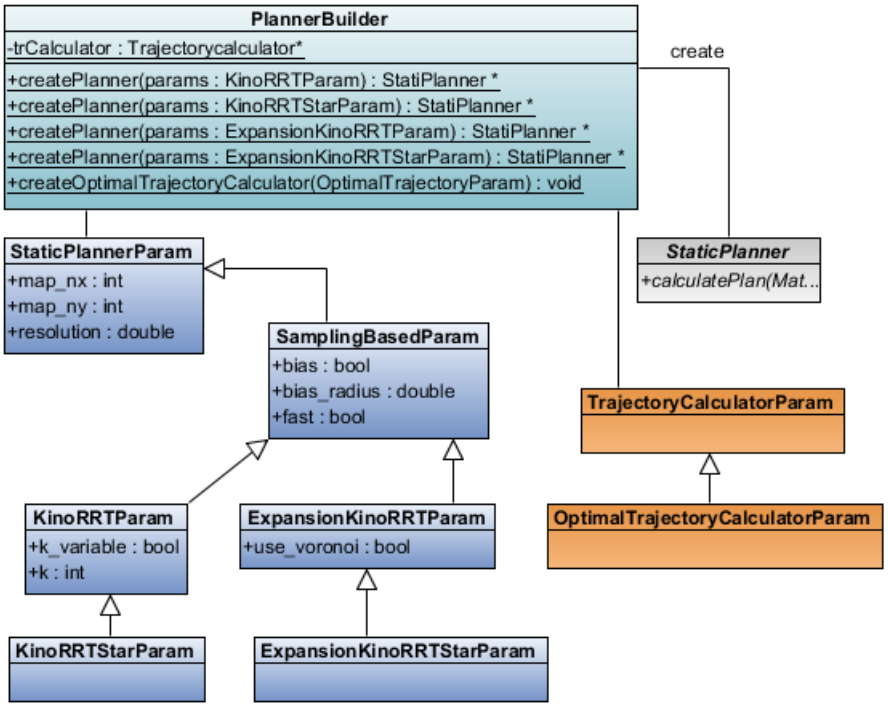


Figura 3.11: Struttura della classi del planner builder

La progettazione strutturale proposta è riassunta in Figura 3.11. Utilizzando uno dei metodi di creazione è possibile allocare in memoria un pianificatore di tipo appropriato e restituirne il riferimento. In questo modo il chiamante manipola l'oggetto come se fosse di tipo StaticPlanner, senza preoccuparsi dell'implementazione effettiva. Sarà il builder a garantire la corretta costruzione del pianificatore e a gestirne i dettagli attraverso l'*overloading* dei metodi: in base al tipo del parametro in ingresso verranno eseguiti comportamenti diversi ed appropriati per impostare i parametri caratterizzanti.

Questo approccio promuove fortemente l'estendibilità e la trasparenza. Infatti, per estendere la classe PlannerBuilder, è sufficiente creare i nuovi tipi di parametri e definire un nuovo overload del metodo di creazione.

L'utilizzo di un builder agevola anche la costruzione automatica del pianificatore attraverso un file di configurazione. Un semplice parser può leggere un documento strutturato (ad esempio xml o json) per valorizzare i parametri degli oggetti da fornire al costruttore. In questo modo viene completata in modo semplice e flessibile la configurazione del sistema di pianificazione.

Capitolo 4

Risultati sperimentali

Nel capitolo 2 sono state proposte delle tecniche innovative per la risoluzione del problema di kinodynamic motion planning, la cui formulazione è stata definita nel Capitolo 1. Esse hanno trovato una applicazione nel Capitolo 3 in cui è stato realizzato il software per il progetto SHERPA. A fronte della definizione di queste nuove tecniche di risoluzione è importante stimare la qualità degli algoritmi proposti. In questo capitolo verranno quindi analizzati i risultati ottenuti in termini quantitativi.

Come già citato, il kinodynamic motion planning si presenta in una forma generale: molte situazioni possono essere modellate come un problema di questo tipo. Data la natura pratica del software sviluppato, i test prenderanno in considerazione il caso dello spostamento di un robot tra due punti fissati in un ambiente con ostacoli.

Determinare le performance degli algoritmi sampling-based non è triviale per diversi motivi. Dato che questi algoritmi sono basati su campionamento, le performance non possono essere giudicate da una singola esecuzione. Occorre quindi un *benchmark* su cui possano essere eseguiti ripetutamente dei test per ottenere la *distribuzione* di metriche di performance significative.

Nella prima parte di questo capitolo verranno presentati i benchmark, cercando di approfondire quelli proposti dallo stato dell'arte. In modo simile si procederà con le metriche di valutazione. Nella seconda parte invece saranno analizzate le performance ottenute dai diversi algoritmi eseguiti su tutti i benchmark scelti.

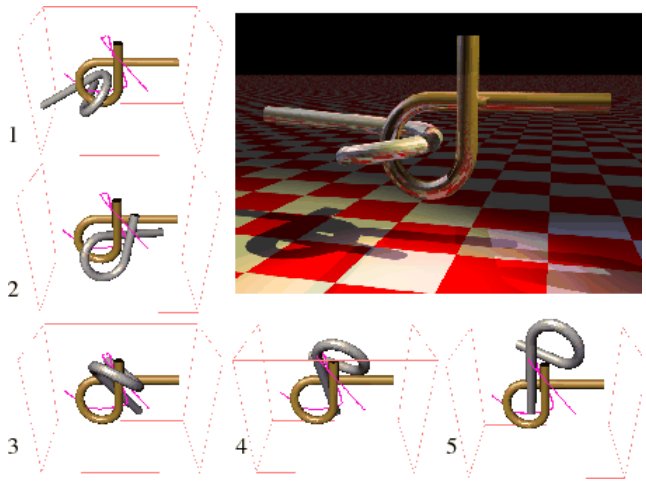


Figura 4.1: Il problema noto come alpha puzzle, creato da Boris Yamrom

4.1 Benchmarking e metriche di valutazione

Definire il benchmark da utilizzare è un aspetto molto importante nella valutazione delle performance e nel confronto degli algoritmi. Una scelta opportuna infatti permette confronti più chiari ed è utile in molte altre situazioni, come la scelta dei parametri o la valutazione delle performance nel tempo. Inoltre utilizzare un benchmark esistente facilita il confronto con gli algoritmi presenti allo stato dell'arte e con quelli che verranno definiti in futuro.

4.1.1 Benchmark disponibili

Lo studio e lo sviluppo degli algoritmi di motion planning è spesso accompagnato da test su un'ampia varietà di benchmark. Il più citato è quello noto come alpha puzzle (Figura 4.1), presentato anche in [7]: si tratta di dividere due anelli non deformabili, mettendo alla prova l'algoritmo su problemi che presentano passaggi stretti.

