Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Automazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Confronto di algoritmi di pianificazione di traiettoria sampling-based per robot mobili

Relatore: Prof. Luca Bascetta

Tesi di Laurea di:
Marco BERTAGGIA Matr. 864220

Anno Accademico 2017–2018

Abstract

The purpose of this thesis is to analyze and compare some **Sampling-based** algorithms for Motion Planning of a mobile robot. The goal of a trajectory planner is to assign an efficient way to drive the robot from an initial state to a final configuration avoiding collisions with obstacles. Unlike some classical techniques, *Sampling-based* algorithms are not based on the a priori knowledge of the complete configuration space. Instead, they sample the space with a fixed number of randomly generated nodes. The complexity of exact planning algorithms is often esponential in the dimensionality of the configuration space and thus they are hardly applicable in practice. The main advantage of *Sampling-based* algorithms is their efficiency for high-dimension configuration spaces. For this reason, these methods have been succesful also in the field of industrial robots. Here, only mobile robots are considered, but the theory could be applied to more complex robots.

The treated algorithms are **PRM**, **RRT** and some of their variants such as their optimal versions **PRM*** and **RRT***[1]. Furthermore, the application of these algorithms to kynodynamic models are reported. In particular **Kinodynamic-RRT***[2] will be presented and simulated. In this work the new algorithm **Kinodynamic-PRM*** is introduced, that is an evolution of PRM* for a kinodynamic model of a mobile robot.

Sommario

Il lavoro svolto in questa tesi si concentra sull'analisi e il confronto di alcuni algoritmi **Sampling-based** per la pianificazione di traiettoria di un robot mobile. Un algoritmo di pianificazione di traiettoria ha lo scopo di calcolare un tragitto il più possibile efficiente che conduca il robot da uno stato iniziale ad uno finale, evitando collisioni con eventuali ostacoli. A differenza di alcune tecniche classiche che prevedono la conoscenza dell'intero spazio delle configurazioni, gli algoritmi *Sampling-based* campionano lo spazio in maniera casuale, generando un numero fissato di nodi. La probabilità con cui questi nodi vengono generati è uniforme; per questo, fissando un adeguato numero di iterazioni, si può assumere di coprire gran parte dello spazio libero. Se negli algoritmi classici il carico computazionale risultava esponenziale all'aumentare dei gradi di libertà del robot, negli algoritmi *Sampling-based* questo non accade. Grazie a questo vantaggio gli algoritmi *Sampling-based* hanno avuto un grande successo anche nell'ambito della robotica industriale. Qui verranno considerati solo robot mobili, ma la teoria che sta alla base degli algoritmi analizzati potrà essere facilmente applicabile a robot più complessi.

Oltre agli algoritmi **PRM** e **RRT**, in questo lavoro verranno analizzate le loro versioni ottime **PRM*** e **RRT***[1]. Inoltre un capitolo intero verrà dedicato all'applicazione degli algoritmi *Sampling-based* a modelli kinodinamici, ovvero a modelli più complessi dotati di vincoli anolonomi. Verrà quindi analizzato e implementato l'algoritmo **Kinodynamic-RRT***[2] e, applicando la stessa teoria all'algoritmo PRM* si è ottenuto un nuovo algoritmo chiamato **Kinodynamic-PRM***.

Introduzione

La Pianificazione di Traiettoria, in Inglese *Motion Planning*, è un argomento di grande rilevanza nella teoria dei sistemi robotici. L'obiettivo è quello di trovare una sequenza di comandi capace di guidare il robot da uno stato iniziale ad una configurazione finale evitando le collisioni con gli eventuali ostacoli. Tale questione viene spesso indicata anche come *Piano Mover's Problem*[4]; si pensi infatti al problema di ricerca della traiettoria considerando l'esempio in cui si voglia trasferire un pianoforte in un'altra stanza all'interno di un appartamento arredato. Quello che potrebbe sembrare un problema semplice, in realtà è decisamente complesso nell'ottica in cui si voglia teorizzarlo in un linguaggio di basso livello comprensibile dalla macchina. Le variabili in gioco sono molteplici, infatti la traiettoria non solo deve ottimizzare il dispendio energetico, ma deve sottostare all'insieme dei vincoli dettati dalla struttura geometrica del pianoforte, dei muri e dell'arredamento, in modo da evitare le possibili collisioni.

Il pianificatore di traiettoria investe un ruolo chiave all'interno del controllore di un robot. Si pensi infatti che un buon pianificatore, e quindi una buona traiettoria, incide enormemente sull'efficienza del robot in termini energetici e di sicurezza.

Un qualsiasi robot è definito da n gradi di libertà, per questo si può definire il suo stato utilizzando un punto dello spazio n-dimensionale. Viene definito $C_{free} \subset \mathbb{R}^n$ il sottospazio delle configurazioni libere, ovvero quelle effettivamente realizzabili dalla fisica del robot e che non lo portino a collidere con gli ostacoli. La pianificazione di traiettoria ha quindi lo scopo di trovare una sequenza di nodi (configurazioni) completamente all'interno di C_{free} partendo dal nodo iniziale q_{start} al nodo finale q_{goal}.

Negli ultimi decenni sono nate diverse classi di algoritmi capaci di risolvere questo problema. Degne di nota sono le classi: *Grid-based search*, *Artificial Potential fields*, *Visibility Graph*, *Reward-based*, e gli algoritmi **Sampling-based**. Appartenenti a quest'ultima classe, vi sono per esempio **PRM** e **RRT** i quali si sono rivelati estremamente vantaggiosi per ogni tipologia di robot. Alla base del loro funzionamento vi è la visita della mappa attraverso la generazione casuale di un numero prestabilito di nodi, ognuno dei quali rappresenta una configurazione par-

ticolare del robot. Il campionamento casuale di C_{free} fa in modo che la complessità dell'algoritmo non aumenti esponenzialmente all'aumentare delle dimensioni dello stato, rendendo tali algoritmi molto efficienti per modelli con spazi di stato di grandi dimensioni. Inoltre è stato dimostrato che le versioni ottime successive a PRM e RRT, ovvero **PRM*** e **RRT***, godono della proprietà di completezza, in quanto si avvicinano alla soluzione ottima all'aumentare del numero di iterazioni dell'algoritmo[1].

Il problema della Pianificazione di Traiettoria va a toccare diversi ambiti legati al campo ingegneristico. Si pensi per esempio al caso di PRM, in cui si modellizza il problema utilizzando un grafo composto da nodi e archi pesati; in questo caso sarebbe inevitabile l'uso di metodologie legate alla Ricerca Operativa aventi l'obiettivo di calcolo di un cammino ottimo sulla base di una cifra di merito definita.

D'altro canto, il problema non può non riguardare anche il campo della meccanica. Infatti, durante l'esecuzione degli algoritmi, viene richiamata più volte una *Steering Function*, la quale calcola la traiettoria che connette una coppia di nodi. Tale traiettoria dovrà essere percorsa dal robot e per questo dovrà tener conto della sua dinamica. E' necessario dunque uno studio sistematico del modello meccanico che ne descriva il moto e i vincoli. Tale analisi condiziona la struttura degli algoritmi di pianificazione della traiettoria, i quali richiederanno logiche più complesse nel caso di modelli con vincoli non olonomi.

In ultima analisi, viene affrontato anche l'ambito della teoria del controllo, ed in particolare la teoria del Controllo Ottimo. Si dimostrerà infatti come il problema della pianificazione di traiettoria può essere definito come problema di minimizzazione di una funzione di costo. Tale funzione sarà scelta in modo da minimizzare tempo necessario al robot per percorrere la traiettoria e le eccessive sollecitazioni sulle variabili di controllo.

Per l'effettiva implementazione degli algoritmi analizzati si è deciso di limitare il campo a **robot mobili**. In questo modo è stato possibile visualizzare graficamente i risultati ottenuti essendo lo spazio delle configurazioni un ambiente bidimensionale.

In questa tesi verrà analizzata innanzitutto la situazione attuale nell'ambito della pianificazione di traiettoria. Dopodiché si farà riferimento al software utilizzato per l'implementazione pratica degli algoritmi studiati e all'utilizzo di alcune librerie di funzioni legate all'ambito robotico. In seguito verranno riportati i modelli considerati per rappresentare il robot e gli ostacoli nell'ambiente.

Nei capitoli tre e quattro si entrerà nel merito degli algoritmi PRM e RRT, nonché di alcune loro varianti degne di nota. L'obiettivo di questi due capitoli sarà quello di fornire le nozioni necessarie alla comprensione degli algoritmi e delle fondamentali caratteristiche che li differenziano. Per una maggiore comprensio-

ne verranno inoltre riportati gli pseudo-codici degli algoritmi e alcune figure dei risultati finali ottenuti.

Nel capitolo cinque, dopo aver presentato la teoria del Controllo Ottimo da un punto di vista generico, si passerà all'analisi della sua applicazione nel caso specifico della pianificazione di traiettoria. Si studierà il principio alla base dell'algoritmo **Kinodynamic-RRT***[2][3], un'estensione di RRT* per modelli di robot con vincoli anolonomi.

Lo studio di quest'ultimo ha portato ad un quesito: perché per la pianificazione di traiettoria nei sistemi kinodinamici l'algoritmo RRT è solitamente più utilizzato rispetto a PRM? Infatti PRM, pur essendo antecedente a RRT, ha diversi vantaggi rispetto a quest'ultimo (i quali saranno trattati nei rispettivi capitoli).

Dopo aver aver compreso le ragioni di tale preferenza, si è cercata comunque un'alternativa simile all'algoritmo PRM per sistemi di questo genere. Questo ha condotto alla stesura dell'algoritmo **Kinodynamic-PRM***. Tale algoritmo campiona C_{free} esattamente come PRM*, ma a differenza di quest'ultimo, crea le connessioni tra i nodi utilizzando un pianificatore locale basato sul controllo ottimo.

Nell'ultimo capitolo si confronteranno i risultati ottenuti da diverse simulazioni, in modo da analizzare l'influenza dei parametri inseriti dall'utente sul risultato dei diversi algoritmi.

Nell'appendice si può trovare un sintetico manuale utente dedicato all'utilizzo dei codici *Matlab* implementati.

Capitolo 1

Stato dell'arte

In questo capitolo verrà illustrata una panoramica generale degli algoritmi utilizzati nella pianificazione di traiettoria e del software utilizzato per l'implementazione.

La ricerca iniziale si è focalizzata su architetture software già esistenti e disponibili riguardanti proprio gli algoritmi in questione. Tale ricerca ha portato quindi all'analisi di diverse librerie e in questo capitolo verranno esposte le due di particolare interesse.

1.1 Algoritmi

Il problema della pianificazione di traiettoria si può definire come la ricerca di una sequenza opportuna di variabili $u(t)$ con cui controllare il robot affinché esso possa raggiungere autonomamente una configurazione finale avendo come dato il modello generalizzato della sua dinamica e la sua configurazione iniziale.

Nel corso degli ultimi decenni sono sorte diverse classi di algoritmi inerenti a questo argomento. Quelle di maggiore rilevanza sono:

- **Grid-based search algorithms**[4]: Alla base di questa tipologia di algoritmi vi è la discretizzazione dello spazio di stato C, ovvero di tutte le possibili configurazioni attuabili dal robot attraverso una griglia. Ogni cella rappresenta infatti una di queste configurazioni. Qualora il robot si trovi in una di queste celle, esso potrà raggiungere solo una configurazione rappresentata da una delle celle adiacenti. Ovviamente, ad ogni variazione di posizione, viene verificato che la configurazione corrente del robot sia *collision-free*, ovvero non faccia collidere il robot con uno degli ostacoli. Dopo aver creato la griglia, l'obiettivo dell'algoritmo sarà quello di trovare al suo interno un cammino che unisca la configurazione iniziale ad una finale desiderata.

  Come si può ben intuire, tale approccio risulta vantaggioso solo per spazi di stato con poche dimensioni. Infatti il numero di configurazioni e di conseguenza le dimensioni della griglia aumenteranno esponenzialmente rispetto ai gradi di libertà del robot.

- **Artificial Potential Field algorithms**[9]: Il concetto base su cui si fondano gli algoritmi di questa famiglia è quello di creare un campo di forze fittizio che copra l'intero spazio delle configurazioni C. Il suddetto campo di forze sarà quindi definito in modo tale da attrarre il robot nella Regione di Goal e allontanarlo allo stesso tempo dalle regioni occupate dagli ostacoli. In alcuni di questi casi sarà necessario l'utilizzo di tecniche particolari per evitare che il robot si stabilizzi nei punti di minimo locale.

- **Visibility Graph algorithms**[10]: Gli algoritmi di questa classe sfruttano il concetto del cosiddetto *Visibility graph*. I nodi del grafo rappresentano le posizioni nel piano Euclideo. Il grafo verrà creato unendo con dei segmenti i nodi solo nel caso in cui il segmento non attraversi un ostacolo. Per comprendere meglio: si supponga di avere un robot dotato di sensori ottici capaci di rilevare l'ostacolo senza però conoscerne l'intera forma geometrica nello spazio, in tal caso si permetterà al robot di raggiungere solo i nodi visibili, ovvero quelli non oscurati dalla presenza degli ostacoli. Una volta raggiunta la nuova posizione il campo visivo cambierà e si aggiungeranno quindi gli archi come fatto precedentemente. Tramite l'utilizzo di algoritmi alla base della Ricerca Operativa, come l'algoritmo *Dijkstra*, si potrà trovare il cammino desiderato all'interno del grafo così costruito.

- **Reward-based algorithms**[8]: Gli algoritmi di questa classe si basano sulla teoria matematica chiamata MDP (Markov decision process), una tecnica stocastica utilizzata nell'ottimizzazione di un problema. Per la pianificazione di traiettoria si assume che in ogni stato in cui si può trovare il robot ci sia un insieme finito di possibili azioni da compiere ognuna delle quali lo condurrebbe ad uno stato successivo differente. Dopo aver scelto in modo del tutto casuale l'azione da compiere, si assegna una ricompensa positiva qualora venga raggiunto un certo obiettivo o negativa nel caso in cui l'azione porta il robot a collidere con un ostacolo. L'algoritmo quindi pianificherà la traiettoria data dalla migliore sequenza di scelte da compiere, massimizzando la ricompensa totale.

Algoritmi Sampling-based

Negli anni si sono rivelate sempre più convenienti le tecniche basate sulla probabilità, essendo quest'ultime decisamente meno dispendiose in termini computazionali. La classe di algoritmi in questione è chiamata **Sampling-based**[1]. Questi algoritmi si basano sul campionamento stocastico dello spazio delle configurazioni creando in questo modo una rete di nodi e archi chiamata *Roadmap*. Tutti questi algoritmi si strutturano fondamentalmente in due fasi: la prima detta, *Learning*

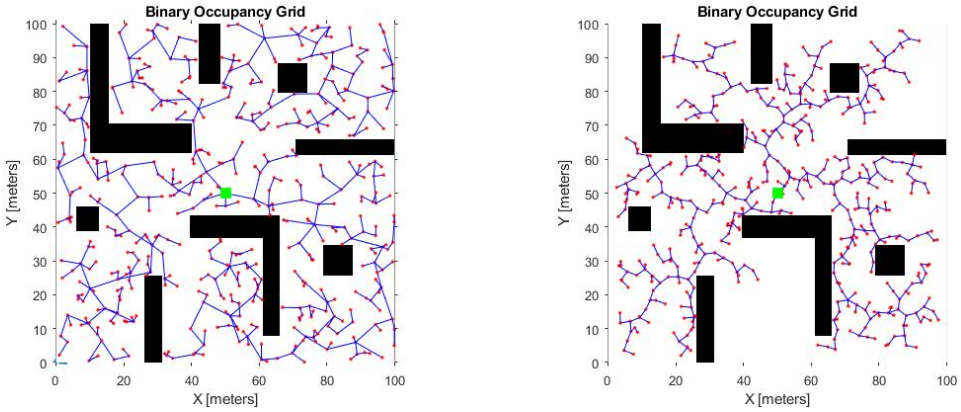
phase, consiste nella fase in cui viene creata la Roadmap generando un numero finito di nodi casualmente in C_{free} e connettendoli tramite archi *collisio-free* secondo logiche che variano in base all'algoritmo particolare. Nella seconda, chiamata *Quering phase*, l'algoritmo calcola il cammino ottimo a partire da un nodo *start* e un nodo *goal* all'interno della RoadMap. In alcuni casi il nodo *goal* può essere sostituito da una regione di goal, ovvero una porzione dello spazio entro la quale deve giungere il robot alla fine del percorso.

Gli algoritmi di questa categoria appartengono essenzialmente a due sottocategorie: **Probabilistic RoadMap (PRM)** e **Rapidly-exploring Random Tree (RRT)**. La differenza tra le due categorie riguarda solo la fase fi *Learning*. Negli algoritmi PRM il grafo viene creato aggiungendo progressivamente nodi alla RoadMap a connettendoli tra loro in un secondo momento in base alla distanza che li separa; quello che si verifica è che il grafo potrà essere composto da diverse componenti connesse con la conseguenza che a partire dal nodo *start*, appartenente ad una componente del grafo, i nodi appartenenti a differenti componenti non potranno essere raggiunti dal robot. Negli algoritmi RRT è più opportuno parlare di Albero piuttosto che di grafo. Infatti questi algoritmi creano una struttura unica radicata nel nodo *start* e composta da nodi e archi. A ogni iterazione verrà generato un nodo con probabilità uniforme all'interno dello spazio libero, questo detterà la direzione di propagazione dell'albero nella singola iterazione del processo.

Gli algoritmi PRM, sebbene permettano di visitare in breve tempo l'intero spazio libero, hanno lo svantaggio di essere poco pratici nei casi in cui il movimento del robot sia limitato da vincoli anolonomi. In questi casi infatti il Pianificatore Locale (funzione che connette due singoli nodi) non si limiterà a connettere i nodi con dei segmenti, ma calcolerà traiettorie complesse attuabili dal robot; di conseguenza sarà necessario definire con chiarezza il verso di percorrenza, il quale condizionerà l'intera traiettoria. Per questa ragione diventa poco vantaggioso avere diverse componenti connesse separate che potrebbero unirsi con il progredire delle iterazioni perchè in tal caso si potrebbe fare confusione sui versi di percorrenza.

Al contrario gli algoritmi RRT si prestano molto a questo genere di modelli dato che l'albero delle traiettorie si ramifica partendo da un unico nodo radice e non crea ambiguità sui versi di percorrenza.

A titolo di esempio in figura 1.1 vengono riportate due reti di traiettorie create dagli algoritmi PRM e RRT rispettivamente. Gli ostacoli vengono rappresentati dalle regioni in nero, il nodo *start* dal quadrato verde, le traiettorie locali dai segmenti blue e i nodi in rosso.



(a) PRM (b) RRT

Figura 1.1: PRM e RRT

1.2 Software

Gli algoritmi sono stati implementati in ambiente *Matlab*, che insieme a *Simulink* è il software più utilizzato e conosciuto della multinazionale *MathWorks*. Il software prevede un linguaggio di programmazione proprio ed è usato in svariati ambiti differenti tra cui la robotica e i sistemi di controllo. Il linguaggio Matlab è un linguaggio semplice e di immediata comprensione, volto soprattutto a calcoli matriciali complessi e alla trattazione grafica di diverse tipologie di dati. Sono disponibili una vasta quantità di librerie contenenti funzioni descritte in modo completo e comprensibile nel sito ufficiale della Mathworks.

Di particolare interesse e degne di nota sono due librerie riguardanti proprio l'ambito robotico: **Robotics Toolbox**[6] del professore Australiano **Peter Corke**, e **Robotics System Toolbox**[7] resa disponibile recentemente nella nuova versione *Matlab R2018a*. Lo studio iniziale si è focalizzato sull'analisi di queste due librerie di funzioni con l'obiettivo di verificare la loro flessibilità a future modifiche in vista dell'implementazione delle varianti degli algoritmi base.

1.2.1 Robotics Toolbox, Peter Corke

Robotics Toolbox[6], disponibile sul sito ufficiale del professore e ingegnere Australiano Peter Corke, contiene svariate funzioni riguardanti la simulazione di modelli robotici di ogni genere e diversi algoritmi tra cui alcuni per la pianificazione di traiettoria. In particolare, rispetto proprio a quest'ultimo ambito, gli algoritmi disponibili sono Bug2, Dstar, Lattice, PRM e RRT. L'attenzione in particolare è stata rivolta agli ultimi due, essendo parte della classe di nostro interesse:

- **PRM**: Per l'utilizzo di questo algoritmo si considera il robot come un semplice punto nel piano, ignorando qualsiasi limite cinematico o dinamico. L'algoritmo quindi genera casualmente nodi che rappresentano semplici coppie di coordinate $[x, y]^T$.

  Tramite il comando `prm = PRM(Mymap)` è possibile creare un Probabilistic Roadmap Navigation object, ovvero un oggetto che, basandosi sulla tabella di valori binari Mymap, permette l'utilizzo di diversi metodi al suo interno. Definendo prima il numero di nodi finale N è possibile generare la RoadMap con il metodo `prm.plan('npoints', N))`, mentre per il calcolo effettivo del percorso più economico dal nodo start a goal si dovrà utilizzare il metodo `prm.query(start,goal)`.

- **RRT**: Per quanto riguarda l'algoritmo RRT, la libreria di Peter Corke considera un modello di robot più realistico. Il robot non sarà rappresentato da un semplice punto adimensionale privo di qualsiasi limite cinematico e dinamico, ma sarà definito dall'oggetto di tipo Vehicle, che comprende diverse tipologie di modelli. In questo caso è stato scelto di considerare il robot come un oggetto di tipo Bicycle tramite l'utilizzo del comando `Vehicle = Bicycle('steermax', 1.2, 'speedmax', 10, 'dt', dt)`. Come si può notare, la funzione che genera l'oggetto di tipo Bicycle prevede l'ingresso di diversi argomenti opzionali, tra i quali: l'angolo di sterzata massimo, la velocità massima in valore assoluto raggiungibile dal mezzo, l'intervallo di tempo che trascorre tra un nodo e il successivo della traiettoria. I nodi che andranno a comporre l'albero delle traiettorie saranno essenzialmente dei vettori di tre componenti $[x, y, \theta]$ e verranno generati casualmente nello spazio delle tre dimensioni considerando i limiti imposti dall'ambiente e dal robot stesso.

  Per prima cosa quindi l'algoritmo genera un nodo casuale q_{Rand}. Dopo aver trovato $q_{Nearest}$, ovvero il nodo più vicino a q_{Rand}, l'algoritmo genera in modo casuale una serie (50 di default) di traiettorie locali realizzabili dal robot e aventi come nodo iniziale $q_{Nearest}$. Tra queste traiettorie verrà eseguito un doppio controllo: il primo sarà fatto all'interno della singola traiettoria con il fine di trovare il nodo più prossimo a q_{Rand}, il secondo avrà lo scopo di trovare quale traiettoria genera il nodo più vicino a q_{Rand}. Il calcolo della distanza viene eseguito per le tre variabili e quindi sarà descritto dall'equazione:

  $$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (\theta_1 - \theta_0)^2}$$

  Ottenuta la traiettoria locale che porterebbe il robot alla distanza minima da q_{Rand}, la aggiunge all'albero e itera il procedimento per N volte.