Molti software che implementano algoritmi di motion planning, come *Open Motion Planning Library* (OMPL) [10], includono anche una forma di benchmarking interno. Nel caso di OMPL, la libreria può essere estesa con algoritmi propri per effettuare dei test su problemi realizzati ad hoc. I risultati possono inoltre essere confrontati con quelli implementati internamente nella libreria.

L'importanza del benchmark ha favorito anche lo sviluppo di sistemi per il confronto di planner. Tra i principali sistemi troviamo *MoveIt!* [3, 12] nato con l'idea di fornire un set di benchmark facilmente accessibile, ampliabile ed eseguibile per tutti i pianificatori. Inoltre questo framework è stato integrato in ROS. L'obiettivo è quello di evidenziare i vantaggi e gli svantaggi dei diversi approcci nelle molteplici situazioni in cui possono trovarsi.

I sistemi di benchmarking citati presentano alcune forti limitazioni. Il problema noto come alpha puzzle è un tipico problema di *manipolazione* in cui è richiesto al pianificatore di determinare i movimenti degli anelli. Al contrario, gli algoritmi proposti sono stati pensati per risolvere i problemi di *navigazione*.

Per quanto riguarda OMPL, la libreria si basa su classi interne, talvolta molto dettagliate, che includono dipendenze non sostituibili. Questo rende difficile incorporare il software sviluppato senza rivedere in modo radicale l'architettura proposta. L'utilizzo di un framework esterno per il solo scopo di benchmarking renderebbe molto difficile il trasferimento della libreria su un robot reale.

Infine il framework di MoveIt! esegue ogni planner disponibile per risolvere tutti i problemi memorizzati nel database, posto su un server centrale, e presenta infine informazioni di tipo statistico. Al contrario l'obiettivo dei nostri test non è determinare le performance degli algoritmi di pianificazione in tutti i possibili problemi, ma concentrarsi unicamente su quello di navigazione.

Data la complessità del problema di motion planning, che comprende molti campi applicativi, individuare un benchmark appropriato si è rivelato un compito complesso. Si è ritenuto quindi opportuno avvalersi di alcune linee guida per definire un set di test su cui determinare la distribuzione di metriche definite dallo stato dell'arte.

4.1.2 Definizione del benchmark utilizzato

Quando si definisce un benchmark per i test è importante evitare di utilizzare problemi specifici o casi particolari per gli algoritmi: realizzare un ambiente di simulazione che non favorisca il programma da testare è necessario per la corretta valutazione dei test, ma anche con questa consapevolezza può essere molto difficile. Spesso vengono fatte assunzioni implicite, anche inconsciamente, sia durante la progettazione del software, sia durante la definizione dei benchmark. Inoltre è noto che alcune strategie di campionamento, quella

indirizzata al goal per citarne una, di solito forniscono risultati migliori per alcune classi di problemi.

In [1] sono state proposte alcune linee guida per la definizione dei benchmark. La caratteristica fondamentale consiste nel trattare un task specifico e misurare le performance del pianificatore. Inoltre le metriche quantitative di analisi devono essere comprensibili senza conoscere i dettagli dell'architettura sottostante. Infine è importante che i problemi di benchmark siano rappresentativi, per quanto possibile, del mondo reale.

La descrizione dei benchmark utilizzati è suggerita da [3], in cui si richiede la specifica secondo quattro dimensioni:

- **Modello dell'ambiente** - È l'ambiente utilizzato per il problema di motion planning; deve rappresentare la sua effettiva struttura, come la posizione di porte, muri e altri oggetti. Può essere generato artificialmente o acquisito attraverso sensori.

- **Modello del robot** - È una descrizione del robot utilizzato per i test, possibilmente completo della sua dinamica e vincoli sul controllo.

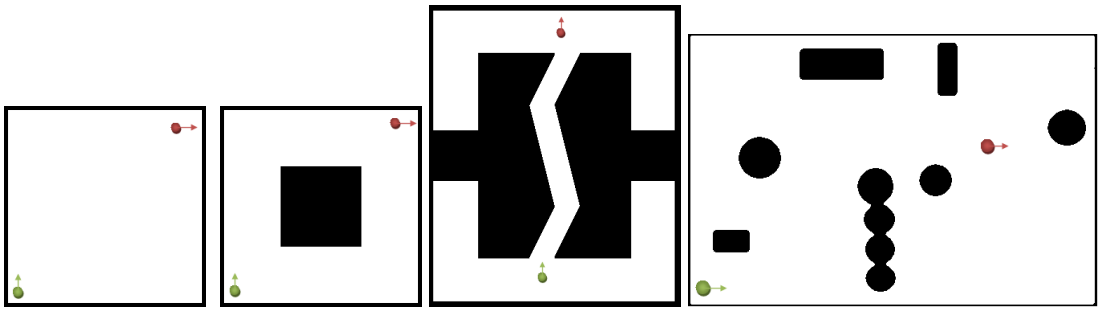
- **Goal** - Associato all'ambiente, deve essere definito un goal o una regione di goal. Dipende dal problema trattato, dal modello del robot e dalla definizione dello spazio degli stati X.

- **Start** - Come per il goal, deve essere specificata anche la configurazione di inizio del robot in relazione all'ambiente definito, al modello del robot e ad X.

Il modello del robot utilizzato è un rover a controllo differenziale la cui dinamica è non-lineare, lo stesso descritto ed analizzato nei Capitoli 2 e 3. Ricordiamo che lo spazio degli stati è composto da $\mathbf{x} = (x, y, \theta, v, k)^T$ e gli input di controllo $\mathbf{u} = (u_v, u_k)^T$ consistono rispettivamente nella derivata della velocità e della curvatura.

Per descrivere l'ambiente sono state predisposte quattro mappe, memorizzate sotto forma di immagini binarie. La posizione sul piano del robot è riportata assumendo gli assi cartesiani aventi origine in basso a sinistra e utilizzando questo come sistema di riferimento inerziale. La risoluzione della mappa determina l'effettiva dimensione della stanza e può essere variata liberamente.

Lo scopo delle mappe è quello di analizzare in modo completo la maggior parte delle casistiche trattabili. Partendo dalle situazioni più semplici di una mappa priva di ostacoli



(a) (b) (c) (d)

Figura 4.2: I quattro ambienti utilizzati per i test

(la prima) si procederà complicando il problema, aggiungendo regioni non transitabili (la seconda) e singolarità per trarre in inganno gli algoritmi (la terza), fino a giungere a un caso reale, con molteplici ostacoli (la quarta).

La prima mappa, mostrata in Figura 4.2a, consiste in una regione completamente libera da ostacoli, ad eccezione dei bordi che delimitano l'area. Ha dimensioni di 50×50 m^2; questo produce vincoli ulteriori sulla dinamica del sistema, in particolare le posizioni sul piano per cui deve valere $p_x, p_y \in [0, 50]$. Lo stato iniziale e finale del problema sono rispettivamente $\mathbf{x}_{start} = (2, 2, 90°, 0.1, 0.0)^T$ e $\mathbf{x}_{goal} = (42, 42, 0°, 0.1, 0.0)^T$. L'obiettivo di questa mappa è quello di verificare il comportamento del pianificatore nella situazione più semplice possibile, in cui non sono presenti ostacoli da aggirare ed è ammesso un collegamento diretto tra start e goal. Nonostante questo, date le condizioni iniziali e finali del problema, il pianificatore dovrà essere in grado di produrre una traiettoria curvilinea, ma regolare ed entro i limiti della mappa.

La seconda mappa, mostrata in Figura 4.2b, consiste in un semplice ostacolo centrale che deve essere circumnavigato per raggiungere il goal. Ha una dimensione di 50×50 m^2 e lo stato iniziale e finale del problema sono rispettivamente $\mathbf{x}_{start} = (2, 2, 90°, 0.1, 0.0)^T$ e $\mathbf{x}_{goal} = (42, 42, 0°, 0.1, 0.0)^T$. L'obiettivo di questa mappa è quello di proporre un problema triviale in cui è richiesto al robot di evitare un ostacolo di grandi dimensioni. Si noti che le due modalità per aggirare l'ostacolo sono equivalenti, in quanto la posizione iniziale e finale sono poste sulla diagonale dell'ostacolo.

La terza mappa[1], mostrata in Figura 4.2c, richiede al robot di passare attraverso un lungo passaggio in cui è presente qualche curva non molto stretta. Ha dimensioni di $150 \times 179\ \mathrm{m}^2$ e lo stato iniziale e finale del problema sono rispettivamente $\mathbf{x}_{start} = (65, 15, 90°, 0.1, 0.0)^T$ e $\mathbf{x}_{goal} = (80, 160, 90°, 0.1, 0.0)^T$. L'obiettivo di questa mappa è di verificare il comportamento del pianificatore in situazioni in cui gli ostacoli sono molto più ampi e i passaggi stretti. Le posizioni iniziali e la conformazione dell'ambiente fa sì che non sia possibile determinare un percorso in linea retta. Inoltre le regioni libere da ostacolo poste a lato del tunnel sono state inserite per aggiungere elementi che possono ingannare il pianificatore.

Infine la quarta mappa (Figura 4.2d) è stata liberamente presa da [11] e descrive quello che può essere un ambiente reale in cui sono stati posti diversi ostacoli di piccole e medie dimensioni. Ha dimensioni di $225 \times 150\ \mathrm{m}^2$ e lo stato iniziale e finale del problema sono rispettivamente $\mathbf{x}_{start} = (10, 10, 0°, 0.1, 0.0)^T$ e $\mathbf{x}_{goal} = (170, 85, 0°, 0.1, 0.0)^T$. L'obiettivo dell'ambiente è di simulare un ambiente reale, in cui cui il robot deve evitare molteplici ostacoli posti sul suo percorso. Anche in questo caso sono presenti delle singolarità che possono ingannare il pianificatore, in particolare nel posizionamento ravvicinato dei quattro oggetti circolari.

4.1.3 Metriche utilizzate

Per misurare le performance degli algoritmi di pianificazione è utilizzato un insieme di metriche, proposte in [3]. Queste permettono di valutare l'affidabilità e la qualità delle soluzioni restituite. Come metrica ulteriore, introdotta dall'utilizzo di un algoritmo di calcolo della traiettoria ottima (π^*), viene utilizzato anche il costo della soluzione π restituita dal pianificatore. La funzione di costo è stata proposta in [13] ed è stata liberamente scelta come indicatore della qualità delle soluzioni.

- Tempo di computazione del piano (t_{plan}) - È il tempo impiegato dall'algoritmo per restituire una soluzione a partire dalla richiesta di pianificazione. Include sia quello necessario alla costruzione dell'albero, sia quello necessario per la risoluzione della query, ovvero per l'estrazione del percorso dall'albero.

[1] `https://parasol.tamu.edu/groups/amatogroup/benchmarks/data/mp/ZigZag/`

- Percentuale di successi (s_r) - Rappresenta la percentuale di casi in cui il pianificatore ha restituito una soluzione entro il tempo limite a disposizione.

- Costo della soluzione (c_{plan}) - È un valore adimensionale che penalizza sia la durata della traiettoria sia lo sforzo per controllare il robot. Il costo associato ad una soluzione è pari alla somma dei costi delle traiettorie che la compongono: $c_{plan} = \sum_i c[\pi_i]$, in cui $c[\pi] = \int_0^\tau \left(1 + \mathbf{u}[t]^T R\mathbf{u}[t]\right) dt$ e $R \in \mathbb{R}^{m \times m}$ è una matrice definita positiva e costante. R ha lo scopo di dare un peso d'importanza diverso agli input di controllo e alla durata della traiettoria. Per il sistema in esame $R = 0.25 \cdot I$.

4.2 Analisi dei risultati

La definizione di benchmark garantisce la possibilità di replicare i test e di confrontare le performance ottenute dai diversi algoritmi. Inoltre i problemi proposti sono stati scelti in un'ottica di riusabilità, così da promuovere il confronto futuro.

In questa parte saranno prima fornite le informazioni complete relative ai test, tenendo come riferimento gli algoritmi descritti nel Capitolo 2, quindi verranno analizzati i dati ottenuti.

4.2.1 Algoritmi Analizzati

Diversi sono gli algoritmi messi a disposizione dal sistema software descritto nel Capitolo 3:

- Kinodynamic RRT*: l'algoritmo proposto in [13], revisionato per rispettare i vincoli di linearizzazione.

- Kinodynamic RRT: la versione non ottima dell'algoritmo precedente. Il costo computazionale suggerisce una maggiore velocità ed è infatti interessante confrontare i tempi di esecuzione.

- Expansion Kino RRT: l'algoritmo basato su espansione dell'albero a partire dai nodi che lo compongono.