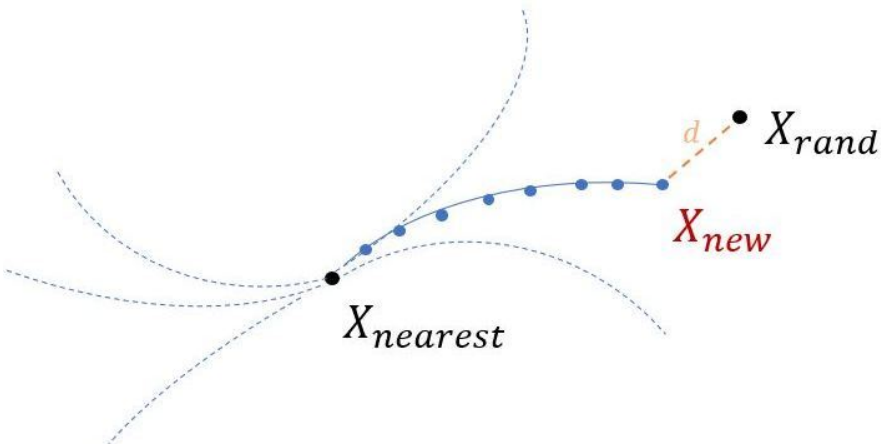


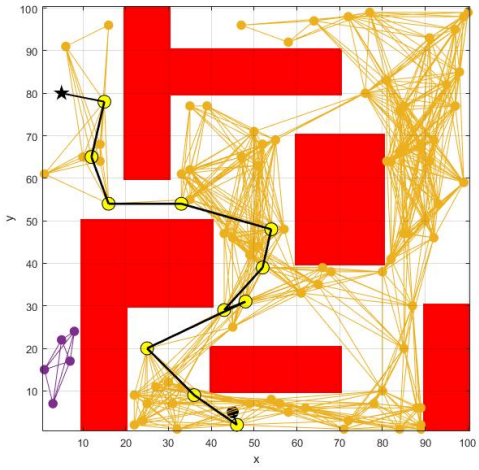
Figura 1.2: RRT Traiettorie Locali

1.2.2 Robotics System Toolbox, MathWorks

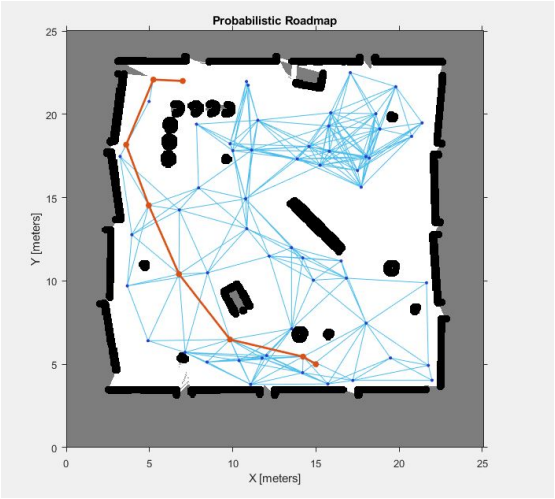
Recentemente la MathWorks ha reso disponibile il **Robotics System Toolbox**[7], ovvero una libreria che permette l'utilizzo di algoritmi e l'interfacciamento a piattaforme hardware legate a robot di terra, robot aerei o robot manipolatori. Questo toolbox permette anche la comunicazione con l'ambiente ROS, sempre più utilizzato nell'ambito della robotica. Purtroppo però sotto il punto di vista della pianificazione di traiettoria, esso offre solo l'algoritmo Probabilistic RoadMap(**PRM**). Sul sito della Mathworks, ed in particolare nella sezione dedicata a questo toolbox, vi sono descritte le funzioni e le istruzioni per il loro utilizzo. In particolare, alla voce **Ground Vehicle Algorithms** si può trovare il capitolo intitolato *Path Planning in Environments of Different Complexity* in cui viene illustrato l'algoritmo PRM e la definizione delle classi di oggetti che ne orbitano attorno.

Anche in questo caso però l'architettura del codice si presta poco a modifiche, essendo ricca di richiami a funzioni molto specifiche interne alla libreria ed è stata quindi abbandonata rapidamente, non essendo adatta allo scopo della tesi.

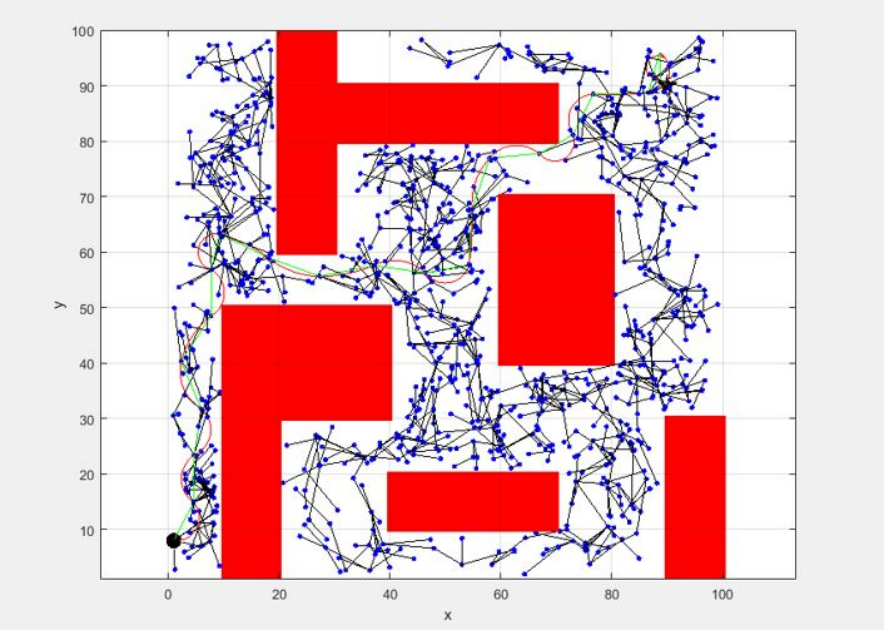
L'analisi del codice però non è stata infruttuosa, infatti, è stato deciso di ereditarne i metodi di definizione della mappa dell'ambiente tramite il comando `robotics.BinaryOccupancyGrid(simpleMap, res)` che crea l'oggetto **BinaryOccupancyGrid**.



(a) PRM Toolbox di Peter Corke



(b) PRM in Robotics System Toolbox



(c) RRT Toolbox di Peter Corke

Capitolo 2

Modello del robot e dell'ambiente circostante

In questo capitolo verranno fornite le informazioni riguardanti i modelli utilizzati per il robot e le metodologie con cui viene rappresentato l'ambiente in cui esso si muove. Per comprendere meglio la scelta adottata nella rappresentazione degli ostacoli verranno descritte le due funzioni fondamentali che la riguardano, ovvero la funzione *SampleFree*, che genera un nodo nello spazio libero, e la funzione *CollisionFree* la quale verifica se una traiettoria non collide con un ostacolo. Si è ritenuto opportuno entrare nel dettaglio di queste funzioni in quanto verranno poi richiamate in tutti gli algoritmi descritti nei capitoli successivi.

2.1 Modello del robot

L'intero lavoro di questo progetto di tesi è stato svolto per passi. Per prima cosa si è voluto analizzare il funzionamento e la correttezza degli algoritmi implementati. In secondo luogo ci si è soffermati sulle modifiche da effettuare agli algoritmi per adattarli ad un modello meccanico del robot più realistico. In questo senso inizialmente si è scelto un modello estremamente semplificato definendo lo stato del robot con il vettore posizione $[x, y]^T$. Inizialmente quindi non sono stati considerati i vincoli dettati dalla meccanica del robot ma ci si è focalizzati sulle strategie da adottare per la stesura dei codici.

Dopo aver verificato l'esatto funzionamento dei codici implementati, si è deciso di aggiungere due variabili allo stato del robot in modo da poterlo rappresentare secondo il modello dell'**Uniciclo**[3][17]. Le equazioni che descrivono la cinematica di tale modello sono:

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = \omega \\ \dot{v} = a \end{cases}$$

Dove le variabili (x, y, θ) rappresentano la posizione lungo gli assi X e Y del sistema di riferimento assoluto e l'orientamento del robot; mentre v, ω e a rappresentano rispettivamente la velocità longitudinale, la velocità angolare e l'accelerazione.

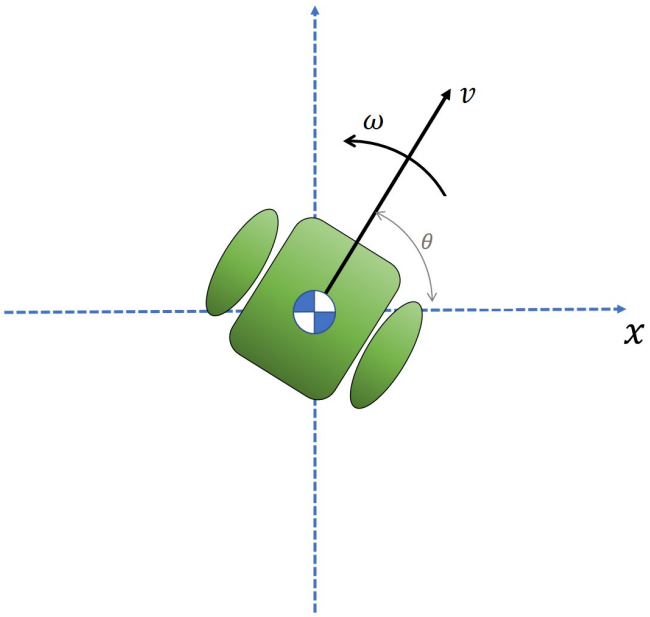


Figura 2.1: Modello Uniciclo

Come si può facilmente notare, si tratta di un sistema non lineare, e questo comporta una serie di complicazioni. Verrà descritto nei prossimi capitoli il adottato per la linearizzazione.

2.2 Modello dell'ambiente e definizione ostacoli

Il robot in questione appartiene alla categoria dei robot mobili e in particolare si assume possa spostarsi in un ambiente terrestre. Si ipotizza inoltre l'esistenza di ostacoli di geometria e localizzazione nota. Si è cercata quindi una struttura che potesse modellizzare l'ambiente nel modo più preciso possibile e che fosse allo stesso tempo facilmente modificabile in base al caso specifico.

Inizialmente si è pensato di adottare una semplice struttura chiamata `Obstacles`, definita a sua volta da due attributi: `centre`, ovvero il vettore $[x_c, y_c]$ che rappresenta la posizione dell'ostacolo i-esimo nel piano e `radius`, che ne rappresenta il raggio massimo. La funzione che verifica la collisione con gli ostacoli avrà il compito di sondare l'intera struttura `Obstacles` per verificare che il nodo di interesse abbia una distanza maggiore del raggio i-esimo da ogni centro i-esimo. Essenzialmente vi sono due grossi svantaggi derivanti dall'utilizzo di questa metodologia:

1. Per prima cosa si deve assumere molto probabile il caso in cui gli ostacoli non sono di sezione circolare, ma di svariate forme geometriche di diversa complessità. Per esempio, nel caso in cui si modellizzasse in questo modo un ostacolo descritto da un poligono, sarebbe necessario utilizzare come diametro della circonferenza la distanza massima tra due punti all'interno del poligono. In questo modo però, specialmente per determinate figure geometriche in cui una dimensione predomina sull'altra, si otterrebbe un ostacolo avente un'area di occupazione di gran lunga più elevata di quella dell'ostacolo reale.

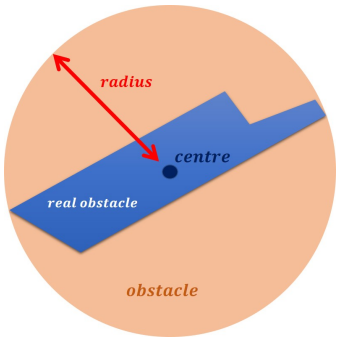


Figura 2.2: Rappresentazione Ostacolo

   Come si può notare in figura 2.2, gran parte dell'area occupata dal modello dell'ostacolo verrebbe scartata anche se non fosse appartenente all'ostacolo reale, limitando i movimenti del robot inutilmente.

2. Come secondo svantaggio vi è il fatto che, specialmente per casi in cui il numero di ostacoli è elevato, il calcolo della collisione risulta leggermente più oneroso. Infatti, all'interno di una singola iterazione negli algoritmi, capita di dover richiamare diverse volte la funzione `CollisionFree` la quale, passando in rassegna l'intera struttura degli ostacoli, verifica che il nodo in questione o l'insieme dei nodi di una traiettoria, sia ad una distanza adeguata da ognuno di essi.

La soluzione più conveniente è la definizione del modello Ambiente tramite l'utilizzo di una `OccupancyGrid`, ovvero di una struttura oggetto che permetta di

suddividere l'ambiente in celle e assegni ad ognuna di esse un valore binario. In questo modo si può rappresentare le aree occupate dagli ostacoli come l'insieme delle celle di valore 1, mentre lo spazio libero sarà rappresentato dall'insieme delle celle di valore 0. Come accennato nel Capitolo 1, il **Robotics System Toolbox**[7] contiene al suo interno un'ampia sezione dedicata alla definizione tramite `OccupancyGrid` dell'ambiente.

L'oggetto `OccupancyGrid` viene creato sulla base di una tabella di valori binari data come argomento ad una specifica funzione di *Matlab*. La tabella rappresenta la mappa dell'intero ambiente suddiviso in celle; ovviamente più il numero di celle è elevato più la risoluzione sarà alta. In base alle esigenze dettate dalla complessità geometrica degli ostacoli, si può suddividere il piano con un numero arbitrario di celle. Per fare questo bisogna definire il numero di righe e colonne della tabella e inserire come argomento in ingresso alla funzione il valore `cellDim`, il quale definisce la lunghezza del lato di una singola cella. In questo caso specifico è stato scelto di creare una tabella di 10^4 righe e 10^4 colonne stabilendo la lunghezza del lato di ogni singola cella pari a 0.01 m. Così facendo si ottiene una mappa di 100 m di larghezza e 100 m di larghezza.

La tabella di valori binari è stata definita tramite il comando `im2bw(Immagine, 0.9)`, che converte l'immagine importata in precedenza in una tabella, assegnando ad ogni pixel il valore 0 nel caso in cui l'intensità del pixel sia superiore alla soglia indicata, mentre assegna 1 qualora sia al di sotto della soglia. In questo modo l'ambiente potrà essere definito semplicemente importando un'immagine in formato `jpg`.

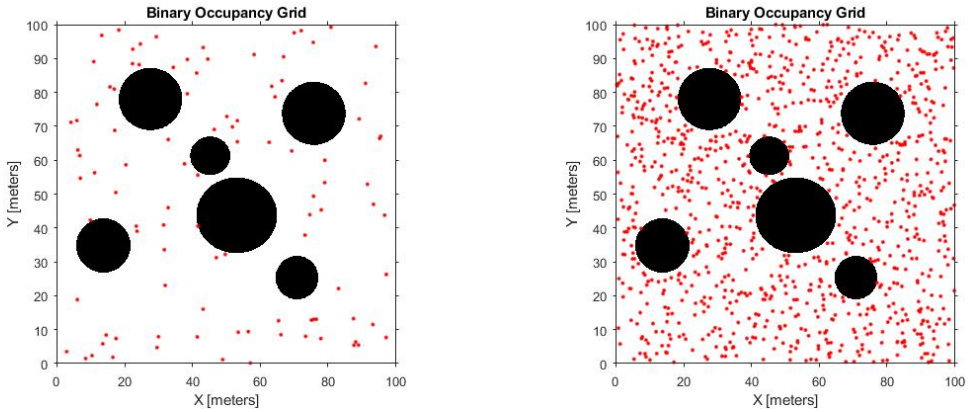
Tramite l'uso del software *Paint*, presente nel sistema operativo *Windows 10*, è stato possibile disegnare alcune mappe contenenti ostacoli. Le mappe in seguito sono state catalogate per complessità e utilizzate nelle diverse simulazioni con lo scopo di analizzare l'efficienza dei vari algoritmi nei diversi ambienti.

2.3 Funzione SampleFree

Gli algoritmi prevedono l'iterazione di una sequenza di comandi principali, primo dei quali è quello riguardante la generazione di una configurazione (nodo) con probabilità uniforme nello spazio di interesse. La funzione `SampleFree` altro non fa che questo.

Per prima cosa genera il vettore che descrive la configurazione del robot dando dei valori casuali alle singole componenti. In seguito poi viene verificata la condizione di appartenenza alla regione libera tramite la funzione `CollisionFree`. La funzione estrae qualsiasi posizione nello spazio delle configurazioni con uguale probabilità; per questa ragione a volte può capitare che non venga generato alcun nodo nella regione di Goal. Questo si verifica in particolare nei casi in cui il numero di iterazioni scelto è basso e la regione di Goal ha un'area molto limitata.

Per ovviare a questo problema si è deciso di implementare una variante chiamata `SampleFree_toGoal`, la quale genera un nodo all'interno della regione di Goal ogni **n** iterazioni, dove **n** è un parametro arbitrario da passare come argomento alla funzione.



(a) 100 nodi (b) 1000 nodi

Figura 2.3: Copertura tramite generazione con probabilità uniforme di C_{free}

2.4 Funzione CollisionFree

All'interno degli algoritmi capita spesso di dover verificare che un nodo non cada all'interno dell'area di un ostacolo, è necessario quindi trovare un metodo estremamente preciso che lo possa fare nel modo più rapido possibile.

La definizione dell'oggetto `OccupancyGrid` offre la possibilità di utilizzare alcuni metodi della classe capaci di verificare se un punto di coordinate passate come argomento sia collocato all'interno di una cella libera (0) o occupata (1).

Il metodo usato per tener conto dello spessore del robot è `inflate(map)`. Tale metodo permette, una volta inizializzata la mappa e quindi le configurazioni degli ostacoli, di "allargare" le regioni occupate dagli ostacoli della misura del raggio del robot. Per questa ragione non si dovrà fare caso al fatto che talvolta le traiettorie generate sembrino portare il robot a collidere con gli ostacoli.

CollisionFree Simple

Nella teoria, gli algoritmi scelti per l'implementazione prevedono l'uso della funzione `CollisionFree` per un'intera traiettoria composta da un insieme di punti.

La scelta più immediata sarebbe quella di applicare un ciclo **for** che controlli ogni singolo nodo della traiettoria. Per quanto possa essere rapido e di facile implementazione questo metodo però non gode di un'assoluta precisione la quale dipende dal periodo di campionamento con cui si decide di creare la traiettoria. I nodi saranno infatti ad una distanza temporale pari a **dt** e qualora questo parametro non fosse adeguatamente basso, la funzione rischierebbe di dichiarare libera la connessione tra due nodi nel caso in cui un ostacolo di diametro inferiore alla loro distanza si trovasse in mezzo.

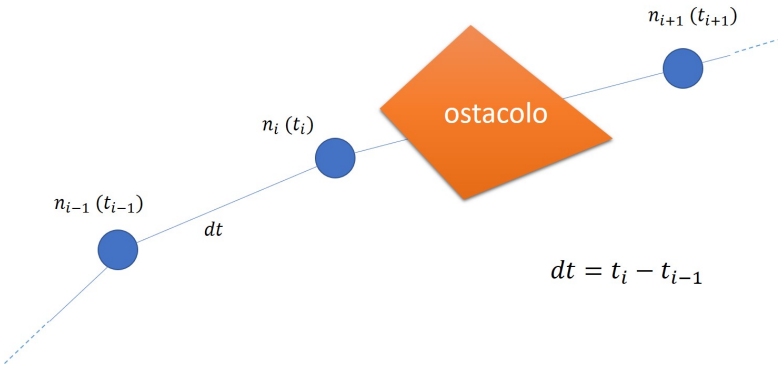


Figura 2.4: caso in cui CollisionFree non rivela l'ostacolo

Al fine di aumentare la precisione della rilevazione e renderla indipendente dalla scelta del parametro **dt**, si è deciso di implementare una variante che sfrutti il metodo di interpolazione **Spline**. L'obiettivo sarà dunque quello di calcolare una funzione interpolante che connetta i punti della traiettoria e, valutandola con passo di campionamento regolare, verificarne l'attuabilità.

CollisionFree Spline

Il metodo **Spline**[5] (smooth path line), nasce con lo scopo di migliorare l'interpolazione cubica tra nodi evitando le discontinuità nel grafico delle accelerazioni nei punti intermedi.

Spline è la funzione interpolante con la minima curvatura, creata sulla base di opportune condizioni di continuità delle derivate e di passaggio per i suddetti nodi.

Ipotizziamo di avere un robot che si muove lungo una sola dimensione. Lo stato del robot potrà quindi essere descritto dalla sola variabile $\mathbf{q(t)}$ per esempio. Assumiamo ora di avere **n** stati differenti $[q_0,\ q_1,\ q_2,\ \cdots,\ q_n]$ relativi agli **n** istanti $[t_0,\ t_1,\ t_2,\ \cdots,\ t_n]$ e di doverli interpolare con lo scopo di trovare una traiettoria unica descritta nel tempo. Il metodo di interpolazione polinomiale cubica ha delle limitazioni dovute alle discontinuità nel grafico dell'accelerazione mentre il metodo

Spline, mantenendo comunque una dipendenza cubica dal tempo, considera anche le condizioni di continuità nelle derivate dello stato.

L'obiettivo è quello di calcolare gli **n-1** polinomi che connettano le coppie di nodi consecutivi e aventi questa forma:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

Ogni polinomio contiene quattro parametri incogniti (a_0, a_1, a_2, a_3). In totale quindi vi saranno $4(n-1)$ incognite. Le condizioni da imporre sono le seguenti:

- $2(n-1)$ condizioni che impongono il passaggio per il punto iniziale e finale della singola traiettoria cubica (basta valutare i singoli polinomi negli estremi);
- $n-2$ condizioni di continuità delle velocità nei punti intermedi;
- $n-2$ condizioni di continuità delle accelerazioni nei punti intermedi;

A questo punto, affinchè il sistema non sia indeterminato e la soluzione ottenibile sia unica, serve aggiungere ulteriori due condizioni, infatti:

$$4(n-1) - 2(n-1) - 2(n-2) = 2$$

Gli ultimi due gradi di libertà si possono bloccare imponendo le condizioni per la velocità e l'accelerazione iniziale.

In ambiente Matlab si può trovare la funzione `spline(X,Y)`, la quale crea l'interpolante tramite il metodo Spline dei valori del vettore Y associati ai valori del vettore X.

A titolo d'esempio, se avessimo un vettore di valori:

$$Y = 2,\ 4,\ 3,\ 3,\ 5,\ 6,\ 5,\ 4,\ 5,\ 4,\ 1,\ 2$$

negli istanti di tempo:

$$X = 0,\ 2,\ 3,\ 5.5,\ 10,\ 11,\ 12,\ 15,\ 20,\ 20.5,\ 21,\ 23$$

chiamando la funzione `PP=spline(X,Y)` otterremmo la funzione Spline che li interpola come mostrato in *figura2.4*.

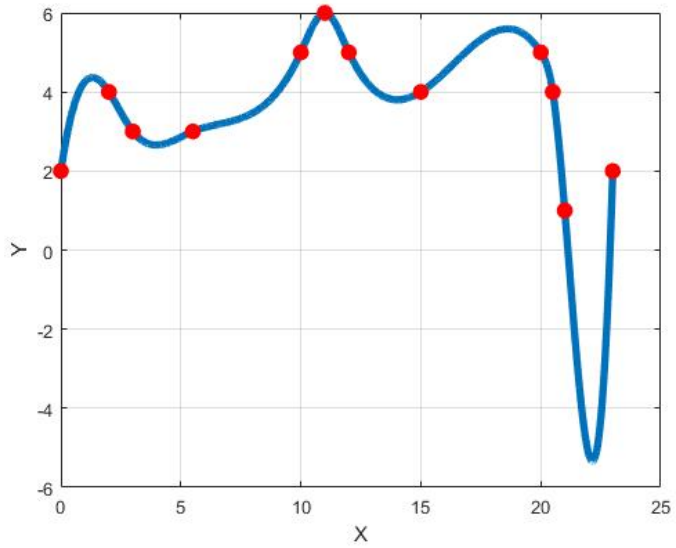


Figura 2.5: Spline

Nel nostro caso, andranno interpolati i singoli punti della traiettoria e quindi si applicherà questo metodo singolarmente per il vettore delle coordinate x e quello delle y dei vari nodi.