	Empty Map	Center Obstacle	ZigZag	Multiple Obstacle
Start	(2, 2, 90°, 0.1, 0)	(2, 2, 90°, 0.1, 0)	(65, 15, 90°, 0.1, 0)	(10, 10, 0°, 0.1, 0)
Goal	(42, 42, 0°, 0.1, 0)	(42, 42, 0°, 0.1, 0)	(80, 160, 90°, 0.1, 0)	(170, 85, 0°, 0.1, 0)
NumMaxCicli	500	500	500	500
NumMaxNodi	500	500	500	500

	KinoRRT	Expansion KinoRRT	KinoRRT*	Expansion KinoRRT*
fast	Sì	Sì	No	No
k	NumMaxNodi	N.A.	NumMaxNodi	N.A.
use_voronoi	N.A	Sì	N.A	Sì
bias	No	No	No	No

Tabella 4.1: Tabella riassuntiva dei parametri

- Expansion Kino RRT*: la versione asintoticamente ottima del precedente in cui viene inclusa la procedura di aggiornamento.

- Expansion KinoRRT*-First: l'algoritmo procede come nella versione ottima, con l'eccezione che interrompe l'esecuzione nel momento in cui determina la prima soluzione

Le performance dei cinque algoritmi verranno valutate sui benchmark definiti, ovvero sulle quattro mappe. Ognuno di questi sarà eseguito 30 volte, generando diversi alberi esplorativi e determinando differenti soluzioni. Per ognuna di queste verranno calcolata le metriche descritte in precedenza, ovvero t_{plan}, s_r e c_{plan}.

Gli algoritmi verranno impostati per eseguire un numero massimo di cicli uguale a 500 per tutte le mappe. Il numero massimo di nodi sarà impostato pari al numero massimo di cicli. Questo valore è stato scelto dopo alcuni test preliminari: il numero di iterazioni doveva essere tale da fornire risultati significativi in tempi ragionevoli. La scelta di un valore che permetta di restituire in poco tempo la soluzione è ulteriormente supportato dal contesto applicativo SHERPA per cui questi algoritmi sono stati sviluppati, ovvero quello del soccorso alpino.

I due algoritmi Kinodynamic RRT e Kinodynamic RRT* utilizzeranno un numero di vicini k costante ad ogni iterazione ed equivalente al numero di nodi. Questo significa considerare tutti i nodi dell'albero. Nel caso degli algoritmi basati su espansione viene utilizzata la scelta del nodo, considerando le regioni di Voronoi. I parametri degli algoritmi sono riassunti nella Tabella 4.1.

Infine ricordiamo che gli algoritmi Kinodynamic RRT e Expansion KinoRRT restituiranno la prima soluzione disponibile. Questo perché la loro esecuzione non garantisce un miglioramento della soluzione all'aumentare del numero di nodi dell'albero, in quanto sono probabilisticamente completi. Al contrario, le controparti ottime completeranno tutte le iterazioni a loro disposizione e, solo alla fine, connetteranno il goal. Unica eccezione è Expansion KinoRRT*-First, il cui obiettivo è quello di valutare la qualità delle prime soluzioni determinate con l'algoritmo ottimo.

4.2.1.1 Ambiente di test

Per completare l'analisi vengono riportate le specifiche dell'ambiente in cui sono stati effettuati i test:

- Il computer utilizzato dispone di un processore a Intel Atom N570 Dual Core a 1.66 GHz e 2 GB di RAM.
- Il sistema utilizzato è Ubuntu 14.04.1 con versione del Kernel Linux 3.19.0-61-generic (i686). Inoltre la distribuzione di ROS installata è Indigo.
- La generazione dei 30 alberi è basata sull'algoritmo pseudo casuale della libreria standard ISO per il linguaggio C. I valori dei seed vanno da 40 a 69, compresi.

4.2.2 Risultati

Al termine di ogni esperimento eseguito sono stati raccolti i tempi di pianificazione e i costi finali delle soluzioni. Questi verranno presentati sottoforma di dati aggregati e box plot.

La tabella 4.2 riporta le percentuali di successi s_r espresse come rapporto tra numero di soluzioni restituite e numero complessivo di prove. Le versioni Kinodynamic RRT e

	EmptyMap	**ObsCenter**	**ZigZag**	**MultipleObs**
KinoRRT	0/30	0/30	0/30	0/30
*KinoRRT**	0/30	0/30	0/30	0/30
Expansion	28/30	28/30	18/30	19/30
Expansion-First*	28/30	28/30	18/30	19/30
*Expansion**	28/30	28/30	18/30	19/30

Tabella 4.2: Percentuale di successi (S_r)

		EmptyMap	**ObsCenter**	**ZigZag**	**MultipleObs**
KinoRRT	Min	0,05	0,05	0,05	0,04
	Max	2,76	2,73	3,07	4,39
	Avg(std)	0,79 (0,66)	0,85 (0,67)	1,31 (0,74)	1,64 (0,91)
*KinoRRT**	Min	0,05	0,05	0,05	0,05
	Max	3,07	3,01	2,81	4,41
	Avg(std)	0,84 (0,74)	0,93 (0,76)	1,25 (0,67)	1,67 (0,92)
Expansion	Min	2,53	6,42	12,65	9,61
	Max	91,56	79,29	94,6	96,01
	Avg(std)	34,72 (27,87)	40,17 (23,16)	48,29 (24,7)	53,27 (31,47)
Expansion-First*	Min	1,27	7,27	13	7,5
	Max	355,59	224,03	159,86	378,62
	Avg(std)	93,48 (102,83)	97,09 (71,36)	56,23 (33,45)	93,85 (109,95)
*Expansion**	Min	302,6	163,23	45,44	53,27
	Max	561,65	437,9	394,07	529,21
	Avg(std)	408,46 (69,59)	298,52 (57,42)	186,3 (71,78)	285,83 (121,12)

Tabella 4.3: Tempi di pianificazione (in secondi) minimi, massimi e medi delle versioni

Kinodynamic RRT* in cui sono stati inseriti i vincoli di linearizzazione non hanno restituito alcuna soluzione. Tuttavia da alcuni test preliminari è emerso che gli algoritmi sono in grado di fornire una soluzione utilizzando un numero maggiore di iterazioni (circa 2500).

Nella tabella 4.3 sono riportati i tempi di computazione minimi, massimi, medi e la deviazione standard dalla media. I tempi molto ridotti degli algoritmi Kinodynamic RRT e Kinodynamic RRT* sono dovuti al numero ridotto di successi. Infatti la procedura che verifica il rispetto dei vincoli di linearizzazione è molto veloce e molte sono le iterazioni dell'algoritmo in cui non sono eseguite operazioni.