I passi fondamentali della funzione `CollisionFree_Spline` risultante sono:

1. Parametrizzazione della curva. Ogni nodo sarà identificato da un singolo valore del vettore **Ascissa curvilinea**

$$S = [\ \Delta s_0;\ \Delta s_0 + \ \Delta s_1;\ \Delta s_0 + \ \Delta s_1 + \ \Delta s_2; \cdots ; L]$$

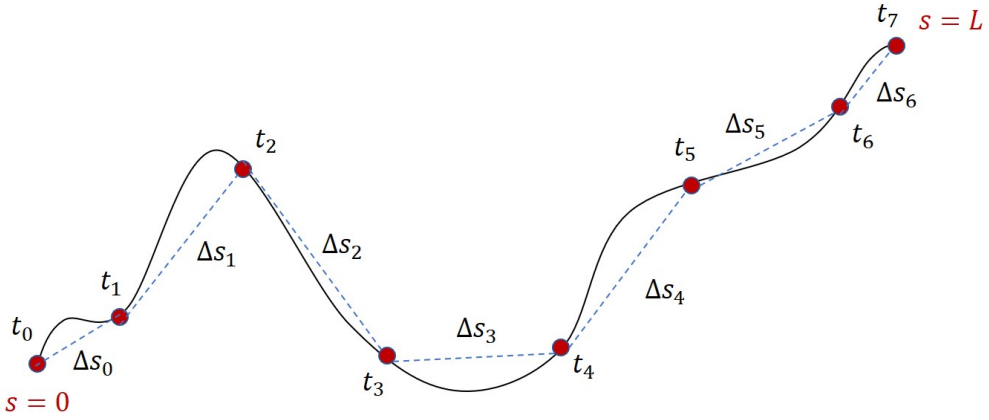


Figura 2.6: Definizione Ascissa curvilinea S

in cui i valori Δs_i indicano la distanza euclidea tra il nodo n_i e il nodo precedente n_{i-1}, mentre L è l'approssimazione dello spazio percorso dal robot

partendo dal nodo iniziale a quello finale. In questo modo la posizione del robot nello spazio sarà identificata da un'ascissa e un'ordinata entrambe funzioni dell'ascissa curvilinea:

$$t = [\ t_0,\ t_1,\ t_2,\ \cdots,\ T\ ]$$
$$S = [\ s_0,\ s_1,\ s_2,\ \cdots,\ L\ ]$$
$$x = [\ x_0,\ x_1,\ x_2,\ \cdots,\ x_f\ ]$$
$$y = [\ y_0,\ y_1,\ y_2,\ \cdots,\ y_f\ ]$$

2. Campionamento con un passo **ds** della traiettoria linearizzata. Questo passaggio permette di dividere la funzione ascissa curvilinea in frammenti di uguale distanza creando il vettore:

$$s' = [\ 0;\ ds;\ 2ds;\ 3ds;\ \cdots;\ L\ ]$$

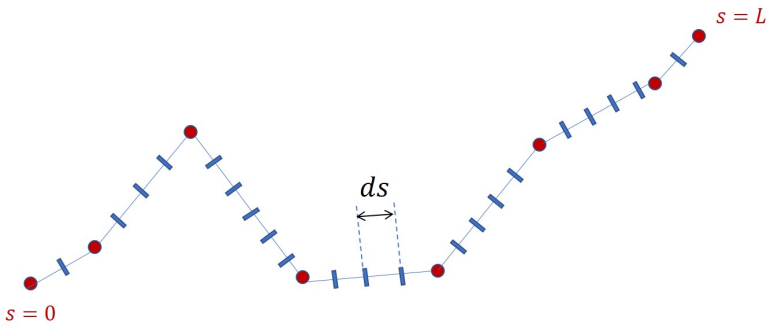


Figura 2.7: Campionamento curva linearizzata

3. Calcolo delle curva Spline rispetto alle singole funzioni $x(S)$ e $y(S)$ tramite i comandi `xsp = spline(s, x)` e `ysp = spline(s, y)`.

4. Controllo di collisione effettuato valutando tutte le coppie $[xsp(s'_i), ysp(s'_i)]$ che rappresentano in sostanza le coordinate di tutti i punti intermedi distanti **ds** tra loro.

In questo modo la precisione del controllo dipenderà dal parametro **ds** e non più dal parametro **dt** con cui vengono campionati gli **n** nodi della traiettoria iniziale.

Capitolo 3

Probabilistic RoadMap

Nella categoria degli algoritmi *Sampling-based*, **PRM**[1] è uno dei primi ad essere stato applicato alla robotica. Come suggerisce il nome, l'algoritmo si basa su un concetto fondamentale ma rivoluzionario in questo ambito: la randomizzazione della mappa che descrive l'ambiente. PRM infatti cerca la traiettoria all'interno di una rete di connessioni (*RoadMap*) tra nodi generati con probabilità uniforme nell'intero spazio libero C_{free}.

Essenzialmente l'algoritmo si suddivide in due fasi: la fase di *Learning* e la fase di *Quering*. Nella prima parte viene creata la RoadMap, mentre nella seconda fase viene trovata la traiettoria a costo minimo che connette il nodo iniziale a quello finale.

3.1 Fase di Learning

In questa fase si genera la **RoadMap**, ovvero la rete di nodi e archi in cui i nodi rappresentano le diverse configurazioni attuabili dal robot e appartenenti allo spazio libero C_{free}, mentre gli archi sono le traiettorie realizzabili che li connettono.

Le connessioni tra i nodi sono calcolate da un **Pianificatore Locale**, ovvero un algoritmo specifico che, in base alle diverse esigenze e casistiche, calcola la sequenza di nodi intermedi e ne verifica l'appartenenza allo spazio libero C_{free}.

La RoadMap essenzialmente è un Grafo $G = (V, E)$ in cui V rappresenta l'insieme dei nodi (o vertici) e E l'insieme degli archi.

Per prima cosa l'algoritmo prevede la generazione di una configurazione casuale chiamata q_{Rand} all'interno di C_{free}. Il nodo verrà quindi aggiunto all'insieme V inizialmente vuoto.

A questo punto, l'algoritmo calcola l'insieme dei nodi appartenenti a V più vicini a q_{Rand} selezionandoli in base a diverse strategie, che possono essere dettate

dalla distanza euclidea, dalla minimizzazione di un'opportuna cifra di merito o da un numero predefinito di nodi da selezionare.

A partire dal nodo meno distante, l'algoritmo connette tutti i nodi del suddetto insieme a q_{Rand} solo nel caso in cui siano verificate le seguenti due condizioni: il Pianificatore Locale dichiara libera da collisioni la traiettoria dei due nodi e i due nodi non appartengono alla stessa componente connessa.

L'arco corrispondente alla connessione dei due nodi verrà aggiunto all'insieme E e questo implica l'unione di due differenti componenti connesse della *RoadMap*. Quello che accade dunque è la nascita di un'unica componente connessa risultante dall'unione delle due (figura 3.2).

La situazione ottimale sarebbe quella di terminare la prima fase avendo ottenuto una RoadMap composta da un'unica componente, in questo modo tutti i nodi sarebbero raggiungibili da un qualunque nodo iniziale appartenente a V.

3.2 Fase di Quering

In questa fase vengono aggiunti il nodo q_{start} e q_{goal} all'insieme V dei nodi della RoadMap.

Come veniva fatto per un qualsiasi nodo q_{rand}, essi verranno quindi connessi ai nodi più vicini di V.

A questo punto si potrà trovare la traiettoria che connette lo stato iniziale a quello finale solo nel caso in cui i due nodi q_{start} e q_{goal} appartengano alla stessa componente connessa.

Nel caso specifico dei codici implementati per questo lavoro, si è deciso di annettere q_{start} all'insieme V già dalla prima iterazione. Per quanto riguarda lo stato finale invece, non si definisce un unico nodo q_{goal}, ma una regione all'interno dello spazio di configurazione composta da nodi.

Si può quindi dire che l'algoritmo ha trovato una soluzione al problema di pianificazione nel caso in cui riesca a connettere il nodo q_{start} ad un qualsiasi nodo appartenente alla Regione di Goal.

3.3 Algoritmo PRM

L'**Algoritmo 1** rappresenta la fase di *Learning*. In questo caso gli archi rappresentano le connessioni dei nodi come dei semplici segmenti dritti. Per questa ragione il Pianificatore Locale, una volta dichiarata libera la connessione trai due nodi, aggiunge all'insieme degli archi E il segmento in entrambi i versi di percorrenza.

In questo primo semplice caso, si assume che lo stato del robot sia descritto semplicemente dal vettore $[x, y]^T$, e che quindi non sia rilevante il verso di percorrenza.

Nei casi successivi invece si considererà nello stato anche le componenti cartesiane della velocità del robot, questo comporterà delle modifiche nella stesura dell'algoritmo.

Inoltre vi è da precisare che in questo caso si assume che i costi delle traiettorie, ovvero i valori con cui vengono pesati gli archi, sono descritti dalla distanza euclidea tra i due nodi estremi $\| q_a - q_b \|$.

Algorithm 1 PRM: Fase di Learning

1: $V \leftarrow q_{start}; E \leftarrow \emptyset;$
2: **for** $i = 1$ to **maxNodes do**
3: $q_{rand} \leftarrow$ `SampleFree`;
4: $U \leftarrow$ `Near`$(G = (V, G))$, q_{rand}, **discRadius**)
5: $V \leftarrow V \cup \{q_{rand}\}$;
6: **for all** $u \in U$ *per ordine crescente di* $\| u - q_{rand} \|$ **do**
7: **if** q_{rand} *e* u *non sono nella stessa componente connessa di* $G = (V, E)$ **then**
8: **if** `CollisionFree(` q_{rand} `,u)` **then**
9: $E \leftarrow E \cup \{(q_{rand}, u), (u, q_{rand})\}$
10: **end if**
11: **end if**
12: **end for**
13: **end for**
14: **return** $G = (V, E)$;

- **Parametri**: i dati necessari alla compilazione dell'algoritmo sono i seguenti:
  - q_{start}: si tratta del vettore composto da **m** componenti che rappresenta lo stato iniziale del robot, dove **m** è la dimensione dello spazio di stato considerato.
  - `maxNodes`: è il numero di nodi che andranno a formare la RoadMap alla fine della compilazione.
  - `discRadius`: dopo la generazione del nodo q_{rand}, l'algoritmo trova tutti i nodi contenuti in V che abbiano una distanza da q_{rand} minore di discRadius.
- **Funzioni**:
  - `SampleFree`: questa funzione genera una configurazione casuale nello spazio libero. In questo caso quindi genera semplicemente una coppia di valori $[x, y]^T$ che rappresentano un punto nel piano. E' necessario

dunque che la funzione riceva come ingresso l'oggetto `OccupancyGrid` che descrive la struttura dell'ambiente e la posizione degli ostacoli.

Come descritto nel Capitolo 2, nel codice implementato sarà possibile utilizzare una versione modificata, in cui la funzione `SampleFree` genera un configurazione q_{rand} all'interno della regione di Goal con una frequenza regolare dettata da **n**, dove **n** è un parametro arbitrario scelto dall'utente.

- `Near`: calcola la distanza tra il nodo appena generato q_{rand} e ogni nodo appartenente all'insieme V, dopodiché salva nell'insieme U tutti i nodi aventi distanza inferiore al parametro *discRadius* passato come parametro di ingresso.
- `CollisionFree`: come ampiamente trattato nel Capitolo 2, *CollisionFree* permette di controllare che non vi siano collisioni con ostacoli previste per la traiettoria che connette i due nodi passati come argomenti. Ovviamente anche in questo caso la funzione deve poter conoscere l'ambiente e la posizione degli ostacoli, quindi sarà necessario che riceva come argomento ulteriore l'oggetto `OccupancyGrid`. La procedura con cui questa funzione esegue il suddetto controllo non viene qui riportata, essendo già ampiamente descritta nel Capitolo 2.

- **Algoritmo**: La sequenza di comandi descritta nello pseudo codice viene qui tradotta con il fine di chiarire meglio i passaggi dell'Algoritmo 1:

  1. Inizializzazione del grafo $G = (V, E)$, definendo V come l'insieme contenente il solo nodo q_{start}, configurazione iniziale del robot, e E come l'insieme vuoto.
  2. La funzione `SampleFree` genera il nodo q_{rand} come punto casuale nello spazio libero.
  3. La funzione `Near` aggiunge all'insieme U tutti i nodi appartenenti all'area circolare di centro q_{rand} e raggio *discRadius*.
  4. q_{rand} viene aggiunto all'insieme V.
  5. Per ordine crescente di distanza, tutti i nodi dell'insieme U vengono connessi a q_{rand} solo nel caso in cui la connessione è realizzabile (non prevede collisioni) e i due nodi non appartengono alla stessa componente connessa.
  6. Qualora i due nodi possono essere connessi, si aggiunge ad E l'arco che li connette con entrambi i versi di percorrenza.
  7. L'algoritmo restituisce il grago G rappresentante l'intera RoadMap.

3.3.1 Implementazione

Sulla base dell'algoritmo appena descritto è stato implementato il codice in ambiente *Matlab*.

L'algoritmo prevede la creazione progressiva di una RoadMap e la ricerca del cammino migliore che connetta lo stato iniziale ad uno appartenente alla regione Goal. In *Matlab* vi è una libreria dedicata alla costruzione di grafi e alla ricerca all'interno di questi di cammini ottimi. Nel caso particolare di PRM l'implementazione basata su tali funzioni risulterebbe sicuramente più immediata ed efficiente, ma anche svantaggiosa per l'implementazione delle varianti successive. Per questo è stato deciso di rappresentare la RoadMap con una variabile di tipo *struct*, adatta all'aggiunta futura di altri attributi.

Learning

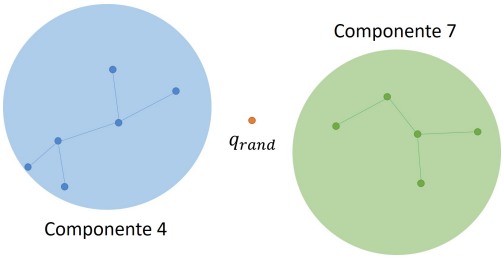
La strategia adottata per la fase di Learning è quella di creare una variabile di tipo *struct* chiamata `Nodes` che rappresenti l'insieme dei nodi della RoadMap e delle loro connessioni. La struttura è composta quindi dall'elenco dei nodi i quali sono a loro volta caratterizzati da diversi parametri e strutture interne. Ogni nodo infatti sarà determinato da un preciso stato che ne identifica la posizione nel piano cartesiano, da un parametro booleano che indica la sua appartenenza alla regione Goal, da un elenco dei nodi precedenti connessi e uno di nodi successivi connessi.
In figura 3.1 è possibile vedere nel dettaglio la composizione della struttura `Nodes`:

Nodes(i)	.State	.x
		.y
	.goalReach	0/1
	.componentNumber	Scalare
	.idPrec	[id1, id2, id3, ...]
	.idSucc	[id1, id2, id3, ...]

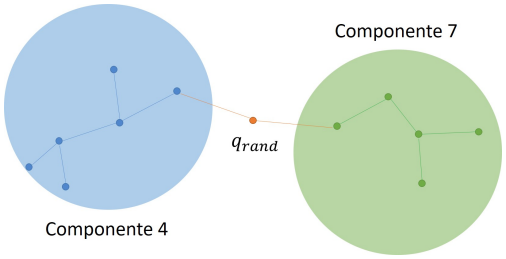
Figura 3.1: Struttura `Nodes`

Come si può notare, ogni nodo è caratterizzato da un ulteriore parametro chiamato `componentNumber`, ovvero uno scalare che ne identifica la componente connessa di appartenenza. Quest'ultima caratteristica del nodo è essenziale in quanto permette poi di verificare, nella fase finale di Quering dell'algoritmo, se esiste almeno un nodo nella regione di Goal appartenente alla stessa componente connessa del nodo iniziale, ovvero quella indicata dal numero 1.

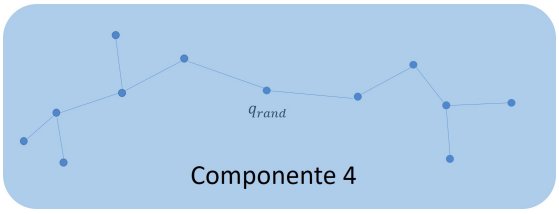
Alla generazione del nodo q_{rand}, il codice crea una nuova componente connessa assegnandole un valore pari alla cardinalità delle componenti aumentata di uno. Conseguentemente, nel caso in cui q_{rand} venisse unito ad un nodo appartenente ad



(a) Generazione q_{rand}



(b) Connessione ad entrambe le componenti



(c) Unione delle due componenti

Figura 3.2: Annessione di due componenti

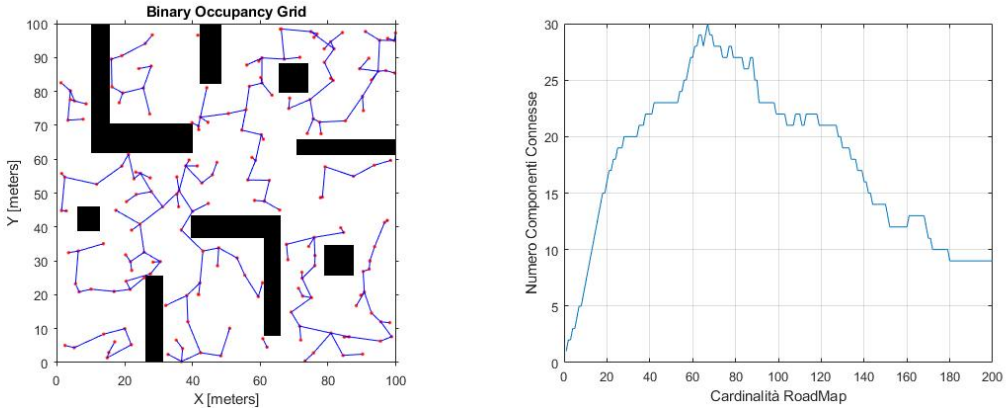
una componente connessa già esistente, ne erediterebbe l'appartenenza a quest'ultima. Nel caso invece in cui q_{rand} venisse generato esattamente in mezzo a due componenti connesse differenti e venisse connesso quindi ad entrambe, il codice creerebbe un'unica componente assegnandole un numero pari al minimo valore identificativo delle due. Quest'ultimo passaggio lo esegue aggiornando al nuovo valore il parametro `componentNumber` di tutti i nodi della componente.

Il numero di componenti connesse avrà un andamento crescente nelle prime iterazioni e calante nelle iterazioni finali (figure 3.3b e 3.4b). Infatti, inizialmente la RoadMap è vuota e i nodi generati difficilmente possono essere connessi tra loro; essi andranno quindi a formare nuove componenti connesse. All'aumentare del numero di nodi aggiunti alla RoadMap vi sarà una probabilità sempre maggiore di poter trovare nodi nelle vicinanze di q_{rand} e la loro conseguente connessione implicherebbe la diminuzione del numero di componenti singole.

Quering

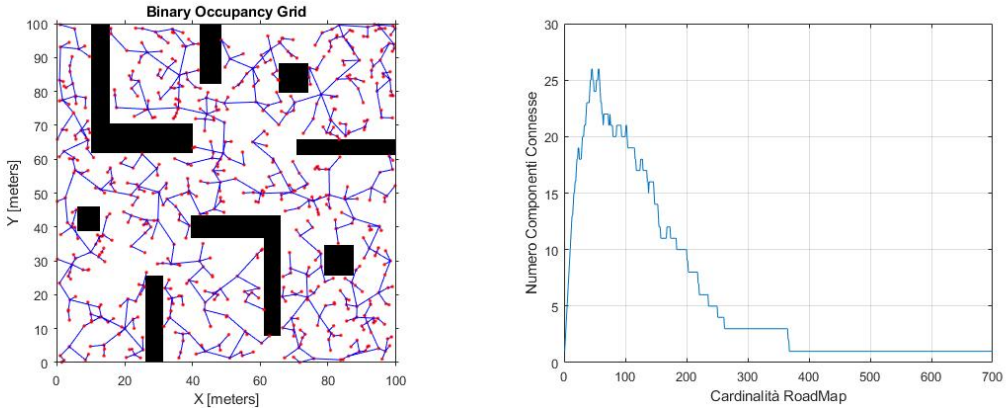
Una volta ottenuta la RoadMap nella fase di Learning, è necessario trovare un cammino che connetta q_{start} a q_{goal}.

Quest'ultimo passaggio consiste essenzialmente nella ricerca di un nodo all'interno della regione di Goal che appartenga alla componente connessa 1; in questo modo si avrebbe la certezza sull'esistenza di tale traiettoria.



(a) RoadMap (b) Numero Componenti

Figura 3.3: PRM simulato per **maxNodes**=200 e **discRadius**=10



(a) RoadMap (b) Numero Componenti

Figura 3.4: PRM simulato per **maxNodes**=700 e **discRadius**=10

Il codice esegue una scansione di tutti i nodi della RoadMap alla ricerca di quelli che che abbiano entrambi i parametri **goalreach** e **componentNumber** uguali a 1.

Per quanto riguarda la ricerca effettiva della traiettoria finale si è optato per l'utilizzo di alcune funzioni esistenti e disponibili nelle librerie di *Matlab* che sfruttano il concetto del **Digrafo**. Il Digrafo essenzialmente è un grafo orientato creato sulla base di una **Matrice delle Adiacenze**, ovvero una matrice il cui valore generico posto nella posizione (i,j) corrisponde al peso dell'arco che unisce il nodo *i-esimo* al nodo *j-esimo*. Nel caso in cui tale arco non esistesse, il peso assumerà il valore 0.

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{bmatrix}$$

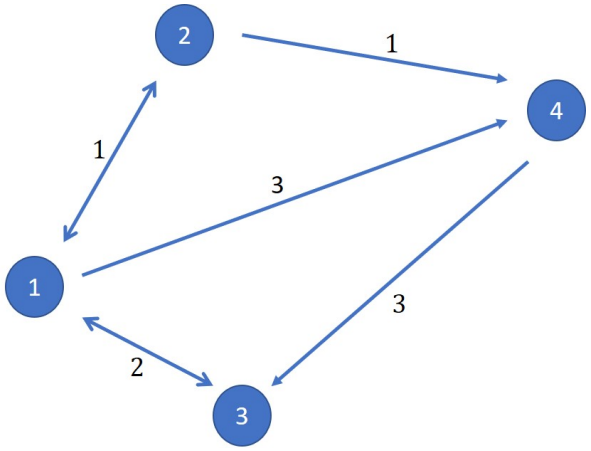


Figura 3.5: Digrafo creato sulla base della Matrice delle Adiacenze **A**

Nella fase di *Quering* per prima cosa viene creata la matrice A sondando l'intera struttura `Nodes`. In seguito viene inizializzato l'oggetto `digraph` tramite il comando `G = digraph(A)`. A questo punto, la funzione `shortestpathtree(G,s,t)` permette di trovare il cammino meno costoso che, partendo da un nodo sorgente **s**, conduca al nodo finale **t** restituendone il relativo insieme di archi e il costo finale della traiettoria.

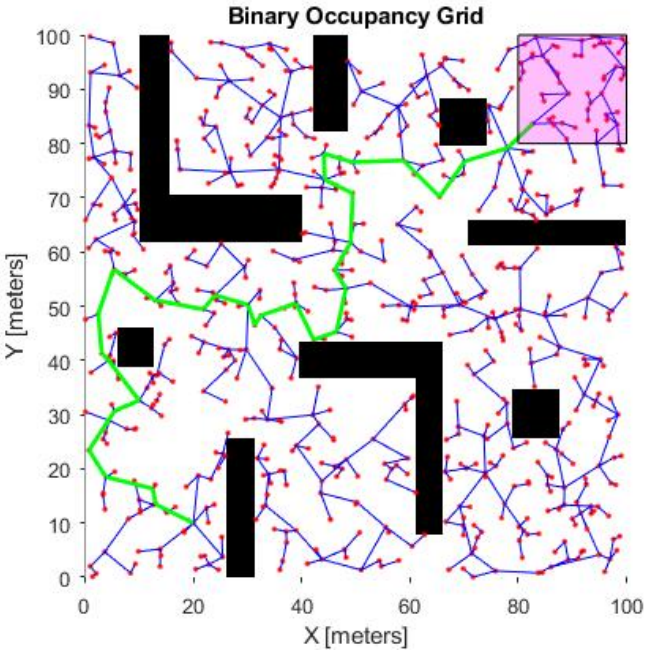


Figura 3.6: PRM: traiettoria finale

3.4 Algoritmo sPRM

L'algoritmo sPRM (simplified PRM)[1] è una versione semplificata dell'algoritmo standard PRM. La differenza rispetto a PRM sta nel semplice fatto che questo algoritmo non prevede una selezione intelligente dei nodi all'interno della regione circolare di centro q_{rand} e raggio **discRadius**.

In sintesi sPRM connette il nodo q_{rand} a tutti i nodi appartenenti alla suddetta regione indipendentemente dal fatto che la relativa connessione possa creare un ciclo all'interno della RoadMap.

Grazie a questa semplificazione non è più necessario generare la RoadMap in modo progressivo, ovvero aggiungendo ad ogni iterazione un nodo alla volta, e per questo l'algoritmo inizializza l'insieme V annettendo tutti i nodi generati unendoli poi in una fase successiva.

Il motivo per cui si è deciso di riportare tale algoritmo è il fatto che questa versione semplificata di PRM in realtà è più efficiente di PRM.

Inoltre è utile descriverne la logica di funzionamento per poter meglio introdurre la versione ottima PRM* la quale si tratta proprio di un'estensione di sPRM.

Algorithm 2 sPRM: Fase di Learning

1: $V \leftarrow \{q_{start}\} \cup [SampleFree_i]_{i=1,\cdots,n}$; $E \leftarrow \emptyset$;
2: **for all** $v \in V$ **do**
3: $U \leftarrow$`Near`$(G = (V, E), v, discRadius) \backslash \{v\}$;
4: **for all** $u \in U$ **do**
5: **if** `CollisionFree`(v,u) **then**
6: $E \leftarrow E \cup \{$(v,u),(u,v)$\}$
7: **end if**
8: **end for**
9: **end for**
10: **return** $G = (V, E)$;

Nelle implementazioni pratiche l'algoritmo sPRM può aggiungere nodi all'insieme U seguendo diverse strategie:

- **Fixed-Radius**: l'algoritmo funziona esattamente come descritto nell'Algoritmo 1, i nodi aggiunti a U sono quelli la cui distanza da q_{rand} è inferiore al parametro Raggio fissato prima dell'esecuzione del codice.
- **Variable-Radius**: in questo caso il parametro Raggio non è un valore fisso, ma viene scelto come funzione del numero di nodi in V.
- k**nearest**: nell'insieme U vengono aggiunti solo i primi k nodi più vicini a q_{rand}. All'elenco dei parametri andrà inserito quindi anche k.

3.5 Algoritmo PRM*

Nell'articolo accademico *"Sampling-based algorithms for optimal motion planning"*[1] vengono introdotte per la prima volta le versioni ottimizzate degli algoritmi PRM e RRT chiamati **PRM*** e **RRT***.

PRM* ha la stessa struttura dell'algoritmo sPRM, l'unica differenza consiste nel fatto che **discRadius** viene definito come funzione del numero di nodi contenuti in V. In questo caso si utilizzerà la funzione riportata nell'articolo:

$$r = r(n) = \gamma_{PRM} \left(\frac{log(n)}{n} \right)^{\frac{1}{d}}$$

in cui:

- **n**: numero di nodi in V;
- **d**: cardinalità dello spazio di stato del robot considerato;

- $\gamma_{PRM} > \gamma_{PRM}* = 2\left(1+\frac{1}{d}\right)^{\frac{1}{d}}\left[\frac{\mu(\chi_{free})}{\zeta_d}\right]^{\frac{1}{d}}$

  - $\mu(\chi_{free})$: misura di Lebesgue dello spazio libero considerato, in questo caso specifico si considera come misura l'area dell'insieme delle regioni del piano non occupate da un ostacolo;

  - χ_{free}: spazio non occupato da ostacoli;

  - ζ_d: volume della bolla di raggio unitario in un spazio Euclideo di dimensione d. Questa misura deriva dalla formula:

$$\zeta_d = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)}$$

    * $\Gamma(x) = (x-1)!$ è la Funzione Gamma;

Algorithm 3 PRM*: Fase di Learning

$V \leftarrow \{q_{start}\} \cup [SampleFree_i]_{i=1,\cdots,n}$; $E \leftarrow \emptyset$;
for all $v \in V$ **do**
  $U \leftarrow$`Near`$(G=(V,E), v, \gamma_{PRM}\left(\frac{log(n)}{n}\right)^{\frac{1}{d}})\backslash\{v\}$;
  for all $u \in U$ **do**
    if `CollisionFree`(v,u) **then**
      $E \leftarrow E \cup \{$(v,u),(u,v)$\}$
    end if
  end for
end for
return $G=(V,E)$;

In figura 3.7 viene riportato l'andamento della funzione $\mathbf{r(n)}$ per $n = 1, 2, \cdots, 3000$.

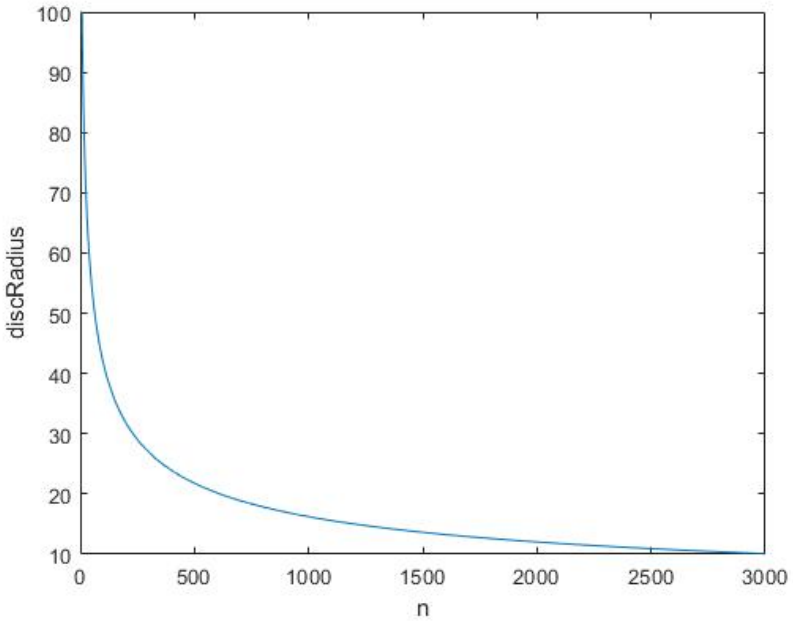


Figura 3.7: funzione **r(n)**

Simulando l'algoritmo per cento nodi si ottiene il risultato riportato in figura 3.8.

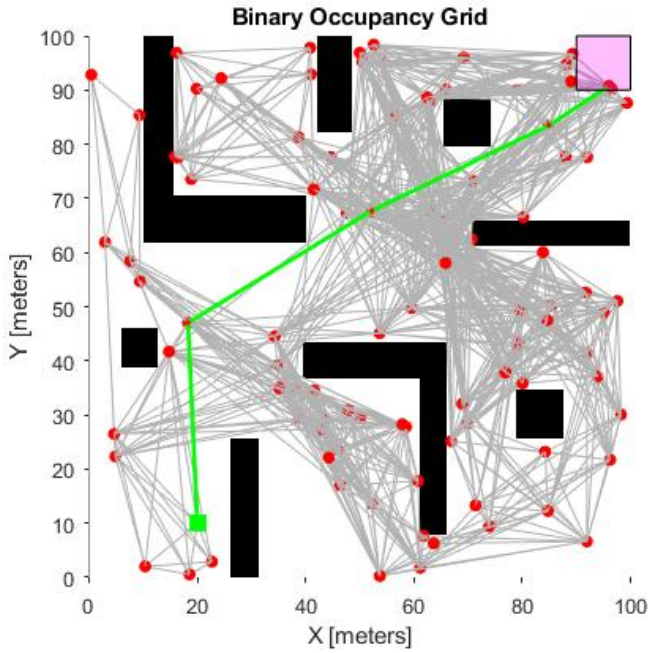
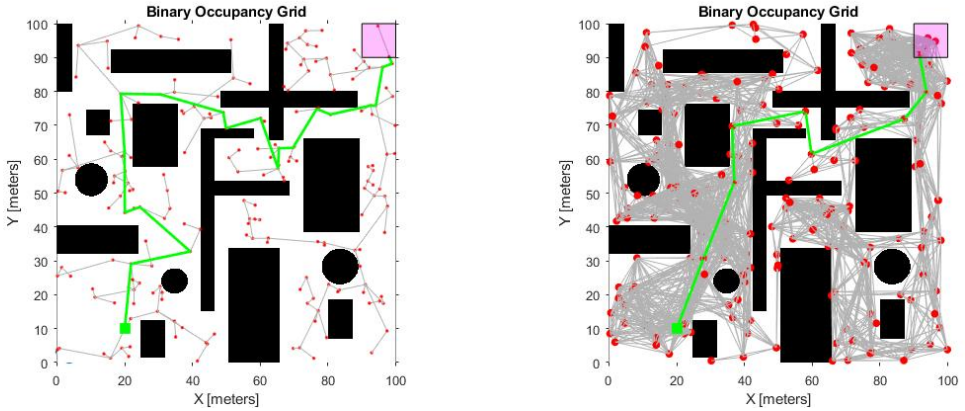


Figura 3.8: PRM*: Traiettoria finale per 100 nodi

Per poter confrontare gli algoritmi PRM e PRM* sono state effettuate diverse simulazioni con lo stesso numero di nodi e all'interno del medesimo ambiente. I risultati, com'è ben visibile nella figura 3.9, evidenziano la superiorità di PRM*, il quale trova traiettorie sempre meno costose in termini di distanza percorsa.



(a) PRM: 217 metri (b) PRM*: 148 metri

Figura 3.9: Simulazioni per **200** nodi

Capitolo 4

Rapidly-exploring random tree

Gli algoritmi della classe RRT, nati successivamente a PRM, non creano una Road-Map connettendo coppie di nodi generati casualmente nello spazio, ma espandono progressivamente un Albero, composto da nodi e archi, a partire dal nodo radice q_{start}. Ad ogni iterazione viene generato un nodo q_{rand} con probabilità uniforme in C_{free} il quale indica la direzione verso cui l'albero si dovrà espandere.

Sebbene PRM garantisca una miglior copertura dell'intero spazio C_{free}, esso presenta dei limiti nelle applicazioni a modelli più complessi con cui viene rappresentato il robot.

In RRT invece, per come viene costruito l'albero, il verso di percorrenza sarà sempre univoco. Ogni nodo dell'albero potrà essere raggiunto sempre e solo da un'unica sequenza di nodi primo dei quali ci sarà q_{start}.

Si pensi al semplice caso di un automobile o più generalmente di un mezzo con movimenti limitati da vincoli anolonomi; in questi casi se lo spazio di stato comprendesse anche la velocità, unire una coppia di nodi significherebbe imporre al robot una traiettoria che non solo lo conduca alla posizione del nodo finale, ma che lo faccia arrivare con un preciso orientamento (dato dalla velocità nello stato finale). In questo senso il verso di percorrenza andrà a cambiare radicalmente la natura della traiettoria.

L'obiettivo di questo capitolo è la comprensione delle logiche fondamentali su cui si basano gli algoritmi RRT. Per questa ragione ci si è limitati ancora al caso base in cui lo stato del robot è identificato dal solo vettore posizione $[x, y]$.

4.1 Algoritmo RRT

L'algoritmo base della categoria Rapidly-exploring Random Tree, studiato nell'articolo accademico *Sampling-based algorithms for optimal motion planning*[1],

viene riportato in forma di pseudo codice nell'algoritmo 4.

Per prima cosa RRT inizializza l'Albero $G = (V, E)$ inserendo nell'insieme dei nodi V il solo nodo iniziale q_{start}.

Per un numero specifico di iterazioni, l'algoritmo genera casualmente un nodo q_{rand} in C_{free}. Questo verrà connesso all'albero attraverso una traiettoria locale calcolata da una specifica `Steering function`.

La traiettoria uscente dal calcolo della `Steering Function` non sarà altro che una fitta sequenza di nodi i quali non dovranno appartenere ad una delle regioni occupate dagli ostacoli.

Nel caso in cui tale traiettoria venga validata dalla funzione `CollisionFree`, essa sarà aggiunta all'insieme degli archi E.

Algorithm 4 RRT

1: $V \leftarrow \{q_{start}\}; E \leftarrow \emptyset;$
2: **for** $i = 1, \cdots,$ **maxNodes do**
3: $q_{rand} \leftarrow$ `SampleFree`;
4: $q_{nearest} \leftarrow$ `Nearest`$(G = (V, E)),\ q_{rand})$
5: $q_{new} \leftarrow$ `Steer`$(q_{nearest}, q_{rand})$
6: **if** `CollisionFree`$(q_{nearest}, q_{new})$ **then**
7: $V \leftarrow V \cup \{q_{new}\}; E \leftarrow E \cup \{(q_{nearest}, q_{new})\}$
8: **end if**
9: **end for**
10: **return** $G = (V, E);$

- **Parametri**
  - **maxNodes**: è il parametro che indica il numero finale di nodi contenuti nell'albero. Nel caso dell'algoritmo PRM esso coincide perfettamente con il numero di iterazioni dell'algoritmo. In questo caso invece può capitare che nonostante q_{rand} sia generato in C_{free} dalla funzione `SampleFree`, la traiettoria che lo connette all'albero sia irrealizzabile a causa di collisioni con ostacoli; in tal caso il nodo q_{rand} verrebbe scartato e l'algoritmo eseguirebbe un ciclo senza aggiungere alcun nodo all'albero.
  - η: il ruolo di questo parametro viene spiegato nel paragrafo successivo dedicato alle Funzioni dell'algoritmo, ed in particolare nella funzione `Steer`.
- **Funzioni**
  - `Nearest`: riceve come argomenti L'intero Albero costituito dall'insieme dei nodi e delle traiettorie locali che li connettono, e un nodo particolare

q_{rand}. La funzione trova tra tutti i nodi dell'Albero quello più vicino a q_{rand}. Il termine "vicino" non è legato ad un concetto assoluto in quanto può essere attribuito al nodo con distanza euclidea minore a q_{rand} o al nodo la cui traiettoria di connessione con q_{rand} minimizza una determinata funzione di costo.

– `ObstacleFree`: La funzione ha lo scopo di verificare secondo specifiche modalità già presentate nel Capitolo 2, l'appartenenza dell'intera traiettoria allo spazio C_{free}.

– `Steer`: è la notazione usata per indicare la specifica `Steering Function` usata in questo caso. Essa riceve come argomenti due nodi, indicati a titolo di esempio come q_x e q_y. La funzione si comporta come pianificatore locale che calcola la traiettoria di connessione q_x a q_y. Nelle implementazioni pratiche degli algoritmi analizzati in questo capitolo, la funzione `Steer` restituisce un terzo nodo q_z come nodo che minimizza la distanza $\| q_z - q_y \|$ e verifichi allo stesso tempo la condizione $\| q_z - q_x \| < \eta$, dove η è un ulteriore argomento da passare in ingresso alla funzione (figura 4.1).

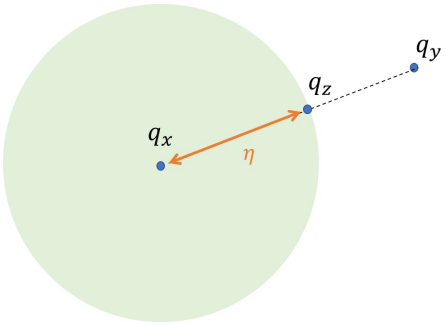
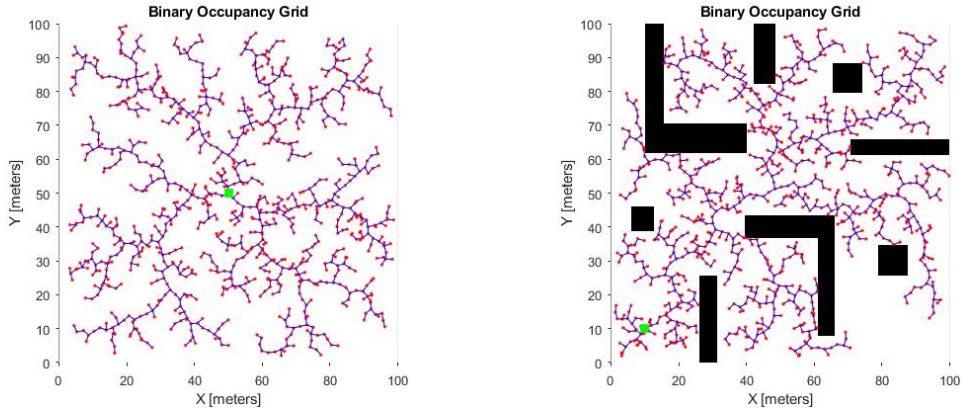


Figura 4.1: funzionamento della `Steering Function`

In figura 4.2a viene riportato l'Albero generato in seguito all'esecuzione dell'algoritmo scegliendo come parametri **maxNodes=1000** e η**=2 m**. In rosso vengono indicati i nodi appartenenti all'insieme V, in blu gli archi contenuti in E e in verde il nodo radice q_{start}. In figura 4.2b viene invece riportato la stessa simulazione nel caso in cui ci siano degli ostacoli all'interno della mappa.



(a) Senza ostacoli (b) Con ostacoli

Figura 4.2: RRT simulato per **maxNodes=1000** e $\eta = 2m$

4.1.1 Implementazione

Come nel caso dell'algoritmo PRM, anche in RRT si può separare il codice in due fasi differenti: nella prima fase chiamata *Learning* si genera l'albero con tecniche basate sulla probabilità, nella seconda o fase di *Quering* si calcola la traiettoria migliore all'interno dell'albero partendo dal nodo q_{start} fino alla Regione di Goal.

Learning

La strategia adottata in questo caso è simile a quella usata per l'implementazione di PRM. Infatti l'Albero viene identificato da una variabile chiamata **Tree** di tipo *struct*. La struttura Tree è composta da diverse strutture interne come si può notare in figura 4.3. Di particolare interesse è la sotto-struttura `vertices`, composta dall'elenco dei nodi appartenenti a V. Ad ogni nodo di questa sotto-struttura saranno assegnati diversi attributi:

- **x,y**: coordinate del punto nel Piano Cartesiano.
- **previous**: indice del nodo precedente.
- **cost**: costo della traiettoria che parte da q_{start} e arriva al nodo corrente. Il costo in questo caso specifico è considerato come la distanza Euclidea tra i nodi, ma in generale può essere attribuito ad una cifra di merito valutata nel caso della traiettoria specifica.
- **goalreach**: valore Booleano che indica l'appartenenza o meno del nodo alla Regione di Goal.
- **trajectory**: variabile di tipo `substruct`, ovvero sottostruttura che indica la traiettoria locale che connette il nodo corrente a quello che lo precede.

Essa contiene diversi attributi: le coordinate del nodo di partenza (nodo precedente), quelle del nodo finale (nodo corrente), costo della traiettoria singola, vettore di tutti gli istanti temporali con cui è stata campionata la traiettoria e l'elenco delle coordinate dei punti che la compongono.

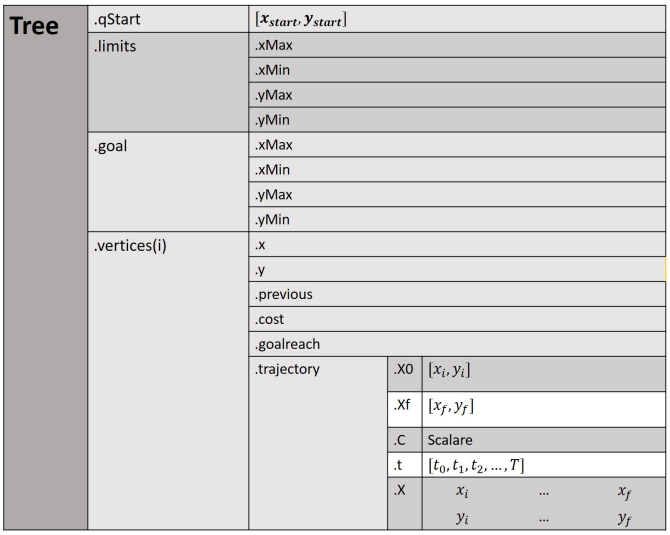


Figura 4.3: Struttura `Tree`

In sintesi nella fase di Learning l'algoritmo segue questa sequenza di passaggi:

1. Generazione nodo q_{rand} all'interno di C_{free} tramite la funzione `SampleFree`.
2. Ricerca di $q_{nearest}$, ovvero del nodo dell'albero più vicino a q_{rand}.
3. Calcolo della traiettoria di connessione tra $q_{nearest}$ e q_{rand} attraverso la funzione `Steer`.
4. Controllo di validità della traiettoria tramite `CollisionFree`.
5. Annessione del nodo q_{new} uscente dal calcolo effettuato da `Steer` alla struttura **Tree**.

Quering

La fase di *Quering* consiste essenzialmente nel ricercare all'interno dell'albero formato nella precedente fase di *Learning* la traiettoria meno costosa che possa condurre il robot dal nodo q_{start} all'interno della Regione di Goal. Per prima cosa quindi è necessario sondare la struttura **Tree** in cerca del nodo appartenente alla regione di Goal con minor costo indicato dall'attributo `cost`. In seguito, ripercorrendo al contrario le singole traiettorie locali e connettendole tra di loro, si può facilmente ricavare la traiettoria ricercata.

Il risultato sarà infine simile a quello riportato in figura 4.4.

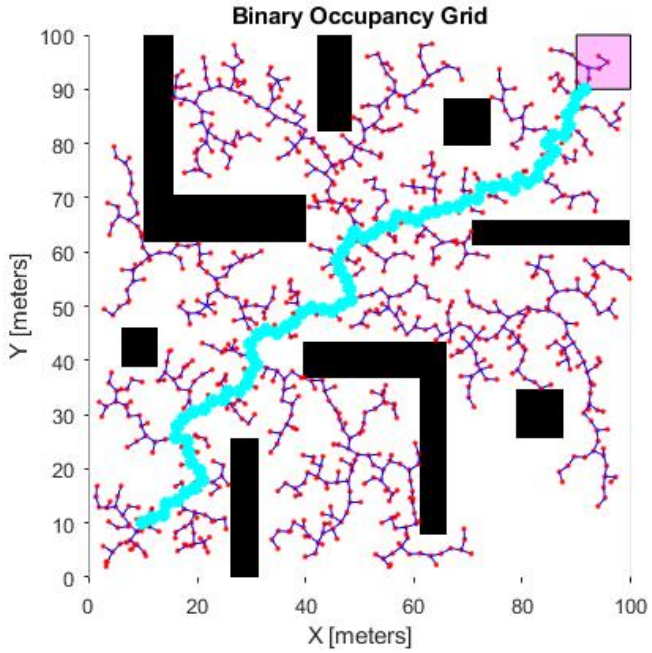


Figura 4.4: Struttura `Tree`

4.2 Algoritmo RRG

L'algoritmo RRG (Rapidly exploring Random Graph) consiste essenzialmente nella costruzione progressiva di una RoadMap in maniera molto simile a come viene costruito l'albero in RRT. A differenza di quest'ultimo però la RoadMap costruita può avere al suo interno cicli. La somiglianza all'algoritmo RRT viene dal fatto che nella fase di Learning, la RoadMap si espande partendo da un nodo di origine esattamente nello stesso modo con cui funziona RRT. Nella fase successiva al calcolo effettuato dal pianificatore locale l'algoritmo prevede di connettere q_{rand} non solo al nodo più vicino $q_{nearest}$, ma a tutti i nodi con una distanza da q_{rand} inferiore ad una soglia stabilita. In questo lavoro di tesi RRG gioca solo un ruolo introduttivo all'algoritmo RRT* il quale non sarà altro che una sua estensione.

Algorithm 5 RRG

1: $V \leftarrow \{q_{start}\}; E \leftarrow \emptyset;$
2: **for** $i = 1, \cdots,$ **maxNodes do**
3: $q_{rand} \leftarrow$ `SampleFree`;
4: $q_{nearest} \leftarrow$ `Nearest`$(G = (V, E)),\ q_{rand})$
5: $q_{new} \leftarrow$ `Steer`$(q_{nearest}, q_{rand})$
6: **if** `ObstacleFree`$(q_{nearest}, q_{new})$ **then**
7: $X_{near} \leftarrow$ `Near`$(G = (V, E), q_{new}, min\{\gamma_{RRG}(\frac{log(card(V))}{card(V)})^{\frac{1}{d}}, \eta\}$
8: $V \leftarrow V \cup \{q_{new}\}$
9: $E \leftarrow E \cup \{(q_{nearest}, q_{new}), (q_{new}, q_{nearest})\}$
10: **for all** $q_{near} \in X_{near}$ **do**
11: **if** `ObstacleFree`(q_{near}, q_{new}) **then**
12: $E \leftarrow E \cup \{(q_{near}, q_{new}), (q_{new}, q_{near})\}$
13: **end if**
14: **end for**
15: **end if**
16: **end for**
17: **return** $G = (V, E);$

I parametri e le funzioni sono gli stessi degli algoritmi analizzati in precedenza. La differenza tra γ_{RRG} e γ_{RRT} è solo una differenza di notazione, infatti il calcolo di γ_{RRG} è uguale a quello di γ_{RRT}.

4.3 Algoritmo RRT*

L'algoritmo **RRT*** (*Optimal*-Rapidly-exploring Random Tree) è un'evoluzione di RRG a cui viene aggiunta una fase di ottimizzazione dell'albero.