Coerentemente con quanto suggerito dai costi computazionali degli algoritmi, la versione Expansion KinoRRT impiega meno tempo per fornire una soluzione, rispetto alla

		EmptyMap	**ObsCenter**	**ZigZag**	**MultipleObs**
Expansion	Min	67,89	78,99	135,59	188,65
	Max	129,85	160,97	293,77	307,48
	Avg(std)	95,87 (16,86)	115,94 (21,79)	214,25 (36,16)	256,73 (38,71)
Expansion-First*	Min	55,7	53,39	101,41	120,39
	Max	115,54	134,93	189,67	243,8
	Avg(std)	80,98 (16,51)	89,42 (19,45)	143,01 (27,25)	175,62 (34,56)
*Expansion**	Min	41,42	53,39	64,33	75,5
	Max	75,13	109,09	164,91	163,13
	Avg(std)	60,06 (8,35)	78,11 (14,13)	106,49 (26,1)	123,43 (26,82)

Tabella 4.4: Costi delle soluzioni minimi, massimi e medi

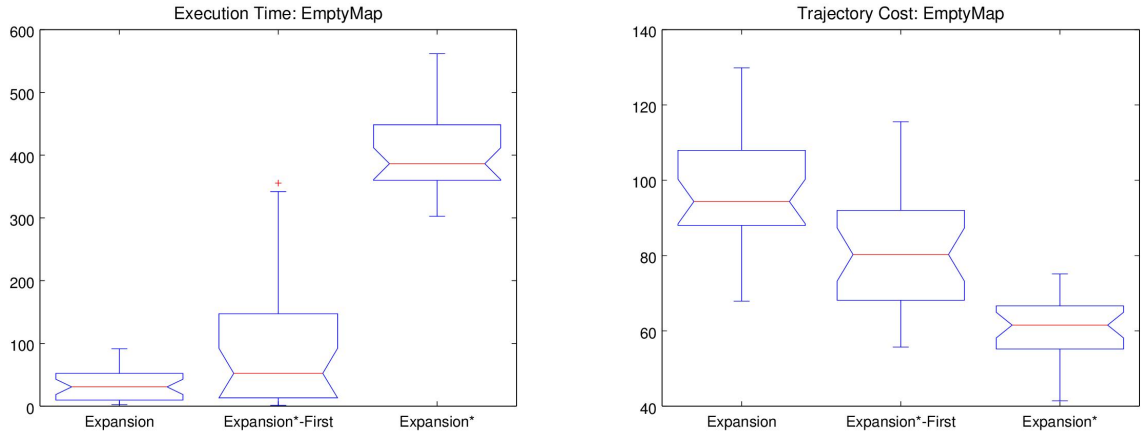
controparte ottima. Una prima soluzione viene restituita dall'algoritmo Expansion KinoRRT* in tempi poco superiori (riga denominata *Expansion*-First*), ma è interessante notare l'elevato scostamento dal valore medio. Questo indica che le diverse esecuzioni hanno restituito una prima soluzione in tempo molto breve, ma altre hanno richiesto un'attesa prolungata.

Nella Tabella 4.4 sono riportati i costi delle traiettorie degli algoritmi con una percentuale di successi superiore allo zero. Ulteriori analisi relative ai costi sono possibili considerando le singole mappe che compongono il benchmark. I dettagli più interessanti possono essere evidenziati comparando i box plot di tempi di pianificazione e costi delle traiettorie.

Analizzando le Figure da 4.3 a 4.6 emergono alcuni dati comuni legati ai tempi di computazione del piano t_{plan} e ai costi c_{plan}. La versione Expansion KinoRRT è in grado di fornire una soluzione in tempi limitati e contenuti; tempi simili sono necessari per determinare una prima soluzione utilizzando la versione ottima, la quale però presenta tempistiche maggiori nel caso in cui si attenda il completamento di tutti i cicli di esecuzione.

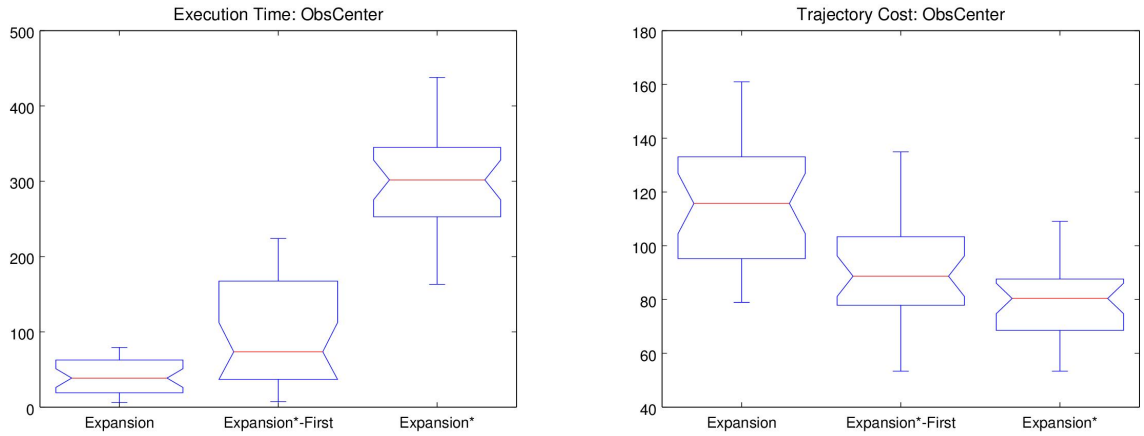
I box plot relativi ai costi provano che l'introduzione dell'aggiornamento dell'albero migliora il valore c_{plan} associato alla soluzione: infatti, in tutte le mappe, Expansion KinoRRT fornisce dei risultati qualitativamente inferiori rispetto le controparti ottime. Inoltre all'aumentare del numero di cicli utilizzabili dall'algoritmo Expansion KinoRRT* il costo delle traiettorie andrà a diminuire; questo è provato dal divario che vi è tra il costo relativo alla prima soluzione restituita e quella ottenuta al termine dei cicli.