Questa fase, detta **Rewiring**, viene richiamata all'interno di ogni iterazione. Come suggerisce il termine, essa rimodella l'albero aggiungendo o togliendo alcuni archi per ottimizzare le diverse traiettorie.

Per rendere l'algoritmo 6 più chiaro si è deciso di separare la fase dedicata al *Rewiring* riportandola nell'algoritmo 7.

Algorithm 6 RRT*

1: $V \leftarrow \{q_{start}\}; E \leftarrow \emptyset;$
2: **for** $i = 1, \cdots,$ **maxNodes do**
3: $q_{rand} \leftarrow \texttt{SampleFree}$;
4: $q_{nearest} \leftarrow \texttt{Nearest}(G = (V, E)),\ q_{rand})$
5: $q_{new} \leftarrow \texttt{Steer}(q_{nearest}, q_{rand})$
6: **if** $\texttt{ObstacleFree}(q_{nearest}, q_{new})$ **then**
7: $X_{near} \leftarrow \texttt{Near}(G = (V, E), q_{new}, min\{\gamma_{RRT*}(\frac{log(card(V))}{card(V)})^{\frac{1}{d}}, \eta\}$
8: $V \leftarrow V \cup \{q_{new}\}$
9: `Rewiring`
10: **end if**
11: **end for**
12: **return** $G = (V, E)$;

Algorithm 7 `Rewiring`

1: $q_{min} \leftarrow q_{nearest}$
2: $c_{min} \leftarrow \texttt{Cost}(q_{nearest}) + c(\texttt{Line}(q_{nearest}, q_{new}))$
3: **for all** $q_{near} \in X_{near}$ **do**
4: **if** $\texttt{ObstacleFree}(q_{near}, q_{new}) \wedge \texttt{Cost}(q_{near}) + c(\texttt{Line}(q_{near}, q_{new})) < c_{min}$ **then**
5: $q_{min} \leftarrow q_{near}$
6: $c_{min} \leftarrow \texttt{Cost}(q_{near}) + c(\texttt{Line}(q_{near}, q_{new}))$
7: **end if**
8: **end for**
9: $E \leftarrow E \cup \{(q_{min}, q_{new})\}$
10: **for all** $q_{near} \in X_{near}$ **do**
11: **if** $\texttt{ObstacleFree}(q_{new}, q_{near}) \wedge \texttt{Cost}(q_{new}) + c(\texttt{Line}(q_{new}, q_{near})) < \texttt{Cost}(q_{near})$ **then**
12: $q_{parent} \leftarrow \texttt{Parent}(q_{near})$
13: $E \leftarrow (E \backslash \{(q_{parent}, q_{near})\}) \cup \{(q_{new}, q_{near})\}$
14: **end if**
15: **end for**

Parametri

I parametri su cui si basa il funzionamento dell'algoritmo sono gli stessi degli algoritmi precedenti. Anche in questo caso γ_{RRT*} si differenzia da γ_{PRM} solo per la notazione e il calcolo rimane lo stesso riportato nel Capitolo 3. Si noti però che in questo caso l'area utilizzata per il *Rewiring* avrà un raggio dinamico rispetto alla cardinalità corrente dell'albero. Infatti, se in PRM* tale misura veniva calcolata

prima dell'esecuzione del codice e rimaneva invariata per tutte le iterazioni di questo, in RRT* il raggio decrescerà all'aumentare del numero di nodi aggiunti all'albero.

Tale procedura permette di connettere inizialmente nodi molto distanti tra loro in modo da arrivare il più presto possibile ad una soluzione. In seguito, con l'aumentare del numero di nodi, le connessioni verranno permesse solo tra nodi più vicini migliorando così le traiettorie già trovate. Inoltre vi è da dire che all'aumentare della cardinalità dell'albero, aumenta anche la probabilità di trovare nodi nella regione circolare necessaria per il *Rewiring* comportando in questo modo una crescita esponenziale nei tempi di calcolo.

In seguito ad alcuni esperimenti si è notato come la logica con cui viene assegnato il valore al raggio può essere estremizzata facendo in modo che il raggio diminuisca più rapidamente:

$$r(n) = \gamma_{PRM} \left(\frac{10\ log(n)}{n^2} \right)^{\frac{1}{d}}$$

Questa variante (figura 4.5) ha un notevole vantaggio rispetto alla precedente dal punto di vista del tempo impiegato dal codice per calcolare la soluzione.

D'altro canto però vi è un peggioramento nella probabilità di successo.

La scelta della metodologia da adottare e del numero di nodi da connettere all'albero si tratta dunque di un trade-off tra il tempo di esecuzione del codice e la probabilità di trovare una soluzione al problema.

La scelta dovrà essere ponderata in base alle priorità del caso e alla configurazione della mappa in cui si muove il robot.

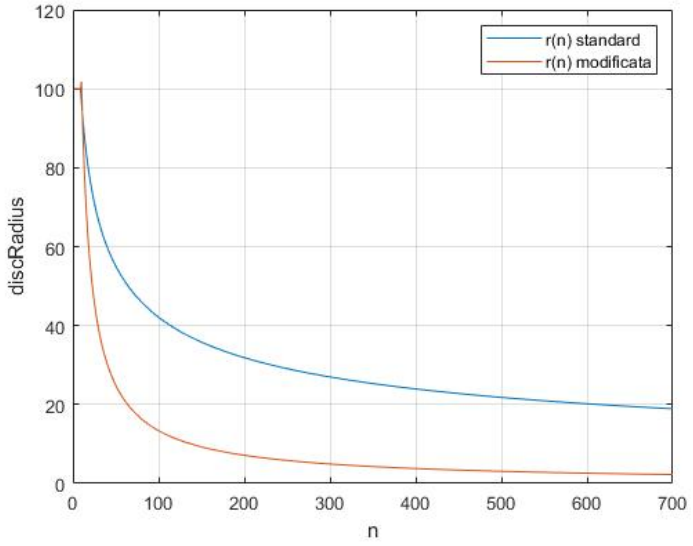


Figura 4.5: Confronto tra le due diverse funzioni **r(n)**

Funzioni

Come si può notare nella stesura dell'Algoritmo 6, vengono introdotte diverse nuove funzioni per questioni di chiarezza:

- `Cost`: riceve come argomento in ingresso il nodo q_x e ne calcola il costo impiegato dal robot per raggiungerlo partendo dallo stato iniziale q_{start}. Anche in questo caso il costo sarà calcolato sommando le distanze Euclidee tra i diversi nodi della traiettoria.
- `Line`: restituisce la traiettoria che connette i due nodi tracciando un segmento.
- `c`: riceve come input una singola traiettoria e ne ricava il costo. Quindi il comando `c(Line(`q_{near}, q_{new}`))` calcola la distanza euclidea tra i nodi q_{near} e q_{new}.
- `Parent`: restituisce il nodo precedente a quello passato come argomento.

4.3.1 Analisi

La sequenza di comandi descritta nello pseudo-codice viene qui tradotta con il fine di chiarire meglio i passaggi dell'Algoritmo 6. In particolare, come si potrà notare nella figura 4.7, si cercherà di spiegare meglio la logica con cui viene fatto il *Rewiring* dell'albero:

1. L'insieme V inizialmente è composto solo dal nodo q_{start}.
2. Generazione del nodo q_{rand} casualmente nello spazio libero C_{free}.
3. Ricerca del nodo $q_{nearest}$ più vicino a q_{rand}.
4. Calcolo del nodo q_{new} secondo la logica seguita dalla funzione `Steer`.
5. Il Pianificatore Locale controlla la validità della traiettoria che connette q_{new} a $q_{nearest}$.
6. Calcolo dell'insieme X_{near} contenente tutti i nodi a distanza da q_{new} inferiore della soglia indicata.
7. Annessione di q_new all'insieme V.
8. Vengono inizializzate due variabili ausiliarie utili per la fase di *Rewiring*:
   - q_{min} inizializzata a $q_{nearest}$, è il nodo precedente a q_{new} che ne minimizza il costo della traiettoria $q_{start} \rightarrow q_{min} \rightarrow q_{new}$.
   - c_{min} è il costo relativo all'intero percorso $q_{start} \rightarrow q_{min} \rightarrow q_{new}$.

9. Dalla riga 11) dello pseudo codice, inizia la fase di **Rewiring** (figura 4.7): in questa fase l'algoritmo trova il nodo all'interno di X_{near} che, se considerato come nodo precedente a q_{new}, possa minimizzare il costo della traiettoria $q_{start} \rightarrow q_{min} \rightarrow q_{new}$ aggiornando in tal caso le variabili q_{min} e c_{min}. In sintesi, l'algoritmo ad ogni iterazione ottimizza i costi delle connessioni valutando se nelle vicinanze di q_{new} vi sia un possibile nodo precedente migliore di $q_{nearest}$. Ovviamente l'aggiornamento viene eseguito solo nel caso in cui la traiettoria da aggiungere non preveda collisioni con gli ostacoli.

10. Dopo aver trovato il modo migliore di connettere q_{new} all'Albero, l'algoritmo esegue un ulteriore fase in cui valuta se i nodi q_{near} dell'insieme X_{near} possano essere raggiunti con un costo minore considerando q_{new} come nodo precedente.

In figura 4.6 è possibile notare la struttura dell'Albero finale dal quale è stata ricavata la traiettoria finale.

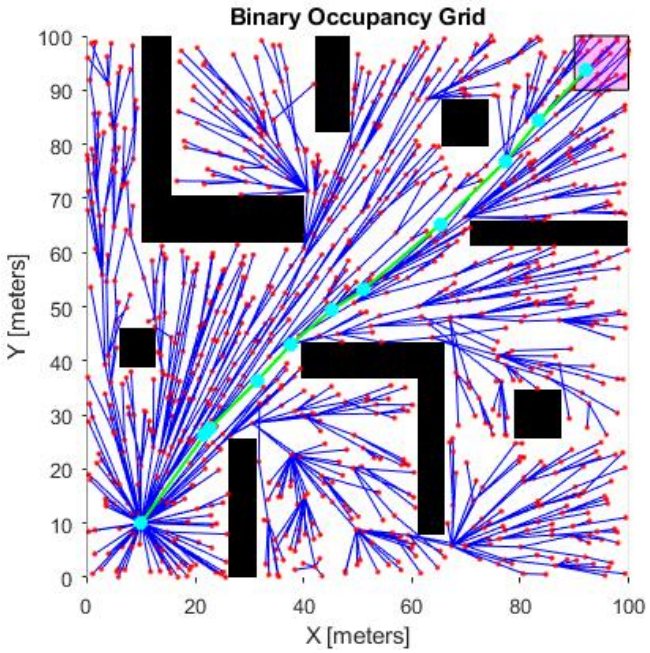
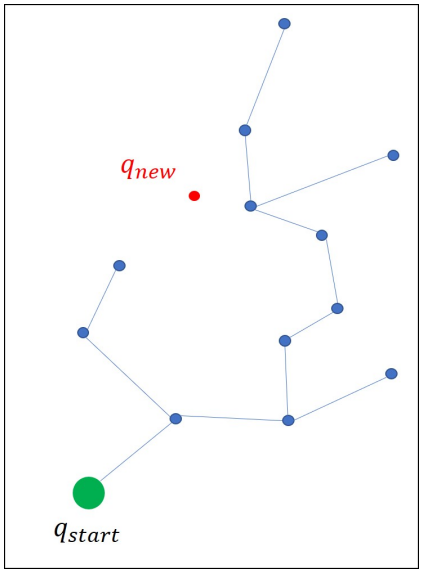
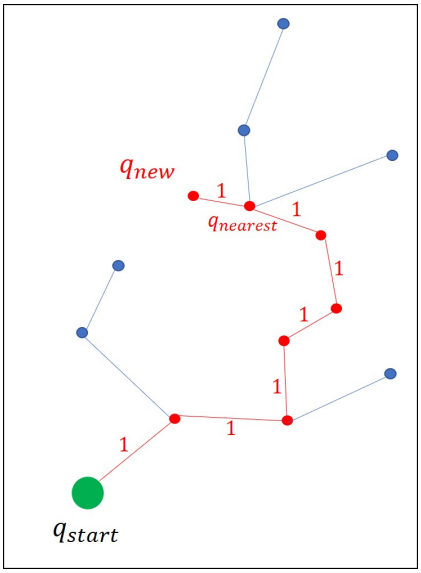


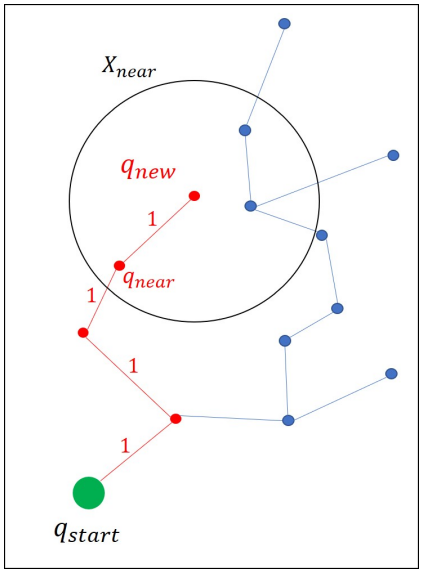
Figura 4.6: Simulazione per **1000 nodi**



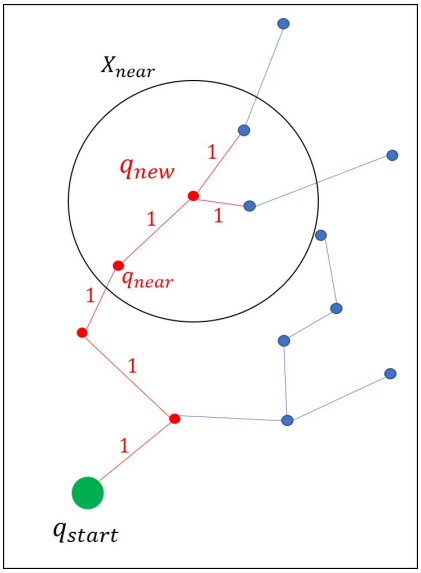
(a) Generazione q_{new}



(b) q_{new} viene connesso a $q_{nearest}$



(c) Calcolo del nodo padre di q_{new}



(d) Rewiring per ogni nodo di X_{near}

Figura 4.7: *Rewiring* dell'Albero

Capitolo 5

Controllo Ottimo applicato alla Pianificazione di Traiettoria

In questo Capitolo si affronterà il problema della Pianificazione di Traiettoria di un robot all'interno di un ambiente con ostacoli utilizzando alcune tecniche derivanti dalla teoria del **Controllo Ottimo**.

Gli algoritmi della classe Sampling-based finora analizzati si basavano sull'assunzione che il moto del robot veniva limitato solo da vincoli olonomi, ovvero vincoli che permettono al robot di muoversi lungo traiettorie create connettendo i nodi con dei semplici segmenti. Questo comportava enormi semplificazioni nell'esecuzione degli algoritmi, ma anche una grande limitazione per applicazioni pratiche di modelli più complessi. Si pensi al semplice caso di un'automobile dotata di quattro ruote, in tal caso la traiettoria che risulta dal calcolo di PRM o RRT è una linea spezzata e risulterebbe inattuabile per la dinamica del mezzo.

Per questa ragione recentemente la ricerca si è focalizzata sul cosiddetto *two point boundary value problem* per modelli dinamici più complessi chiamati *Kinodynamic systems*. Alcuni approcci numerici al problema, come per esempio *shooting method*, si sono rivelati troppo complessi dal punto di vista computazionale e, non godendo della proprietà di ottimalità si sono rivelati poco efficienti.

Kinodynmic-RRT* nasce proprio dall'esigenza di avere un algoritmo capace di adattarsi a situazioni dinamiche più complesse, mantenendo la proprietà di ottimalità di RRT* e l'efficienza per robot dotati di un numero elevato di dimensioni.

Kinodynamic-RRT* fu presentato per la prima volta nell'articolo "*Kinodynamic RRT*: Optimal Motion Planning for Systems with Linear Differential Constraints*"[2] del 2012; qui il problema della pianificazione di traiettoria viene rivalutato dal punto di vista del controllo ottimo.

La strategia adottata consiste fondamentalmente nel calcolo della legge di controllo considerando il caso in cui sia lo stato finale che il tempo per raggiungerlo

sono conosciuti.

Il risultato sarà un algoritmo che trova una soluzione asintoticamente ottima e capace di soddisfare in modo preciso le condizioni ai contorni.

In questo lavoro di tesi ci si è posti in seguito il problema dell'implementazione di un algoritmo che sfrutti il controllo ottimo anche nel caso di PRM*.

Sebbene in prima analisi si potrebbe pensare che PRM* non si presti molto all'applicazione per sistemi dinamici di questo tipo, il risultato è stato piuttosto soddisfacente.

Tale studio ha dato vita all'algoritmo **Kinodynamic-PRM***, capace di soddisfare le esigenze dettate dalla dinamica pur mantenendo i vantaggi relativi a PRM*.

5.1 Controllo Ottimo

Prima di analizzare l'applicazione del **Controllo Ottimo** al problema della Pianificazione di Traiettoria, è utile dare una panoramica generale su tale tecnica di controllo.

Il **Controllo Ottimo**[11][12][13] si pone l'obiettivo di determinare una legge di controllo $u(t)$ tale per cui il sistema da controllare soddisfi determinati vincoli fisici e allo stesso tempo minimizzi o massimizzi un criterio scelto, detto *funzione di costo* o *cifra di merito*.

I tre ingredienti fondamentali nella teoria del Controllo Ottimo sono: un modello in forma di stato del sistema dinamico da controllare, una funzione di costo che misura la performance della soluzione ottenuta, delle specifiche condizioni al contorno o vincoli fisici sugli stati e sul controllo.

Per prima cosa quindi si assuma di avere un sistema dinamico descritto dall'equazione:

$$\dot{x}(t) = f(x(t), u(t), t) \tag{5.1}$$

dove $x(t) \in \mathcal{X} = \mathbb{R}^n$ è la variabile di stato, $u(t) \in \mathcal{U} = \mathbb{R}^m$ è la variabile di controllo, $A \in \mathbb{R}^{n \times n}$.

Si assuma di avere una funzione di costo di questo tipo:

$$J := S(x(t_f), t_f) + \int_{t_0}^{t_f} f_0(x(t), u(t), t)dt \tag{5.2}$$

in cui il primo addendo pesa lo stato e il tempo finali, mentre il secondo addendo pesa l'intera evoluzione dello stato e delle variabili di controllo.

Le condizioni al contorno invece sono vincoli sullo stato iniziale ed eventualmente anche sullo stato e il tempo finali x_f e t_f.

L'obiettivo sarà dunque quello di trovare una funzione $u^*(t)$ con $t \in [t_0, t_f]$ tale che la funzione di costo 5.2 sia minimizzata.

5.1.1 Procedura per il calcolo della soluzione

Si definisce **Hamiltoniana** la funzione:

$$H(x(t), u(t), \lambda, t) := \; f_0(x(t), u(t), t) \; + \; \lambda^T(t) f(x(t), u(t), t) \tag{5.3}$$

in cui $\lambda(t)$ è chiamato *co-stato* ed è un vettore di dimensioni $(n \times 1)$.

Il problema del controllo ottimo avrà soluzione solo se sono verificate le seguenti condizioni:

Equazioni del *co-stato*:

$$\dot{\lambda}(t) = -\frac{\partial H}{\partial x}^T \tag{5.4}$$

Equazioni dello stato:

$$\dot{x}(t) = \frac{\partial H}{\partial \lambda} \tag{5.5}$$

Equazioni del controllo:

$$\frac{\partial H}{\partial u} = 0 \tag{5.6}$$

mentre le **condizioni finali** si traducono in questo modo:

$$\left[H + \frac{\partial S}{\partial t}\right]_{t_f} \delta t_f \; + \; \left[\frac{\partial S}{\partial x} - \lambda^T(t)\right]_{t_f} \delta x_f = 0 \tag{5.7}$$

dove δt_f e δx_f sono variazioni arbitrarie di t_f e x_f.

L'equazione è verificata se i singoli coefficienti sono nulli:

$$\left[H + \frac{\partial S}{\partial t}\right]_{t_f} = 0 \tag{5.8}$$

$$\left[\frac{\partial S}{\partial x} - \lambda^T(t)\right]_{t_f} = 0 \; \rightarrow \; \lambda(t_f) = \left(\frac{\partial S}{\partial x}\right)_{t_f}^T \tag{5.9}$$

Nel caso in cui lo stato finale x_f sia fissato e il tempo finale t_f sia libero, allora $\delta x_f = 0$ mentre $\delta t_f \neq 0$, l'unica condizione al contorno sarà quindi la 5.8 mentre nel caso contrario sarà la 5.9. Qualora siano fissati entrambi, l'equazione 5.7 è sempre verificata e non serve aggiungere nessuna delle due condizioni sopra citate.

Per trovare la soluzione al problema di minimizzazione nel sistema dinamico vincolato, si procede seguendo questa sequenza di passi:

1. Definire la funzione Hamiltoniana:

$$H(x(t), u(t), \lambda, t) = f_0(x(t), u(t), t) + \lambda^T(t) f(x(t), u(t), t)$$

2. Minimizzare H rispetto a u:

$$\frac{\partial H}{\partial u} = 0 \ \rightarrow \ u^*(t) = h(x(t), \lambda(t), t)$$

3. Sostituire $u^*(t)$ in $H(x(t), u(t), \lambda, t)$ ottenendo così $H^*(x(t), u^*(t), \lambda, t)$

4. Definire il sistema composto da *2n* equazioni differenziali ottenuto considerando l'equazione di stato e del *co*-stato per la funzione H^*:

$$\dot{x}(t) = f(x(t), u^*(t), t)$$

$$\dot{\lambda}(t) = -\frac{\partial {H^*}^T}{\partial x}$$

5. Risolvere il sistema di equazioni differenziali del passaggio precedente, considerando le condizioni al contorno definite dallo stato iniziale e dallo stato e tempo finali qualora siano fissati.

6. Sostituire in $u^*(t) = h(x(t), \lambda(t), t)$ la funzione dello stato $x(t)$ e del *co*-stato $\lambda(t)$ ottenute dalla risoluzione del sistema.

5.2 Definizione del Problema nella Pianificazione di Traiettoria

Si ipotizzi di avere un robot la cui dinamica è descritta dalla seguente equazione lineare:

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{5.10}$$

dove $x(t) \in \mathcal{X} = \mathbb{R}^n$ è la variabile di stato, $u(t) \in \mathcal{U} = \mathbb{R}^m$ è la variabile di controllo, $A \in \mathbb{R}^{n \times n}$ e $B \in \mathbb{R}^{n \times m}$ sono conosciute e costanti.

Si assuma inoltre di conoscere il tempo finale della traiettoria τ. Una traiettoria è una variabile definita da una sequenza specifica di stati, input di controllo e un tempo finale τ. La notazione usata per la traiettoria è: $\pi = (x(), u(), \tau)$ in cui $x : [0, \tau] \mapsto \mathbb{R}^n$ e $u : [0, \tau] \mapsto \mathbb{R}^m$ sono le funzioni che associano ad ogni istante di tempo compreso tra 0 e τ uno stato e una variabile di controllo rispettivamente nei limiti imposti dall'equazione 5.10.

La funzione di costo che pesa la traiettoria π è rappresentata dalla seguente equazione:

$$c[\pi] = \int_0^\tau (1 + u(t)^T R u(t)) dt \tag{5.11}$$

Tale funzione penalizza allo stesso tempo la durata della traiettoria τ e gli sforzi dovuti all'azione di controllo $u(t)$, mentre la matrice $R \in \mathbb{R}^{m \times m}$ è una matrice definita positiva, costante e conosciuta che pesa l'azione delle singole variabili di controllo. Si noti che in questo caso specifico sarebbe ridondante aggiungere il termine fuori dall'integrale in quanto lo stato e il tempo finali sono fissati.

Si definisce ora l'insieme $\mathcal{X}_{free} \subseteq \mathcal{X}$ come l'insieme degli stati del robot all'interno dei limiti imposti dall'utente e fuori dalle regioni occupate dagli ostacoli. Allo stesso modo si definisce anche $\mathcal{U}_{free} \subseteq \mathcal{U}$, ovvero l'insieme delle variabili di controllo realizzabili dagli attuatori fisici.