Nella mappa in Figura 4.2c in cui la regione di ostacoli è molto ampia e il passaggio



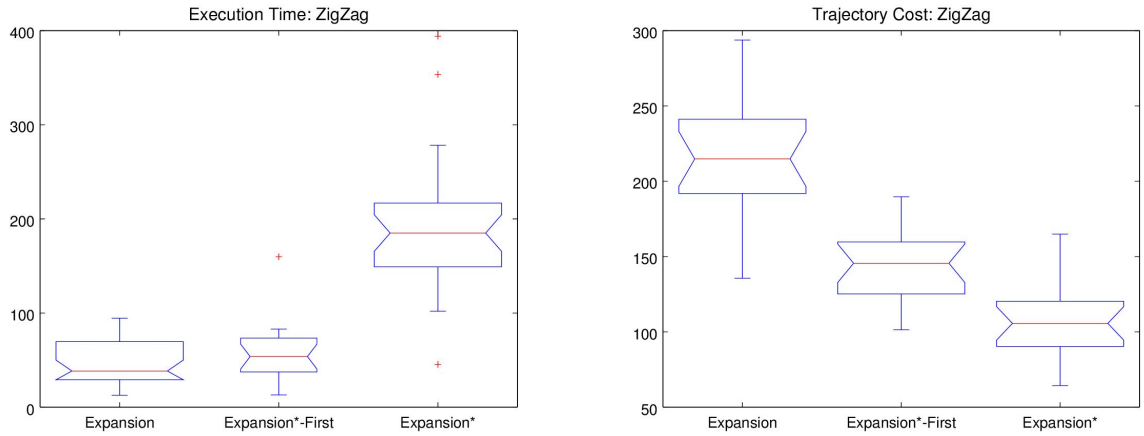
(a) (b)

Figura 4.3: Tempi di pianificazione e costi relativi alla mappa priva di ostacoli



(a) (b)

Figura 4.4: Tempi di pianificazione e costi relativi alla mappa con un ostacolo centrale



(a) (b)

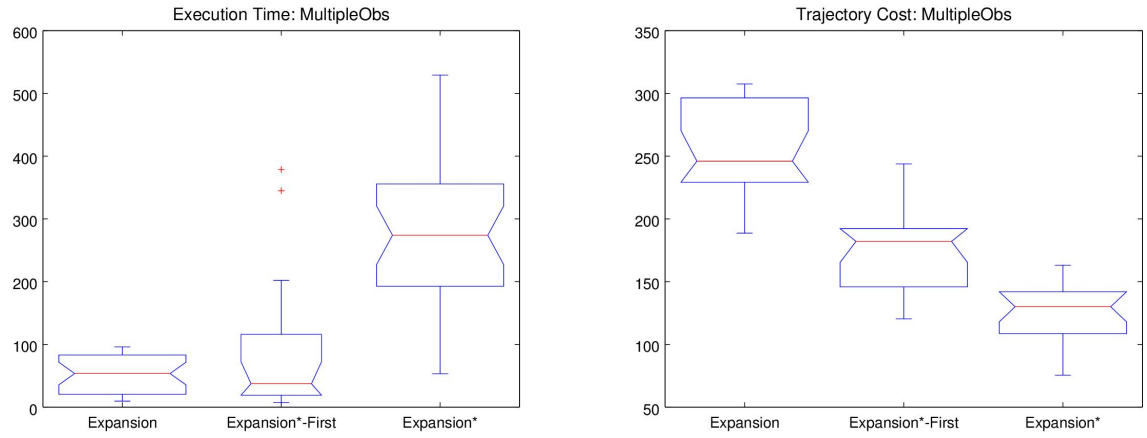
Figura 4.5: Tempi di pianificazione e costi relativi alla mappa con il passaggio stretto

per giungere al goal è stretto, i tempi di pianificazione sono sorprendentemente ridotti, ad eccezione di qualche raro caso. È interessante osservare che i tempi degli algoritmi Expansion ed Expansion*-First sono tra loro molto simili, nonostante i costi delle soluzioni siano significativamente diversi (Figura 4.5). Caratteristica importante di questa mappa è tuttavia il posizionamento dei punti $\mathbf{x}_{start}$ e $\mathbf{x}_{goal}$ aventi la medesima orientazione: questo elemento potrebbe essere un dettaglio in grado di favorire la convergenza dell'algoritmo.

Alcuni aspetti peculiari, deducibili confrontando i grafici, caratterizzano le singole mappe e il comportamento degli algoritmi. Nel caso della mappa in cui è presente un ostacolo centrale (Figura 4.2b) è interessante notare la diminuzione dei tempi per quanto riguarda Expansion KinoRRT* rispetto quelli riportati in 4.3: il dato è indice del fatto che la presenza di un ostacolo che limita lo spazio degli stati X_{free} esplorabile favorisce la convergenza dell'algoritmo.

Infine la mappa con diversi ostacoli (Figura 4.2d) fornisce dettagli interessanti relativi al tempo di pianificazione relativi alla versione ottima. Questa infatti presenta un box plot (Figura 4.6) che evidenzia l'elevata variabilità dell'algoritmo per giungere ad una soluzione. Al contrario per restituire una prima soluzione è richiesto un tempo ridotto, ad eccezione di qualche raro caso.

Al fine di completare l'analisi relativa ai risultati ottenuti si è ritenuto interessante riportare l'evoluzione del costo associato al goal durante la computazione del piano. Si ricordi



(a) (b)

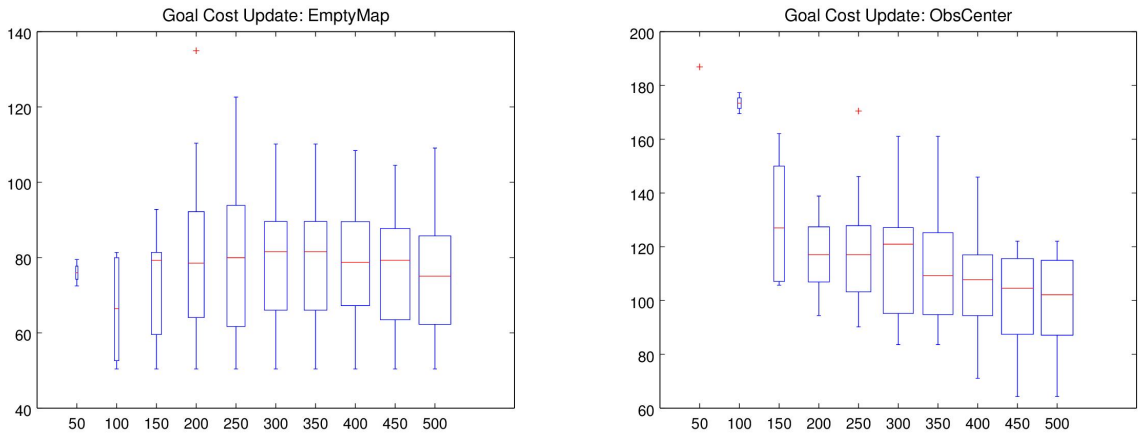
Figura 4.6: Tempi di pianificazione e costi relativi alla mappa con molteplici ostacoli

che il valore associato a un nodo x_i indica qual'è il costo della traiettoria minima che dalla radice x_{start} porta ad x_i. Quindi il valore relativo a x_{goal} indica il costo della soluzione calcolata all'iterazione corrente ed è sempre minimo.