Il problema può ora essere definito in questo modo:

Dato uno stato iniziale $x_{start} \in \mathcal{X}_{free}$ e uno stato finale $x_{goal} \in \mathcal{X}_{free}$ si vuole trovare la traiettoria ${\pi^*}_{free}$ priva di collisioni con gli ostacoli che minimizzi la funzione di costo (5.10).

$$\begin{aligned} {\pi^*}_{free} = argmin\{\pi \mid x(0) = x_{start} \ \wedge \ x(\tau) = x_{goal} \ \wedge \forall\{t \in [0,\tau]\}(x(t) \in \mathcal{X}_{free} \wedge \\ u(t) \in \mathcal{U}_{free})\} c(\pi) \end{aligned} \tag{5.12}$$

Chiamiamo π^* la traiettoria locale ottima che connette due generici punti x_0 e x_1 senza tener conto dei limiti e degli ostacoli e c^* il costo di tale traiettoria:

$$c^*[x_0, x_1] = min\{\pi \mid x(0) = x_0 \ \wedge \ x(\tau) = x_1\} c(\pi) \tag{5.13}$$

$$\pi^*[x_0, x_1] = argmin\{\pi \mid x(0) = x_0 \ \wedge \ x(\tau) = x_1\} c(\pi) \tag{5.14}$$

Si noti che per ogni $0 < t < \tau$ si ha che $c^*(x_0, x_1) = c^*(x_0, x(t)) + c^*(x(t), x_1)$. Dunque la traiettoria *collision-free* ottima ${\pi^*}_{free}$ tra gli stati x_{start} e x_{goal} consiste in una catena di connessioni tra diversi stati $(x_{start}, x_1, x_2, \cdots, x_{goal})$ appartenenti a $\mathcal{X}_{free}$.

Il problema si traduce di conseguenza nel calcolo della traiettoria locale ottima tra due singoli stati.

5.3 Traiettoria locale ottima tra due stati

La strategia adottata è quella di porci nel caso di un *fixed-final-time-fixed-final-state problem* con *Controllo in anello aperto*, chiamato così in quanto nella legge del controllo finale non vi è nessuna azione di *feedback* dello stato. In seguito ci si porrà il problema di come ricavare il tempo finale ottimo τ.

5.3.1 *fixed-final-state* e controllo in anello aperto

Supponiamo di avere uno stato iniziale x_0 e che l'obiettivo sia quello di guidare il robot esattamente fino allo stato finale x_1 in un tempo τ. Le condizioni al contorno saranno quindi semplicemente:

$$x(t_0) = x_0$$

$$x(\tau) = x_1$$

dato che δx_f e $\delta\tau$ sono uguali a zero.

Dalla 5.10 e la 5.11 si può ricavare la funzione Hamiltoniana:

$$H = u^T R u + \lambda^T (Ax + Bu) \tag{5.15}$$

Dall'equazione del controllo $\dfrac{\partial H}{\partial u} = 0$, si ricava:

$$u^* = -R^{-1} B^T \lambda \tag{5.16}$$

Sostituendo u^* all'interno dell'equazione dello Stato (5.10) e risolvendo l'equazione del *co*-stato $\dot{\lambda}(t) = -\dfrac{\partial H}{\partial x}^T$ otteniamo le seguenti equazioni differenziali:

$$\dot{x} = Ax - BR^{-1}B^T\lambda \tag{5.17}$$

$$\dot{\lambda} = -A^T\lambda \tag{5.18}$$

La soluzione dell'equazione differenziale 5.18 è:

$$\lambda(t) = e^{A^T(\tau - t)}\lambda(\tau) \tag{5.19}$$

in cui $\lambda(\tau)$ è ancora un valore sconosciuto.

Sostituendo la 5.19 in 5.17 otteniamo:

$$\dot{x} = Ax - BR^{-1}B^T e^{A^T(\tau - t)}\lambda(\tau) \tag{5.20}$$

la cui soluzione è:

$$x(t) = e^{A(t-t_0)}x(t_0) \; + \int_{t_0}^{t} e^{A(t-t')} B R^{-1} B^T e^{A^T(\tau - t')}\lambda(\tau) dt' \tag{5.21}$$

Per calcolare $\lambda(\tau)$ si deve valutare 5.21 per $t = \tau$, ottenendo così:

$$x(\tau) = e^{A(\tau - t_0)}x(t_0) \ - G(t_0, \tau)\lambda(\tau) \tag{5.22}$$

in cui

$$G(t_0, \tau) = \int_{t_0}^{\tau} e^{A(\tau - t')}BR^{-1}B^T e^{A^T(\tau - t')}dt' \tag{5.23}$$

è il Gramiano di controllabilità pesato dalla matrice R e valutato nell'istante finale τ.

Sostituendo nella condizione dello stato finale $x(\tau) = x_1$ l'equazione 5.22, si ottiene il *co*-stato valutato nell'istante finale:

$$\lambda(\tau) = -G^{-1}(t_0, \tau)[x_1 - e^{A(\tau - t_0)}x(t_0)] \tag{5.24}$$

Ora si può ottenere la funzione *co*-stato $\lambda(t)$ sostituendo semplicemente la 5.24 nella 5.19

$$\lambda(t) = -e^{A^T(\tau - t)}G^{-1}(t_0, \tau)[x_1 - e^{A(\tau - t_0)}x(t_0)] \tag{5.25}$$

Per concludere, la legge del controllo ottima si ricava dall'unione di 5.25 e 5.16

$$u^*(t) = R^{-1}B^T e^{A^T(\tau - t)}G^{-1}(t_0, \tau)[x_1 - e^{A(\tau - t_0)}x(t_0)] \tag{5.26}$$

5.3.2 Calcolo del tempo finale

Il calcolo del tempo ottimo deriva dalla minimizzazione della funzione di costo 5.11 valutata per $u(t) = u^*(t)$.

In pratica, sostituendo la 5.26 in 5.11 e eseguendo i calcoli, si ottiene:

$$c(\tau) = \tau + (x_1 - \bar{x}(\tau))^T G(\tau)^{-1}(x_1 - \bar{x}(\tau)) \tag{5.27}$$

dove $\bar{x}(\tau)$ è il *movimento libero*, ovvero il movimento dello stato che si avrebbe in mancanza di ingressi e che quindi dipende solo dallo stato iniziale:

$$\bar{x}(t) = e^{At}x_0 \tag{5.28}$$

Come si può notare, l'equazione 5.27 è una funzione nella sola variabile τ. Il tempo finale ottimo τ^* è dunque il valore che minimizza tale cifra di merito:

$$\tau^* = argmin\{\tau > 0\} \ c(\tau) \tag{5.29}$$

Assumendo che il sistema sia controllabile, il Gramiano sarà una matrice definita positiva per ogni $t > 0$. Per questa ragione, nell'equazione 5.27, il secondo addendo sarà sempre non-negativo e quindi $c(\tau) > \tau$ per ogni $\tau > 0$. Di conseguenza, per ricavare il tempo finale ottimo basterà trovare il valore τ per cui si verifica: $\dot{c}(\tau) = 0$.

Una volta ottenuto il tempo finale ottimo τ^*, non ci saranno più incognite nella legge del controllo $u^*(t)$ dell'equazione 5.26. Applicando quindi tale ingresso al sistema dinamico dell'equazione 5.10, si otterrà la funzione variabile di stato $x(t)$ che descrive la traiettoria ottima di congiunzione tra il nodo x_0 e x_1.

5.4 Kinodynamic-RRT*

L'Algortimo RRT* riportato e analizzato nel capitolo precedente, si basava sull'assunzione di poter connettere i nodi dell'Albero tramite semplici segmenti. In particolare la procedura di congiunzione tra il nodo considerato come *padre* e quello considerato come *figlio* veniva effettuata dalla `Steering function`.

Nel caso di **Kinodynamic-RRT***[11] quello che cambia rispetto a RRT* è essenzialmente l'utilizzo di una *Steering function* differente, la quale crea le connessioni sulla base dei calcoli relativi al controllo ottimo.

Nell'algoritmo 8 viene riportata la struttura di Kinodynamic-RRT*. Si noti che l'albero dei cammini viene identificato dalla notazione T.

Algorithm 8 Kinodynamic-RRT*

1: $T \leftarrow \{q_{start}\}$;
2: **for** $i = 1, \cdots,$ **maxNodes do**
3: $q_{rand} \leftarrow$ `SampleFree`;
4: $q_{nearest} \leftarrow argmin\{q \in T \mid c^*(q_{nearest}, q_{rand}) < r \wedge$ `CollisionFree` $(\pi^*(q_{nearest}, q_{rand}))\}(\texttt{cost}(q_{nearest}) + c^*(q_{nearest}, q_{rand}))$
5: $\texttt{parent}(q_{rand}) \leftarrow q_{nearest}$
6: $\texttt{cost}(q_{rand}) \leftarrow \texttt{cost}(q_{nearest}) + c^*(q_{nearest}, q_{rand})$
7: **for all** $q_i \in T \cup \{q_{goal}\} \mid c^*(q_{rand}, q_i) < r \wedge \texttt{cost}(q_{rand}) + c^*(q_{rand}, q) < \texttt{cost}(q_i) \wedge \texttt{CollisionFree}(\pi^*(q_{rand}, q_i))$ **do**
8: $\texttt{cost}(q_i) \leftarrow \texttt{cost}(q_{rand}) + c^*(q_{rand}, q_i)$;
9: **end for**
10: **end for**
11: **return** $T \leftarrow T \cup \{q_{rand}\}$

- **Parametri**
  - **maxNodes**: Parametro che identifica il numero finale di nodi contenuti in T.

- **r**: E' il parametro che sostituisce *discRadius* in RRT* per la fase di Rewiring di T. A differenza di quest'ultimo, Kinodynamic-RRT* non considera il costo di connessione calcolando la distanza euclidea tra i nodi, ma lo fa valutando la funzione di costo per la traiettoria ottima ipotetica di congiunzione. Il parametro **r** viene usato per trovare tutti i nodi vicini a quello generato casualmente, ovvero quelli che per essere raggiunti partendo da q_{rand} impiegano un costo inferiore alla soglia **r**. Nella pratica però, questa procedura risulta molto dispendiosa in termini computazionali, in quanto bisognerebbe ad ogni *Rewiring* di T calcolare la traiettoria ottima $\pi^*(q_{rand}, q_i)$ per ogni nodo $q_i \in T$, per poi scartare quelli con costo maggiore a **r**.

  Una strategia che si può adottare per evitare eccessivi sforzi computazionali è quella di utilizzare la distanza Euclidea come criterio per trovare i nodi vicini a q_{rand}, ma connettere poi il nodo migliore tramite la teoria del controllo ottimo. Questa soluzione ibrida abbassa notevolmente i tempi di computazione, ma sconvolge radicalmente la logica di creazione dell'Albero. Infatti, due nodi molto distanti tra loro non è detto che non possano essere connessi da una traiettoria di basso costo dato che la cifra di merito penalizza soprattutto i cambi direzionali e non i tempi di percorrenza.

- **A,B**: Sono le matrici del sistema dinamico lineare preso in considerazione. Tali matrici andranno poi inserite come argomenti nella `Steering function` affinché essa possa conoscere i vincoli dinamici del modello e calcolare la traiettoria ottima.

- **Funzioni**

  - `SampleFree`: la funzione genera, come nei casi precedenti, un nodo casuale in C_{free}. Anche in questo caso è possibile sostituirla alla funzione `SampleFree-toGoal`, la quale genera con una frequenza costante un nodo all'interno della regione di Goal.
  - `CollisionFree`: per i dettagli di questa funzione rimandiamo il lettore al Capitolo 2.
  - $c^*(q_1, q_2)$: la funzione calcola per prima cosa la traiettoria ottima $\pi^*(q_1, q_2)$ con il metodo del controllo ottimo, in seguito restituisce la funzione di costo valutata per tale traiettoria.
  - `cost`(q): calcola il costo totale della traiettoria da q_{start} a q.
  - `Steering`: Nell'algoritmo 8 vi è un richiamo implicito alla *Steering function* la quale funziona come pianificatore locale.

Infatti, come si può notare nelle righe (4),(6) e (7), viene più volte richiesto il costo della traiettoria c^* o la traiettoria ottima stessa π^*. Il pianificatore locale in questo caso dovrà connettere i due nodi applicando la teoria del controllo ottimo sopra riportata.

5.4.1 Modello utilizzato

Come descritto nel paragrafo 5.2, il controllo ottimo prevede la conoscenza del sistema attraverso la stesura dell'equazione di stato lineare 5.10.

In questo caso si è deciso di utilizzare il modello dell'Uniciclo riportato nel paragrafo 2.1:

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = \omega \\ \dot{v} = a \end{cases}$$

Come si può notare il sistema è non lineare a causa della presenza delle due funzioni trigonometriche. Per poter applicare la teoria descritta precedentemente sarà quindi necessario ricondursi ad un modello equivalente lineare.

Per la linearizzazione si è deciso di adottare l'approccio presentato nell'articolo *Poli-RRT*: optimal RRT-based planning for constrained and feedback linearisable vehicle dynamics*[3].

Tale tecnica consiste nell'applicare le seguenti trasformazioni:

$$\begin{bmatrix} a \\ \omega \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -\dfrac{sin(\theta)}{v} & \dfrac{cos(\theta)}{v} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{5.30}$$

Sostituendo quindi le equazioni del sistema 5.30 al sistema non lineare, otteniamo il sistema linearizzato:

$$\begin{cases} \dot{x} = V_x \\ \dot{y} = V_y \\ \dot{v_x} = u_1 \\ \dot{v_y} = u_2 \end{cases}$$

in cui si definisce $v_x = v\cos(\theta)$ e $v_y = v\sin(\theta)$. Definendo il vettore delle variabili di stato $= [x, y, v_x, v_y]^T$, quello delle variabili di controllo $u = [u_1, u_2]^T$ e le matrici A e B come segue:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.31}$$

$$B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{5.32}$$

si potrà descrivere la dinamica del sistema attraverso la seguente equazione matriciale:

$$\dot{x} = Ax + Bu \tag{5.33}$$

5.4.2 Implementazione

L'algoritmo 8 risulta abbastanza inadatto ad un'implementazione pratica data la complessità elevata dal punto di vista computazionale. Per questo si è deciso di adottare alcune strategie in modo da poter alleggerire il carico computazionale mantenendo però la stessa struttura di base dell'algoritmo:

1. Per prima cosa si è deciso di implementare un codice specifico per il modello descritto in 5.33 svolgendo il maggior numero di calcoli a mano in modo da alleggerire dove possibile il carico computazionale del compilatore. Si noti infatti che nel sistema preso in considerazione la matrice A è nilpotente e che quindi i calcoli matriciali risultano abbastanza accessibili ad una computazione manuale.

   Di seguito vengono riportati i passi seguiti per l'implementazione della `Steering function` la quale genera la traiettoria ottima tra i due nodi x_0 e x_1:

   (a) Innanzitutto si è deciso di definire la matrice dei pesi come matrice diagonale:

$$R = \begin{bmatrix} r_1 & 0 \\ 0 & r_2 \end{bmatrix} \tag{5.34}$$

(b) E' stato quindi calcolato il Gramiano pesato di controllabilità (5.23). Invertendo la matrice il risultato ottenuto sarà:

$$G^{-1}(t) = \begin{bmatrix} \frac{12r_1}{t^3} & 0 & -\frac{6r_1}{t^2} & 0 \\ 0 & \frac{12r_2}{t^3} & 0 & -\frac{6r_2}{t^2} \\ -\frac{6r_1}{t^2} & 0 & \frac{4r_1}{t} & 0 \\ 0 & -\frac{6r_2}{t^2} & 0 & \frac{4r_2}{t} \end{bmatrix} \tag{5.35}$$

(c) Il movimento libero dell'equazione 5.28, in questo caso sarà:

$$\bar{x}(t) = \begin{bmatrix} x_0 + V_{x_0} t \\ y_0 + V_{y_0} t \\ V_{x_0} \\ V_{y_0} \end{bmatrix} \tag{5.36}$$

in cui $[x_0, y_0, V_{x_0}, V_{y_0}]^T$ è lo stato al tempo iniziale t_0.

(d) A questo punto si è risolta l'equazione 5.27, in modo da ottenere la funzione di costo $c(\tau)$ nella sola variabile τ.

(e) Attraverso la funzione `fminunc`$(c(\tau), \tau_0)$ presente nel software *Matlab*, è possibile calcolare il valore τ^* che minimizza la funzione $c(\tau)$. Quest'operazione viene effettuata all'interno di una *Matlab-function* chiamata `optimal-final-time`, mentre `optimal-cost` valuta il costo relativo a τ^*.

(f) In *Matlab* vi è la possibilità di creare un modello che simula il sistema dinamico lineare. Per fare questo bisogna prima creare l'oggetto *sys* di tipo `ss` passando come ingressi le matrici A, B, C e D che descrivono il sistema:

$$\begin{cases} \dot{x} = Ax + Bu \\ \dot{y} = Cx + Du \end{cases}$$

L'oggetto `sys` permette di utilizzare il metodo `lsim`, il quale restituisce le variabili d'uscita $y(t)$ simulando la risposta del sistema all'ingresso $u^*(t)$ calcolato dall'equazione 5.26.

In questo caso la variabile delle uscite coincide con la variabile di stato, la quale a sua volta rappresenta l'evoluzione di tutti i movimenti del robot nella traiettoria che lo conducono allo stato finale x_1 partendo da quello iniziale x_0.

La `Steering function` quindi non fa altro che ricevere in ingresso x_0, x_1 *e* τ^*, calcolare $u^*(t)$, simulare il comportamento del sistema tramite il comando $x(t) =$`lsim`(sys, u^*, t, x_0) (dove $t \in [0, \tau^*]$) e restituire l'intera traiettoria $x(t)$ risultante.

2. La seconda semplificazione riguarda la funzione `Near`. Tale funzione svolge il compito di trovare tutti i nodi appartenenti all'albero nelle immediate vicinanze del nodo q_{rand} passato in ingresso.

   In una logica in cui non si considera la distanza Euclidea come peso da assegnare agli archi, ma una cifra di merito $c(\pi)$, `Near` seleziona solo i nodi $q_i \in V$ la cui traiettoria locale $q_{rand} \rightarrow q_i$ abbia un costo inferiore ad una certa soglia r. Questo implica inevitabilmente che ad ogni iterazione dell'algoritmo si debbano calcolare tutte le traiettorie di connessione tra il nodo appena generato e i nodi di V con il fine di valutarne i costi. Il carico di calcoli da eseguire dunque crescerà in modo proporzionale alla cardinalità di V. Per alleggerire il codice si è deciso di applicare una logica ibrida tra le due strategie:

   Il nodo $q_{nearest}$, ovvero il nodo più vicino a q_{rand}, viene trovato attraverso il calcolo della cifra di merito 5.11. Invece, per quanto riguarda la fase di Rewiring dell'Albero in cui si ottimizzano le connessioni nei dintorni di q_{rand}, vengono considerati solo i nodi con distanza euclidea da q_{rand} inferiore ad una soglia. Questo valore limite viene calcolato con la stessa logica con cui veniva calcolato in RRT*.

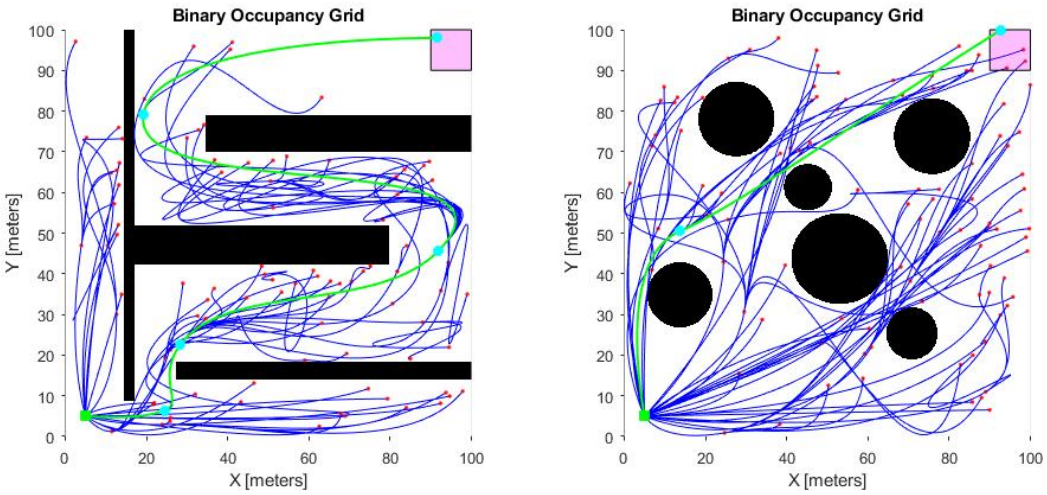
A titolo dimostrativo in figura 5.1 vengono riportate le simulazioni per cento nodi, in due ambienti differenti.

Si noti che non sono state imposte limitazioni sulle variabili di controllo e quindi, nonostante l'algoritmo trovi soluzioni più lineari possibili, può accadere che alcune traiettorie impongano movimenti non realizzabili dal robot:

5.5 Kinodynamic-PRM*

L'algoritmo **Kinodynamic-PRM*** ha la stessa struttura dell'algoritmo PRM* 3. La differenza sostanziale riguarda l'uso di una *Steering function* differente, ovvero la stessa utilizzata per *Kinodynamic-RRT**.

Vi è inoltre da aggiungere che l'introduzione delle variabili di stato V_x e V_y impone una piccola modifica nella fase dell'algoritmo in cui si aggiunge un arco all'insieme E. Infatti, come si può facilmente intuire, se ad ogni nodo vengono associate le componenti cartesiane della velocità, la traiettoria locale che connette due nodi non sarà uguale nei due versi di percorrenza (figura 5.2). Questo fatto



(a) (b)

Figura 5.1: Kinodynamic-RRT*

viene introdotto nell'algoritmo valutando separatamente la fattibilità per i due versi di percorrenza.

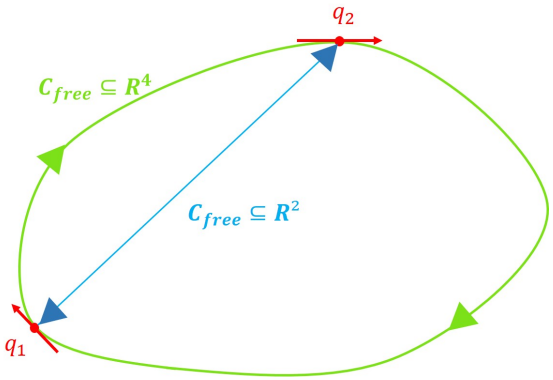


Figura 5.2: Nel caso in cui i nodi sono rappresentati da vettori $[x, y, V_x, V_y] \in R^4$ le traiettorie cambiano in base al verso di percorrenza.

La definizione dei parametri all'interno dell'algoritmo è uguale a quella dei casi precedenti.

Per quanto riguarda le funzioni utilizzate, vi è da sottolineare che nell'implementazione pratica, textttNear si basa sul calcolo della cifra di merito e non sulla distanza Euclidea. Dato il nodo v, tale funzione trova l'insieme dei nodi q_{near} tali che $c^*(\pi(v, q_{near})) < treshold$. La scelta del valore di limite di costo $treshold$ sarà arbitraria in base all'applicazione specifica.

Algorithm 9 Kinodynamic-PRM*: Fase di Learning

$V \leftarrow \{q_{start}\} \cup [SampleFree_i]_{i=1,\cdots,n}$; $E \leftarrow \emptyset$;
for all $v \in V$ **do**
  $Z_{near} \leftarrow$`Near`$(G = (V, E), v, \gamma_{PRM} \left(\frac{log(n)}{n}\right)^{\frac{1}{d}})\backslash\{v\}$;
  for all $q_{near} \in Znear$ **do**
    if `CollisionFree`$(q_{near}$,v$)$ **then**
      $E \leftarrow E \cup \{(q_{near}, v)\}$
    end if
  end for
end for
return $G = (V, E)$;

Si ricordi che la funzione di costo 5.11 penalizza sia il tempo impiegato che lo sforzo delle variabili di controllo, per questa ragione verranno selezionato le traiettorie più conservative, ovvero quelle che non richiedono eccessivi cambi direzionali.

Per la scelta del valore *treshold* si è deciso di muoversi seguendo una logica basata su dati empirici:

Per prima cosa si è scelto un valore di soglia molto alto in modo da favorire tutte le connessioni tra i nodi. Così facendo si è potuto calcolare il costo medio di tutte le traiettorie ottenute (figura 5.3):

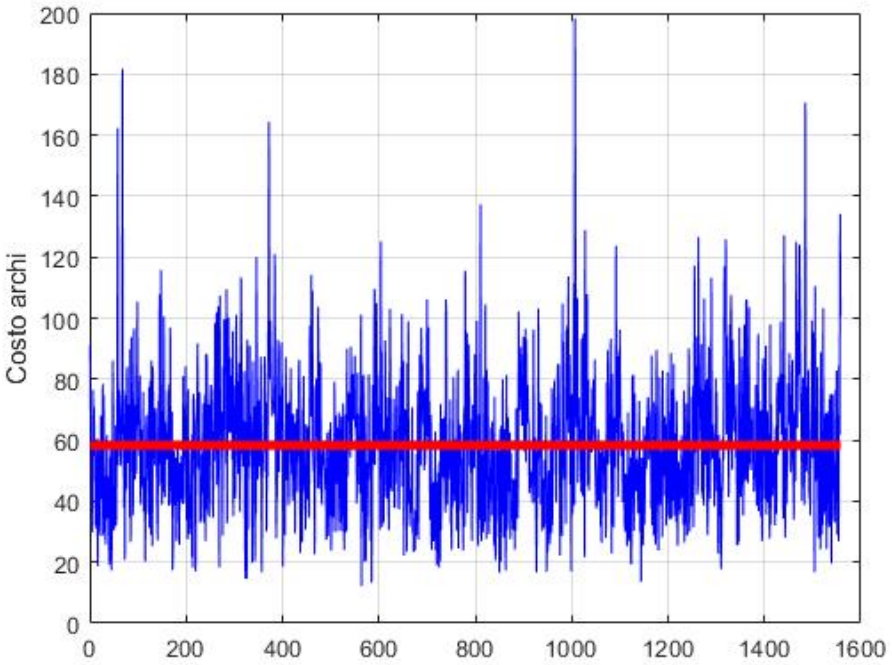
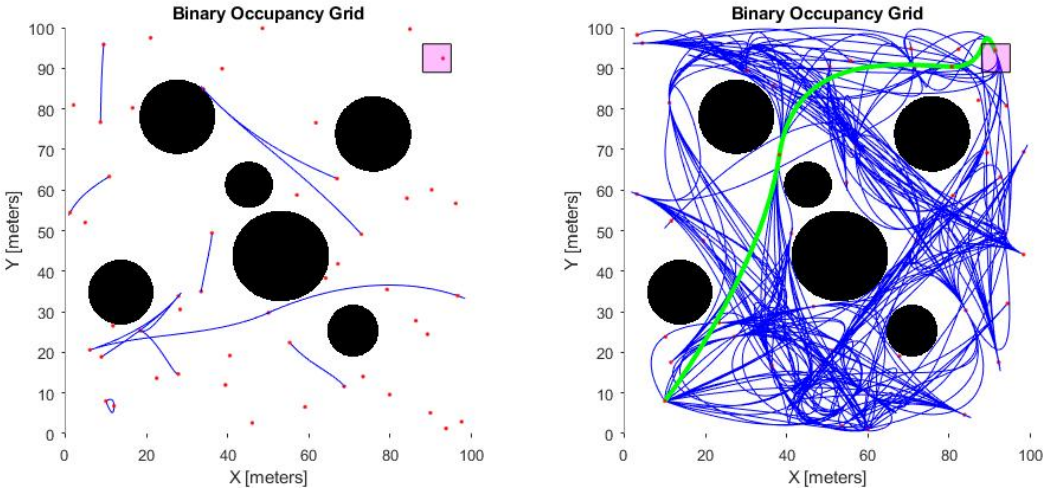


Figura 5.3: Media dei costi di tutti gli archi

Avendo ora un termine di paragone si è deciso di condurre una simulazione per un valore molto basso di *treshold*. In questo modo è stato possibile notare come la cifra di merito favorisca le traiettorie più lineari, ovvero quelle che conservano maggiormente la direzione del moto.



(a) $soglia = 20$ (b) $soglia = 60$

Figura 5.4: Kinodynamic-PRM*

Il risultato finale di una simulazione e riportato in 5.4 si può ritenere soddisfacente in termini di dispendio energetico. Si noti che la traiettoria finale è decisamente lineare e in questo modo sia ha una minima sollecitazione delle variabili di controllo.

Capitolo 6

Simulazioni e risultati

In questo capitolo verranno analizzati gli algoritmi finora citati riportando i risultati ottenuti nelle simulazioni. Lo scopo fondamentale è quello di valutare la qualità delle soluzioni ottenute nei diversi casi, variando i parametri fondamentali da inserire nei codici.

Gli algoritmi citati nella prima parte della tesi considerano la distanza Euclidea come costo della traiettoria ottenuta, mentre quelli successivi valutano una specifica funzione di costo. Per questa ragione si è deciso di svolgere separatamente le simulazioni per le due differenti classi di algoritmi.

Vi è da aggiungere inoltre che per ogni simulazione sono stati scelti i parametri e gli ambienti in modo da evidenziare le caratteristiche di interesse.

6.1 Algoritmi basati sulla distanza Euclidea

6.1.1 Probabilistic RoadMap

In questo paragrafo si effettua un confronto tra gli algoritmi della classe Probabilistic RoadMap.

La priorità è quella di analizzare la qualità delle soluzioni ottenute al crescere del numero di iterazioni. Per fare questo è stato necessario scegliere un ambiente non troppo restrittivo, in cui gli ostacoli permettano agli algoritmi di trovare numerose soluzioni differenti.

La mappa costruita è riportata in figura 6.1. Come si può notare si è optato per un labirinto non eccessivamente complesso.

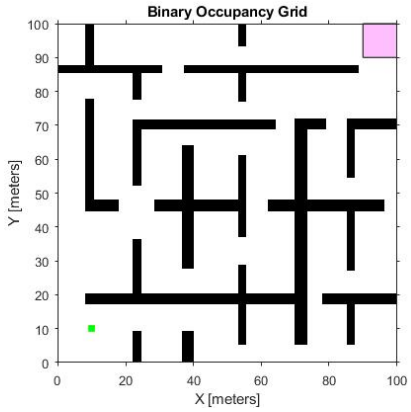


Figura 6.1: **Ambiente 1**

Nell'immagine viene rappresentata in nero la regione occupata dagli ostacoli, in rosa la regione di Goal e in verde lo stato iniziale del robot. Essendo gli algoritmi *Sampling-based* logiche basate sulla probabilità, è stato necessario condurre un vasto campione di simulazioni per comprenderne il comportamento generale.

Confronto tra PRM e sPRM

Per PRM e sPRM i parametri più importanti da scegliere sono: il numero di nodi della RoadMap e il raggio dell'area entro cui vengono permesse le connessioni tra i nodi.

Il raggio è stato fissato ad un valore unico pari a venti metri mentre il numero di nodi è stato incrementato gradualmente. Per osservare nel modo corretto il miglioramento delle traiettorie pianificate è stato necessario eseguire il codice per un numero crescente di nodi sulla stessa RoadMap.

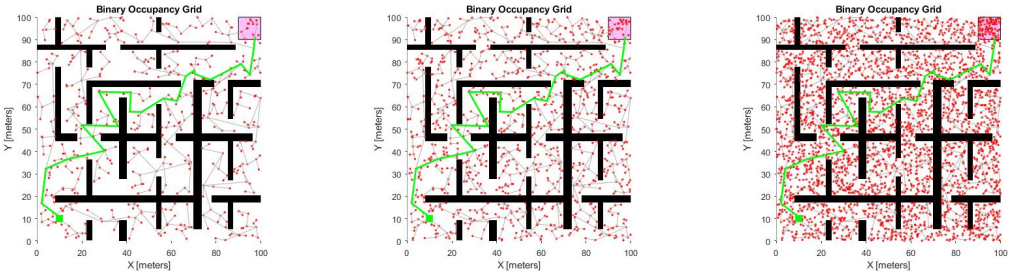


Figura 6.2: **PRM** per 500, 1000 e 3000 iterazioni

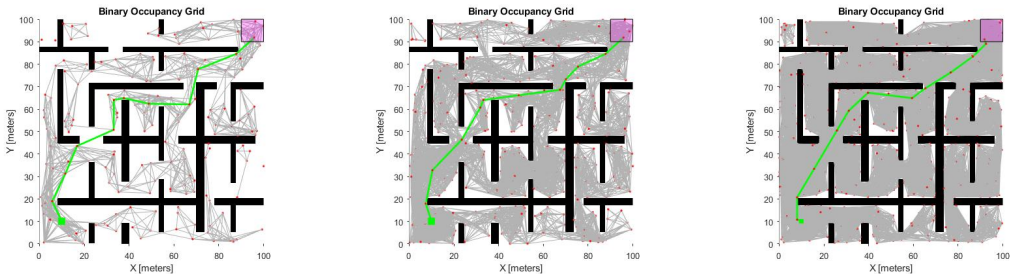


Figura 6.3: **sPRM** per 500, 1000 e 3000 iterazioni

I risultati rendono evidente la superiorità dell'algoritmo **sPRM** in termini di efficienza rispetto a **PRM**. Infatti **sPRM**, già per 1000 nodi, riesce a pianificare traiettorie molto simili a quella ottima mentre PRM mantiene invariata la soluzione inizialmente trovata.

La causa di questo è da attribuire al fatto che PRM (algoritmo 1) aggiunge gli archi all'insieme E verificando che questi non creino cicli all'interno della RoadMap. Per questo, dal momento in cui si crea una prima traiettoria, l'algoritmo è più in grado di trovare "scorciatoie" che accorcino il percorso. Proprio l'introduzione dei cicli rende migliore sPRM, il quale nella fase finale di *Quering* si trova ad avere un numero maggiore di traiettorie possibili tra cui scegliere. Si può affermare dunque che PRM è un algoritmo più statico rispetto a sPRM dato che evolve in maniera poco consistente all'aumentare delle iterazioni.

In figura 6.4 vengono riportati i grafici dei costi delle traiettorie pianificate per un numero crescente di nodi. Da questo grafico risulta evidente come la traiettoria pianificata da PRM rimane invariata per tutto il corso della simulazione.

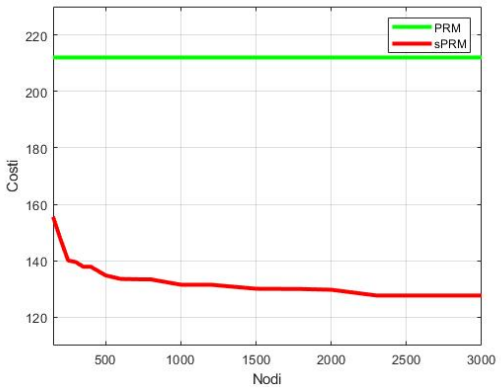


Figura 6.4: Evoluzione dei costi

PRM*

PRM* non è altro che un'evoluzione dell'algoritmo sPRM. Infatti l'unica differenza che lo contraddistingue da sPRM è la logica con cui viene fissato il parametro **r**, ovvero il raggio entro cui vengono connessi i nodi. Se in PRM veniva definito a discrezione dell'utente come parametro costante, in PRM* esso dipende dal numero finale di nodi **n** della RoadMap. La scelta della funzione **r(n)** è la stessa utilizzata nell'articolo *Sampling-based algorithms for optimal motion planning*[1] che, come viene dimostrato, rende ottimo l'algoritmo PRM*.

6.1.2 Rapidly-exploring Random Tree

Confronto tra RRT e RRT*

I due algoritmi analizzati in questo paragrafo, ovvero RRT e RRT* appartengono alla classe degli algoritmi Rapidly-exploring Random Tree. Come si può intuire dal nome, questi due algoritmi prevedono una progressiva espansione di una struttura chiamata Albero, radicata nel nodo di partenza q_{start}.
La differenza sostanziale tra i due è che nel caso di RRT*, come è stato spiegato nel Capitolo 4, l'Albero delle connessioni viene ottimizzato ad ogni iterazione grazie alla fase di *Rewiring*. Proprio questa fase rende l'algoritmo RRT* molto versatile, permettendo in questo modo dei continui miglioramenti della traiettoria al progredire delle iterazioni. In RRT invece, tale fase non è prevista e risulta poco probabile un miglioramento della traiettoria inizialmente trovata.

Per visualizzare meglio questo fatto si è deciso di condurre delle simulazioni nell'ambiente di figura 6.5.

Questa mappa è stata scelta in modo tale da permettere agli algoritmi di trovare inizialmente una traiettoria poco conveniente in termini di spazio percorso (zona gialla), e una più economica in seguito (zona arancione).

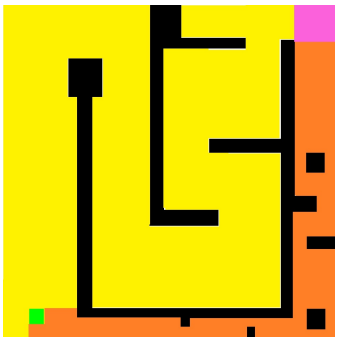
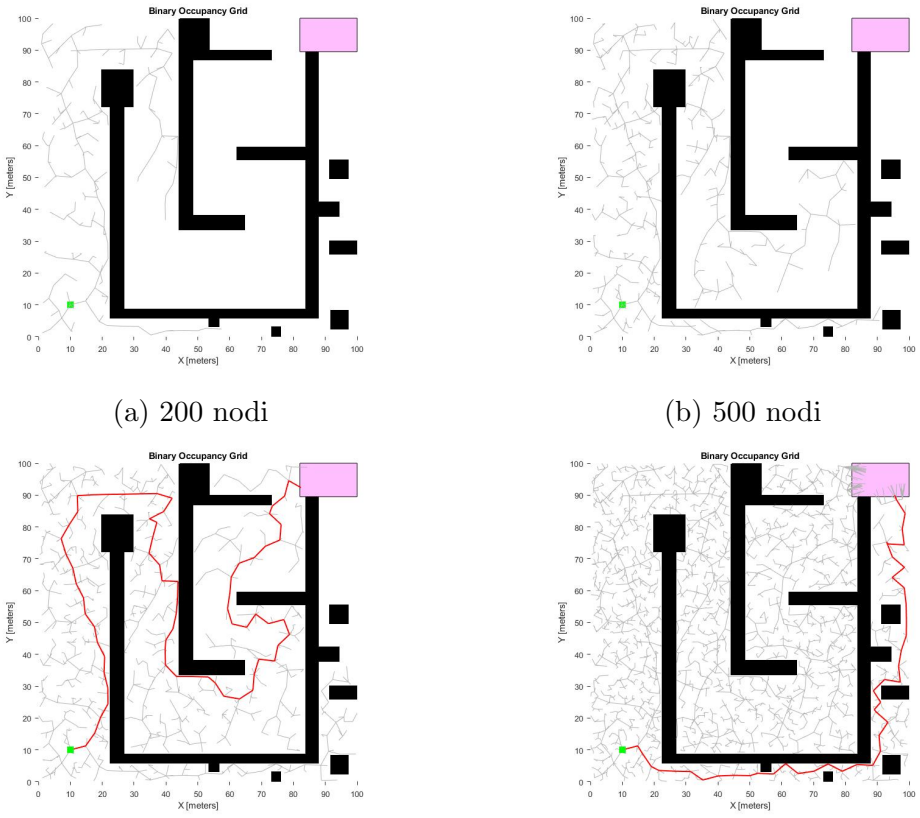


Figura 6.5: **Ambiente 2**

Sono state eseguite diverse simulazioni all'aumentare del numero di nodi e fissando il parametro η a cinque metri. Nelle figure 6.6 e 6.7 vengono riportati due esempi particolari.



(a) 200 nodi

(b) 500 nodi

(c) 1000 nodi

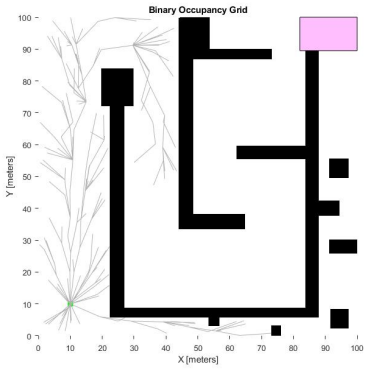
(d) 3000 nodi

Figura 6.6: RRT

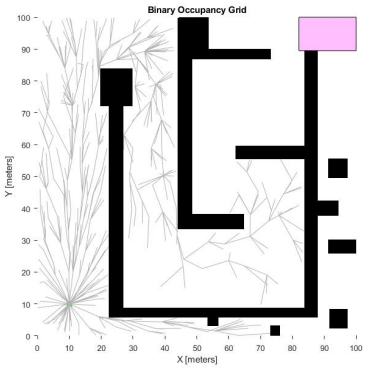
Quello che si può dedurre è che l'aggiunta della fase di *Rewiring* nell'algoritmo RRT* rende molto meno dispersivo l'albero delle traiettorie (figura 6.7). La differenza tra i due algoritmi si vede chiaramente, infatti i nodi dell'albero generato da RRT sono spesso raggiunti da traiettorie poco economiche, mentre in RRT* i costi delle traiettorie sono sempre più ottimizzati all'aumentare del numero di iterazioni.

In figura 6.8 è possibile notare la staticità dell'algoritmo RRT. Infatti, una volta trovata una prima grezza soluzione non riesce a migliorarla fino a quando il secondo ramo principale dell'Albero non percorre l'intera zona arancione di figura 6.5.

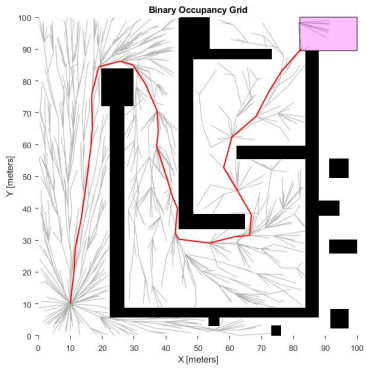
RRT* invece risulta decisamente più dinamico in quanto all'aumentare del numero di iterazioni calcola traiettorie di costi sempre minori.



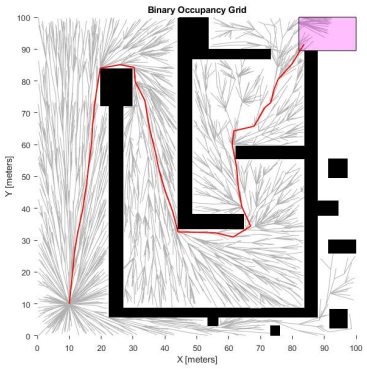
(a) 200 nodi



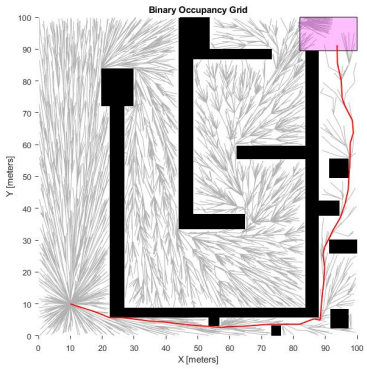
(b) 500 nodi



(c) 1000 nodi



(d) 3000 nodi



(e) 5000 nodi

Figura 6.7: RRT*

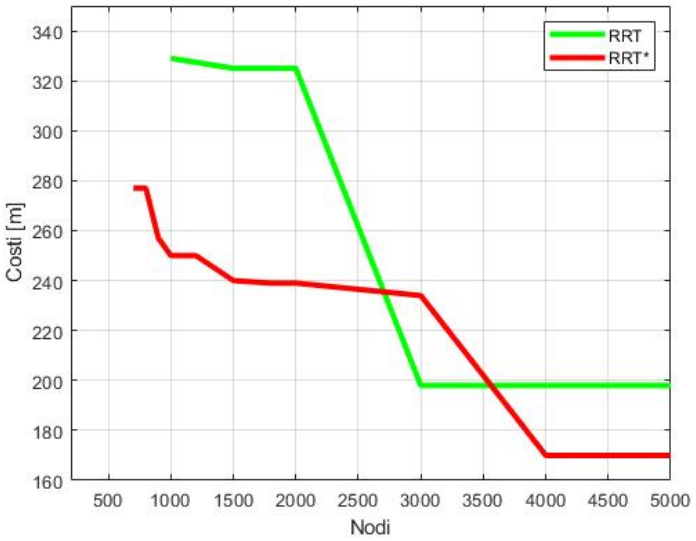
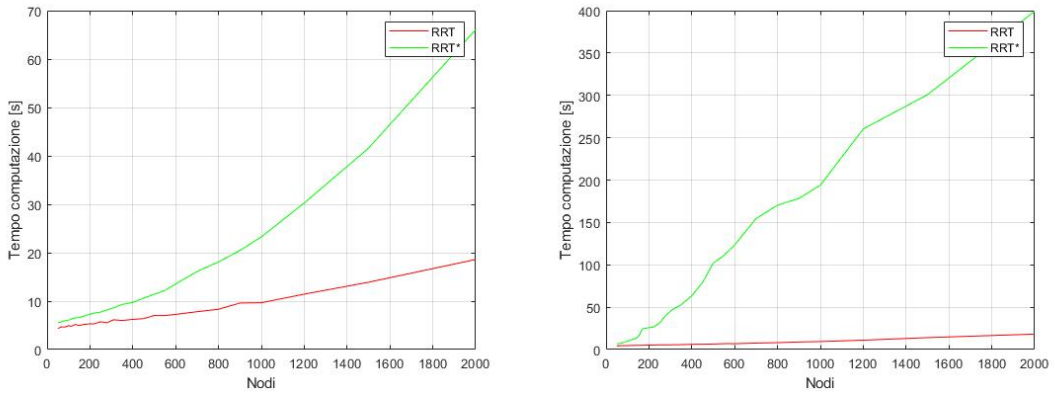


Figura 6.8: Costi delle traiettorie pianificate da RRT e RRT*

L'ottimizzazione dell'albero introdotta in RRT* ha però un notevole svantaggio nel tempo di computazione del codice. Sono state quindi eseguite ulteriori simulazioni con lo scopo di analizzare tale aumento.

Grazie ad una funzione specifica dell'ambiente *Matlab* è stato possibile salvare i tempi di esecuzione e riportarli graficamente in figura 6.9.



(a) $\eta = 5$ *metri* (b) $\eta = 20$ *metri*

Figura 6.9: Tempi Computazionali

Come si può notare, il divario tra i tempi computazionali di RRT e RRT* aumenta proporzionalmente al raggio η. Ricordiamo infatti che η rappresenta il raggio massimo dell'area entro la quale vengono ottimizzati gli archi in RRT*. Per questa ragione, al suo incremento, corrisponde un aumento del numero di archi da ottimizzare ad ogni iterazione, con la conseguenza di un incremento nei tempi d'esecuzione.

6.1.3 Confronto tra PRM* e RRT*

Dopo aver dimostrato empiricamente la superiorità degli algoritmi PRM* e RRT* rispetto alle loro versioni base PRM e RRT, si è voluto effettuare un loro confronto analizzando i risultati di ulteriori simulazioni.

Per questa fase è stato scelto un ambiente più complesso dei precedenti. Si è voluto infatti rendere la configurazione degli ostacoli più stringente in modo da osservare il comportamento dei due algoritmi per situazioni estreme. L'ambiente viene riportato in figura 6.10.

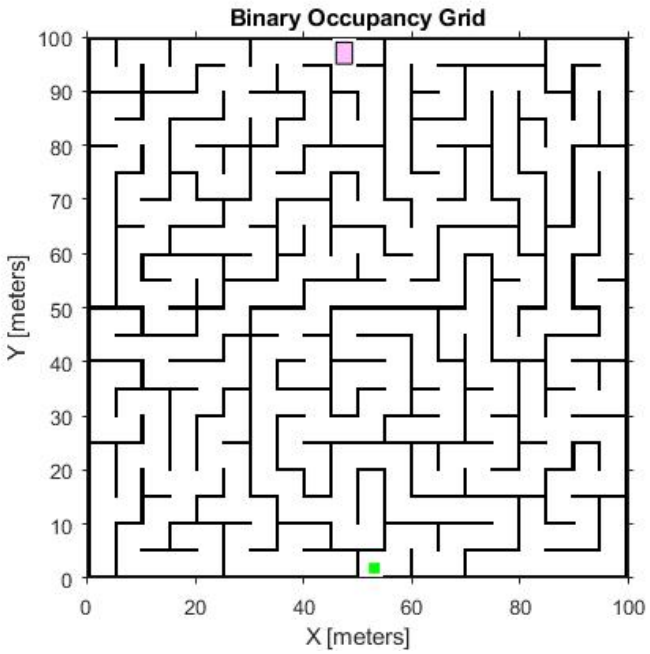
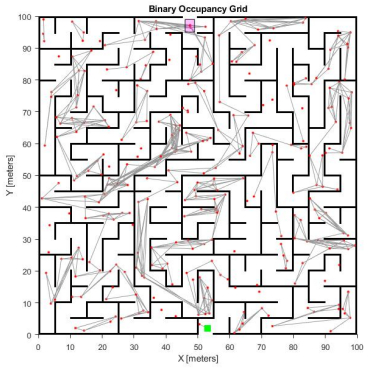


Figura 6.10: **Ambiente 3**

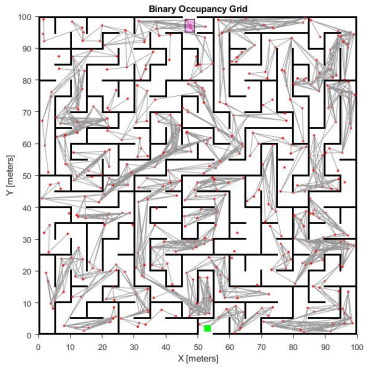
Un fatto significativo rivelato da queste simulazioni e riportato nel grafico di figura 6.13, è che PRM* trova la soluzione in tempi notevolmente più brevi rispetto a RRT*.