Per ognuno dei 30 esperimenti è stato letto, ad intervalli regolari di 50 iterazioni, il costo associato al goal: i valori sono riportati nei boxplot di Figura 4.7, uno per ogni mappa. L'ampiezza dei grafici è indice del numero di dati disponibili: i box più stretti indicano che la rappresentazione è basata su pochi valori, dove invece quelli più grandi su un numero maggiore. Dalle Figure, in particolare 4.7a, 4.7b e 4.7d, risulta quindi che ad un numero ristretto di iterazioni l'algoritmo avrà infrequentemente determinato un percorso verso il goal. All'aumentare del numero di iterazioni, al contrario, è più probabile che sia stata determinata una prima soluzione. Inoltre la riduzione della mediana e dell'ampiezza del box suggeriscono una diminuzione del costo all'aumentare del numero di iterazioni, come riportato dalle Figure 4.7b, 4.7c e 4.7d. I dati raccolti relativi a tutte le mappe suggeriscono tuttavia che questa convergenza è piuttosto lenta.

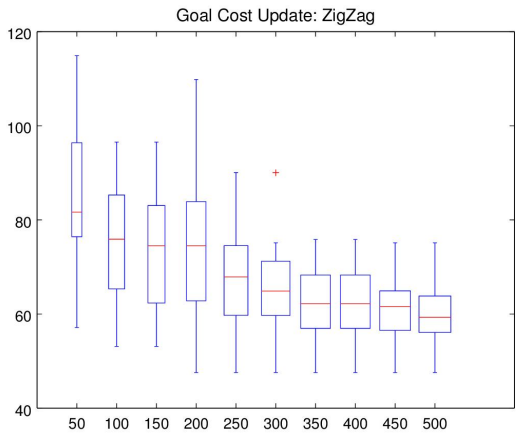
I risultati ottenuti dagli esperimenti permette di trarre alcune interessanti considerazioni di natura pratica:

1. Se l'interesse nella risoluzione del problema riguarda maggiormente le tempistiche con cui è restituita la soluzione del problema, allora è possibile utilizzare l'algoritmo

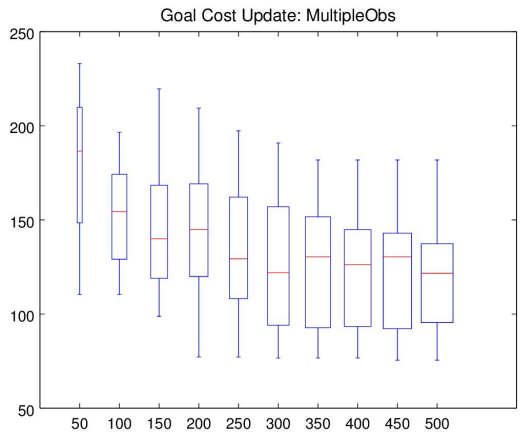


(a)

(b)



(c)



(d)

Figura 4.7: Evoluzione dei costi relativi al goal

Expansion KinoRRT. Questa versione infatti è in grado di fornire una soluzione in tempi ridotti e costi associati non molto elevati.

2. I dati riguardanti i tempi di computazione e i costi della prima soluzione calcolata dall'algoritmo Expansion KinoRRT* suggeriscono che questa versione è una valida alternativa per contesti applicativi. Infatti si caratterizza per il costo delle soluzioni, migliore rispetto quello di Expansion KinoRRT, e un tempo di calcolo leggermente superiore.

3. I diversi approcci possono essere anche combinati: inizialmente è possibile attendere una prima soluzione e, una volta ottenuta, eseguire il piano prodotto. Durante l'esecuzione l'algoritmo Expansion KinoRRT* potrà continuare a migliorare l'albero e suggerire correzioni migliorative alla traiettoria. In alternativa è possibile avviare due computazioni parallele.

Conclusioni e sviluppi futuri

Al termine di questa tesi è stato prodotto un software di pianificazione per il rover SHERPA in grado di garantire il rispetto dei vincoli nonolonomici del sistema. Lo sviluppo ha richiesto una prima fase di studio dei problemi di motion planning e kinodynamic motion planning, nonchè l'analisi degli algoritmi disponibili allo stato dell'arte, descritti nel Capitolo 1. Il lavoro iniziale ha permesso di formare una solida base per descrivere gli algoritmi di pianificazione basati su campionamento, come Rapidly-exploring Random Tree (RRT), Optimal Rapidly-exploring Random Tree (RRT*) e Kinodynamic RRT*. Elementi fondamentali di questi sono la tecniche di campionamento e l'update dell'albero per memorizzare i percorsi minimi dal nodo radice. Inoltre la versione Kinodynamic presenta diverse peculiarità nel calcolo della traiettoria e nella linearizzazione rispetto il punto campionato.

I maggiori contributi sono individuabili nel Capitolo 2, in cui sono state evidenziate le limitazioni della tecnica di linearizzazione per il sistema in esame. Questi vincoli sono stati sfruttati per definire una nuova tecnica di risoluzione che prende il nome di Expansion KinoRRT. L'algoritmo proposto sceglie, iterazione dopo iterazione, un nodo da espandere appartenente all'albero e determina il punto campione entro una regione ammissibile in grado di rispettare i vincoli di linearizzazione; se è possibile calcolare una traiettoria in grado di connettere il nodo espanso e quello campione, allora quest'ultimo verrà aggiunto all'albero. Includendo la procedura di update, parte integrante degli algoritmi di pianificazione ottimi, è stata prodotta anche una versione Expansion KinoRRT*. Il capitolo prende inoltre in considerazioni ulteriori aspetti algoritmici, come l'utilizzo delle regioni di Voronoi per il campionamento, la definizione della versione non ottima di Kinodynamic RRT* e l'introduzione di un bias verso il goal.

La progettazione del pianificatore è riportata nel Capitolo 3, in cui i requisiti tecnici trovano la loro definizione e formalizzazione. In questo contesto i capitoli precedenti hanno

permesso di costruire il modello del dominio, dettagliando cosa si intende per pianificatore, stato del robot, dinamica del sistema e traiettoria. A seguito dell'analisi dei requisiti è stata proposta l'architettura logica di riferimento basata sul paradigma ad oggetti e descritta in termini strutturali, di interazione e di comportamento. L'architettura è stata poi raffinata e arricchita dai diversi algoritmi analizzati in precedenza, rendendo possibile la coesistenza di diversi approcci per la risoluzione del problema di navigazione del robot. Volendo garantire il rispetto della dinamica del sistema sono state previste anche le classi per il calcolo della traiettoria.

Dato che il principale contributo di questo lavoro risiede nella definizione di nuovi approcci alla risoluzione del problema di kinodynamic motion planning, nel Capitolo 4 sono presentati i risultati ottenuti. A tal proposito sono riportati alcuni benchmark disponibili allo stato dell'arte. Tuttavia, non essendo questi appropriati per la trattazione del problema di navigazione, è stato proposto un nuovo insieme di mappe descritte sistematicamente seguendo i criteri proposti in letteratura. È stata quindi portata avanti una serie di esperimenti per determinare le performance degli algoritmi in termini di tempo di computazione, costo della soluzione e percentuale di successi. Dai test è emerso che le versioni basate su espansione sono in grado di restituire più frequentemente una soluzione rispetto a Kinodynamic RRT e con un tempo ridotto. Inoltre il costo associato alla traiettoria decresce all'aumentare del numero di iterazioni eseguite. Infine Expansion KinoRRT risulta un buon compromesso per quei sistemi interessati primariamente al tempo di calcolo perchè le soluzioni non hanno un costo troppo elevato e il tempo è il più basso. Vi è anche la possibilità, qualora non si voglia utilizzare Expansion KinoRRT, di utilizzare la versione ottima per determinare una prima soluzione e, durante l'inseguimento della traiettoria prodotta, continuare la computazione per migliorarla. Nel caso siano disponibili diversi core le due versioni dell'algoritmo possono essere avviate in parallelo.

Il lavoro svolto ha raggiunto gli obiettivi prefissati, ottenendo risultati molto rilevanti. Tuttavia la trattazione del problema di kinodynamic motion planning è ancora in una fase iniziale e diverse sono le tematiche possibili da approfondire e sviluppare.

Relativamente agli algoritmi basati su espansione proposti, merita un'indagine più approfondita l'effetto del bias verso il goal. Durante la fase sperimentale infatti non è stato possibile determinare la variazione del tempo di computazione e la qualità delle soluzioni quando la ricerca è indirizzata verso uno stato specifico. Similmente non sono noti gli effetti del raggio del bias ovvero il parametro che regola la regione, piuttosto che lo stato,

a cui è indirizzato il campionamento.

L'aspetto di maggiore interesse, sia per la rilevanza pratica sia per quella scientifica, riguarda il calcolo della traiettoria di sistemi non lineari. Infatti l'utilizzo della tecnica di linearizzazione per gestire questi ultimi, introduce inevitabili errori di approssimazione. Per questo motivo è molto interessante investigare la possibilità di evitare calcoli inuti quando il problema è sottovincolato.

Merita altri studi approfonditi anche la possibilità di utilizzare un algoritmo per determinare una legge di controllo non ottima. Si ricordi infatti che ad ogni pianificatore viene affiancato un modulo per il calcolo della traiettoria. Quello utilizzato in questo lavoro è in grado di determinare una soluzione ottima a scapito di un elevato tempo di calcolo. Se l'algoritmo di pianificazione utilizzato è basato sul criterio di ammissibilità, la risoluzione del calcolo della traiettoria ottima non è certo un vantaggio. In questo caso infatti si è maggiormente interessati ad ottenere una soluzione nel minore tempo possibile. Questa è una giustificazione sufficiente per spingere a considerare altre tecniche per il calcolo di una traiettoria ammissibile.

Infine dall'analisi degli algoritmi proposti risulta evidente che le tecniche basate su RRT sono tecniche di pianificazione deliberativa. L'approccio seguito è di tipo forward search, ovvero dallo stato iniziale al raggiungimento di uno stato che soddisfa il goal. Per completare l'analisi di questi algoritmi è interessante confrontare le performance del sistema con altri tipi di pianificatori; tra i principali e di maggiore interesse ricordiamo i controlli basati su campi di potenziale.

Bibliografia

[1] J. Baltes. A benchmark suite for mobile robots. In *Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, volume 2, pages 1101–1106 vol.2, 2000.

[2] B. R. Call. *Obstacle avoidance for unmanned air vehicles*. PhD thesis, Brigham Young University, 2006.

[3] B. Cohen, I. A. Sucan, and S. Chitta. A generic infrastructure for benchmarking motion planners. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 589–595, 2012.

[4] B. Donald, P. Xavier, and J. Reif. Kinodynamic motion planning, 1993.

[5] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *Int. J. Rob. Res.*, 30(7):846–894, June 2011.

[6] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[7] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[8] S. M. Lavalle and J. J. Kuffner. Rapidly-Exploring Random Trees: Progress and Prospects. pages 293–308. A K Peters, 2001.

[9] F. L. Lewis and V. L. Syrmos. *Optimal control*. J. Wiley, New York, 1995. A Wiley-Interscience publication.

[10] M. Moll, I. A. Sucan, and L. E. Kavraki. Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization. *IEEE Robotics Automation Magazine*, 22(3):96–102, 2015.

[11] N. D. Munoz-Ceballos and J. A. Valencia-Vel'asquez. Benchmark framework for mobile robots navigation algorithms. *Facultad de Ingeniería*, 23(36):65–73, 2014.

[12] I. A. Sucan and S. Chitta. Moveit!, August 2016.

[13] D. J. Webb and J. van den Berg. Kinodynamic rrt*: Optimal motion planning for systems with linear differential constraints. *CoRR*, abs/1205.5088, 2012.

Ringraziamenti

Desidero ringraziare sentitamente Andrea Roli e Nicola Mimmo per tutto il tempo che mi hanno dedicato in questo periodo. È stato un piacere e un onore lavorare con voi.

Un ringraziamento speciale va anche alla mia morosa Barbara per il suo supporto in questi anni. Questo percorso sarebbe stato impossibile senza di te. Non smetterò mai di apprezzare la tua gentilezza e i tuoi brillanti consigli.

Questa tesi è dedicata alla mia famiglia: voglio ringraziare i miei genitori e mio fratello Roberto per la pazienza che hanno saputo dimostrare nei miei confronti. Senza di voi tutto questo non sarebbe stato possibile: siete stati il mio punto di riferimento in tutti questi anni.

Vorrei ringraziare anche mio zio Sandro, per me fonte continua di orgoglio e ispirazione. Mi hai dimostrato che con l'impegno e la determinazione è possibile realizzare i propri sogni.

Vorrei ringraziare i miei amici e colleghi che hanno percorso questo cammino con me: Pier e Tomas, Lorenzo e Valentina, Erica, Marco e Maurizio. Ognuno di voi ha aggiunto qualcosa a questa esperienza, qualcosa di inestimabile.

Ringrazio anche Francesca con cui ho avuto il piacere di lavorare in questo ultimo periodo e che è all'inizio di questo percorso universitario.

Un grazie anche a Dario, Nicola, Andrea, Manuel, Mattia e Simone. Dopo tutto questo tempo siete ancora con me e per questo voglio ringraziarvi.