Questo fatto si accentua per ambienti complessi come questo. In effetti, per configurazioni in cui il robot è costretto a passare attraverso strette corsie, RRT* espande l'Albero solo se il nodo generato appartiene alla direzione del passaggio. Per questa ragione l'albero progredisce in maniera tanto lenta quanto strette sono le corsie del labirinto. PRM* invece genera i nodi coprendo con probabilità uniforme tutti gli spazi liberi e, dopo averli connessi, trova (se possibile) la traiettoria finale.

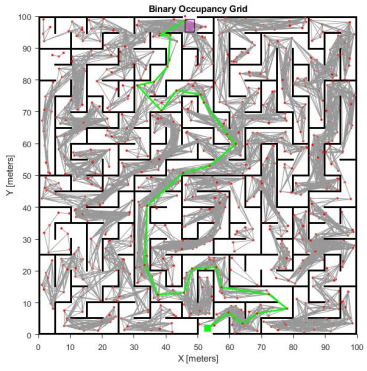
Inoltre vi è da aggiungere che in questo caso la mappa non lascia molto margine di miglioramento della soluzione trovata, ed essendo RRT* improntato su una continua ottimizzazione della traiettoria, esso risulta meno efficiente per questi ambienti.



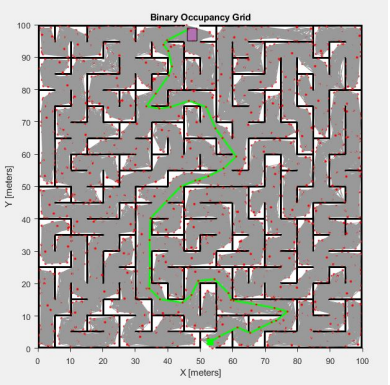
(a) 300 nodi



(b) 500 nodi

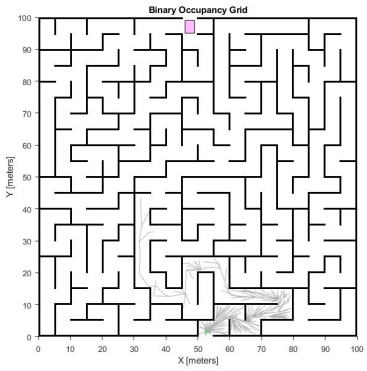


(c) 1000 nodi

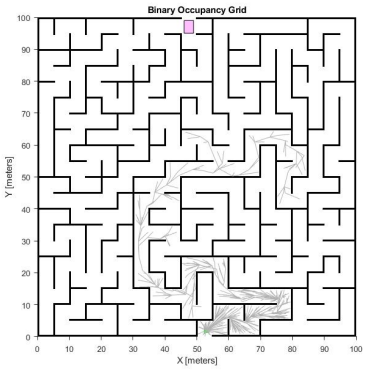


(d) 5000 nodi

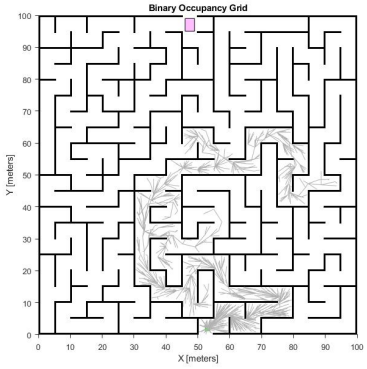
Figura 6.11: PRM*



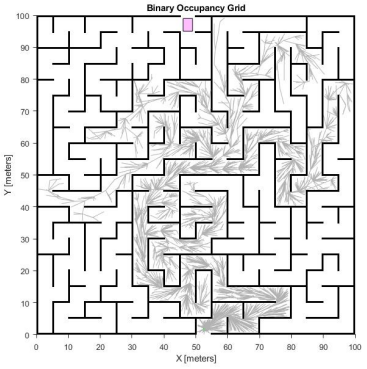
(a) 300 nodi



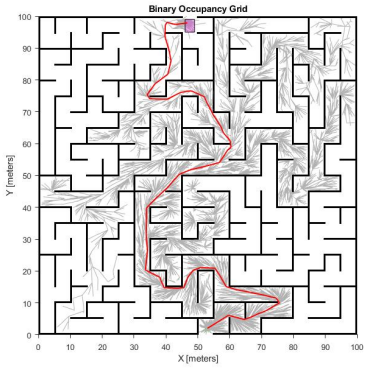
(b) 500 nodi



(c) 1000 nodi



(d) 3000 nodi



(e) 5000 nodi

Figura 6.12: RRT*

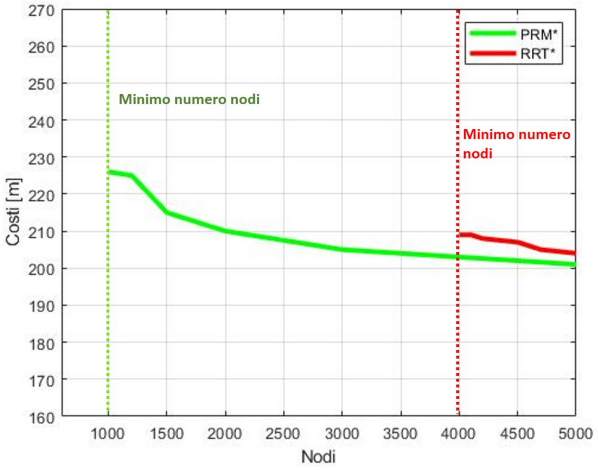


Figura 6.13: Evoluzione dei costi delle traiettorie trovate da **PRM*** e **RRT***

Le simulazioni successive sono state eseguite in un ambiente più favorevole all'algoritmo RRT*. Come si può notare dalla figura 6.14, l'ambiente è stato suddiviso in stanze separate da linee che rappresentano dei muri. Tale ambiente è stato creato appositamente per evidenziare uno svantaggio dell'algoritmo PRM*. Infatti, sebbene PRM* esplori l'intero spazio delle configurazioni in breve tempo, in questo caso non è conveniente trovare dei nodi all'interno delle stanze dato che la traiettoria ottima è evidentemente quella che percorre il corridoio principale.

Quello che si può facilmente notare dalle figure 6.15 e 6.16 è che RRT* trova la prima soluzione per un numero di iterazioni minore rispetto a quanto faccia PRM*. D'altro canto però, nel grafico dei costi di figura 6.17 si vede come le traiettorie pianificate da PRM* migliorino più rapidamente.

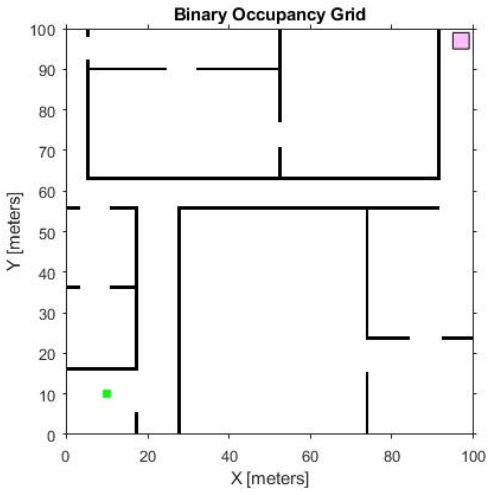
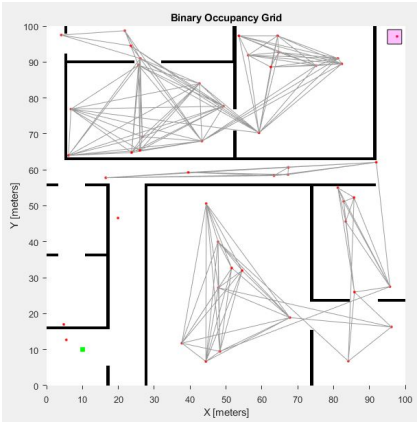
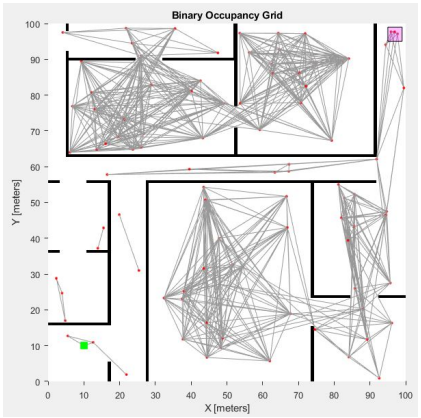


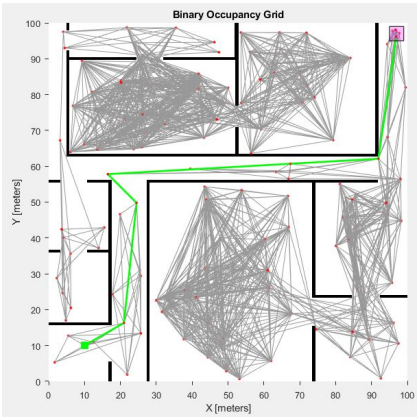
Figura 6.14: **Ambiente 4**



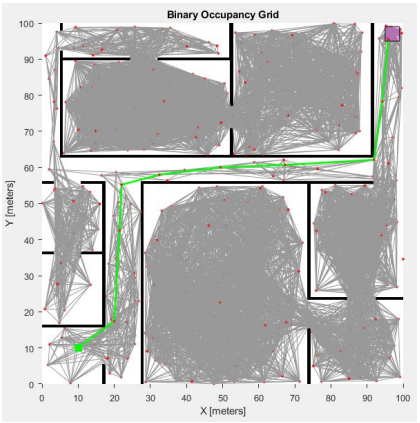
(a) 50 nodi



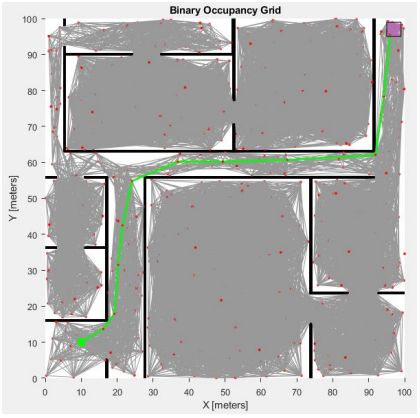
(b) 100 nodi



(c) 150 nodi

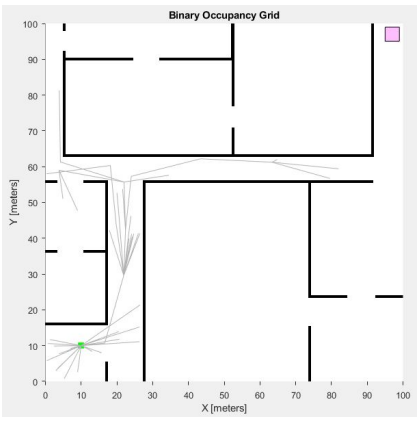


(d) 500 nodi

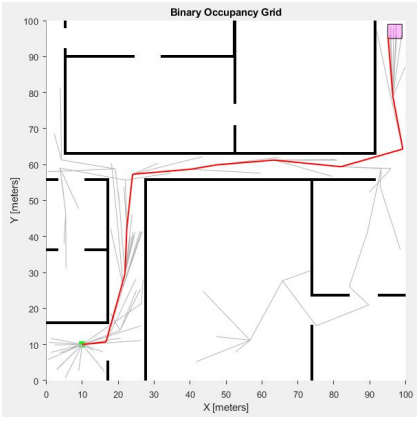


(e) 1000 nodi

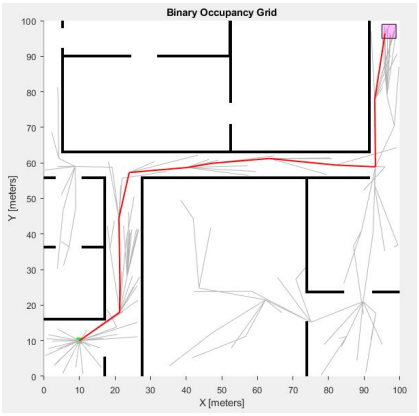
Figura 6.15: PRM*



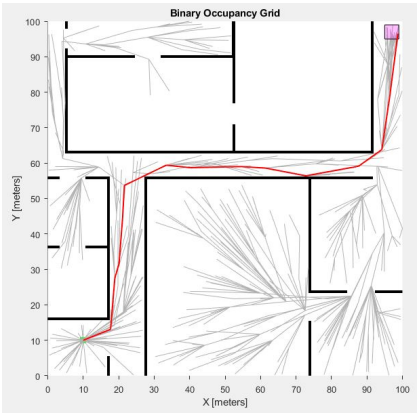
(a) 50 nodi



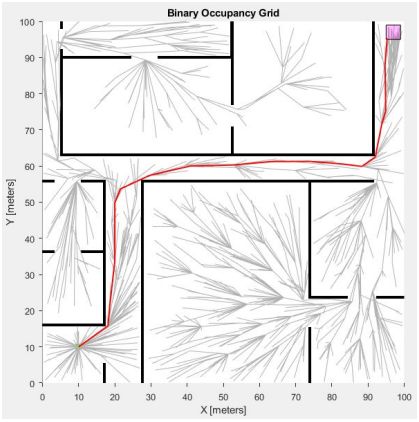
(b) 100 nodi



(c) 150 nodi



(d) 500 nodi



(e) 1000 nodi

Figura 6.16: RRT*

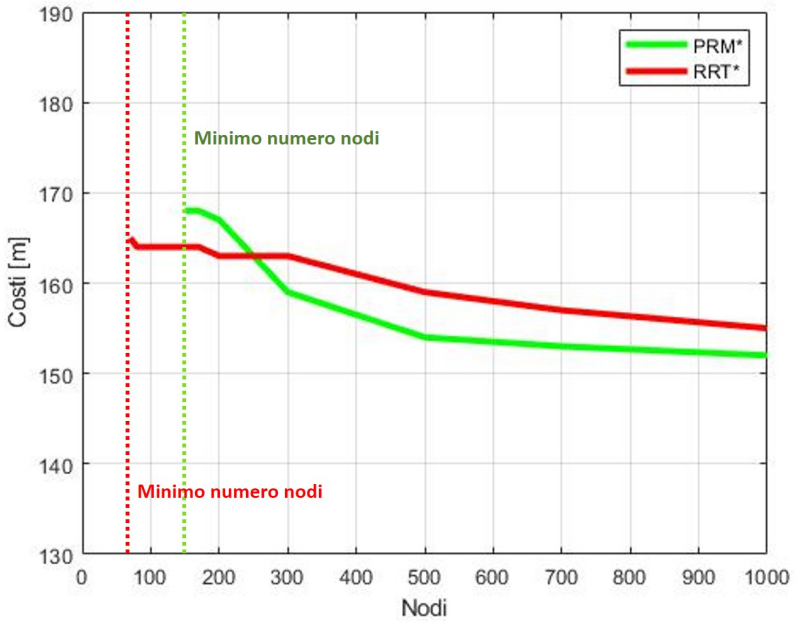


Figura 6.17: Evoluzione dei costi delle traiettorie trovate da **PRM*** e **RRT***

6.1.4 Problema delle collisioni non previste

In alcuni casi particolari può capitare che gli algoritmi PRM* e RRT* pianifichino traiettorie irrealizzabili nella pratica.

Si assuma per esempio di avere un robot mobile limitato da un massimo angolo di sterzatura θ_{max}. In questo caso, PRM* e RRT* non terrebbero conto dei limiti di movimento imposti da tale vincolo. Questi algoritmi infatti si fondano sull'assunzione di pianificare traiettorie per robot ideali, modellizzati come dei semplici punti omnidirezionali.

Questo problema viene superato dagli algoritmi Kinodynamic-PRM* e Kinodynamic-RRT* i quali calcolano traiettorie unicamente realizzabili, tenendo conto quindi dei vincoli cinematici del robot.

In figura 6.18 viene riportato un modello esemplificativo che mostra in rosso una possibile traiettora pianificata da PRM* o RRT* e irrealizzabile nella pratica.

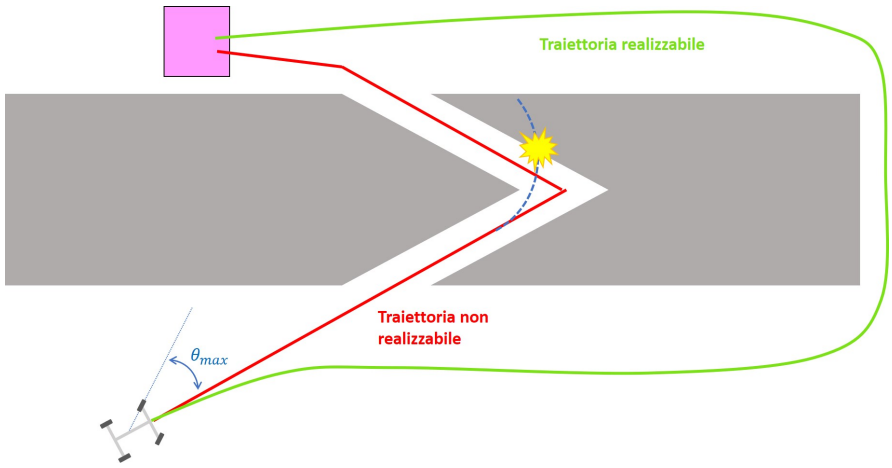


Figura 6.18

6.2 Algoritmi basati su una funzione di costo

Gli algoritmi analizzati in questa sezione sono **Kinodynamic-PRM*** e **Kinodynamic-RRT***.

In questi algoritmi la *steering function*, ovvero la funzione che crea le connessioni locali, ricava le traiettorie affidandosi alla minimizzazione di un'opportuna funzione di costo.

Per le prossime simulazioni è stata scelta la seguente funzione di costo:

$$c[\pi] = \int_0^{\tau} (1 + u(t)^T R u(t)) dt.$$

Essa assegna ad ogni traiettoria π un relativo costo $c[\pi]$ penalizzando le sollecitazioni sulla variabile di controllo $u(t)$ e il tempo di percorrenza τ delle traiettorie.

La matrice R è una matrice quadrata semi-definita positiva che pesa i contributi delle variabili di controllo del vettore $U(t)$. In questo caso si è deciso di utilizzare una matrice identità non essendoci particolari necessità di privilegiare l'azione di una piuttosto che l'altra variabile.

Per questa ragione, all'aumentare del numero di iterazioni, le traiettorie pianificate presenteranno cambi direzionali sempre meno drastici in favore di soluzioni più lineari.

Le traiettorie pianificate in questo modo dovranno inoltre sottostare alle leggi cinematiche dettate dal modello meccanico del robot.

Come viene descritto nel Capitolo 5, il modello considerato in questo caso è quello dell'uniciclo:

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = \omega \\ \dot{v} = a \end{cases}$$

il quale viene linearizzato attraverso un'opportuna trasformazione delle variabili ottenendo così il sistema lineare in forma di stato:

$$\begin{cases} \dot{x} = V_x \\ \dot{y} = V_y \\ \dot{v_x} = u_1 \\ \dot{v_y} = u_2 \end{cases}$$

Si noti come l'inclusione nello stato delle due componenti Cartesiane della velocità implica una continuità nell'orientamento del robot.

6.2.1 Confronto tra Kinodynamic-PRM* e Kinodynamic-RRT*

Sono state eseguite diverse simulazioni in un ambiente non eccessivamente limitante in modo da poter meglio osservare l'evoluzione delle traiettorie pianificate all'aumentare del numero di nodi.

Un esempio pratico viene riportato nelle figure 6.19 e 6.22 in cui le soluzioni sono state trovate aggiungendo un numero di nodi crescente alla stessa struttura di base.

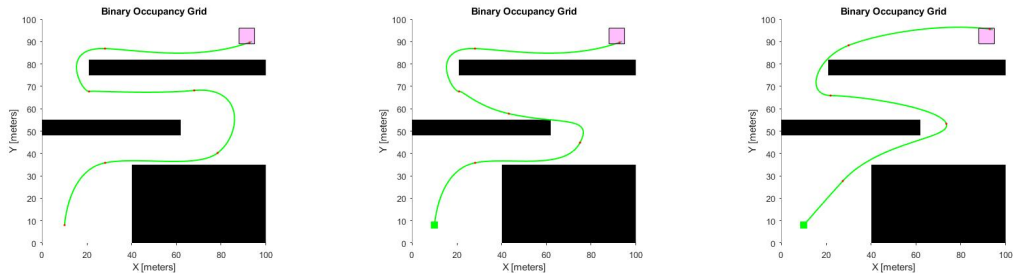


Figura 6.19: Kinodynamic-PRM* per 150, 200 e 500 nodi

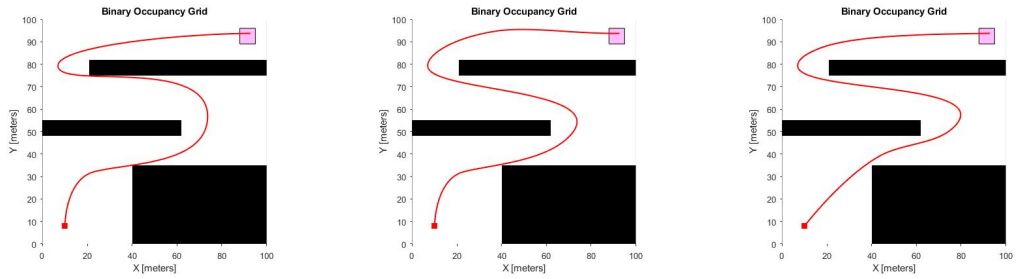


Figura 6.20: Kinodynamic-RRT* per 150, 200 e 500 nodi

A titolo dimostrativo, nelle figure , riportiamo l'evoluzione delle intere mappe di connessioni dei due algoritmi.

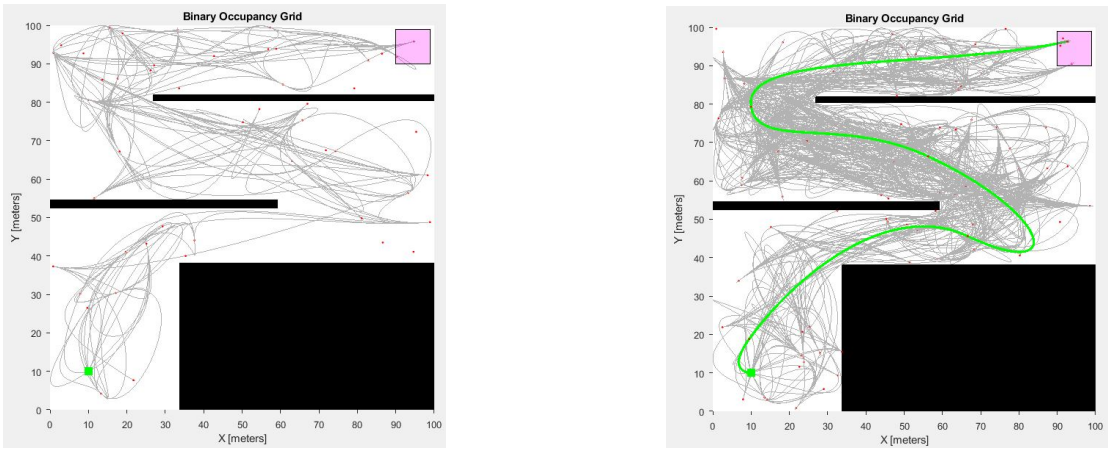


Figura 6.21: Kinodynamic-PRM* per 50 e 100 nodi

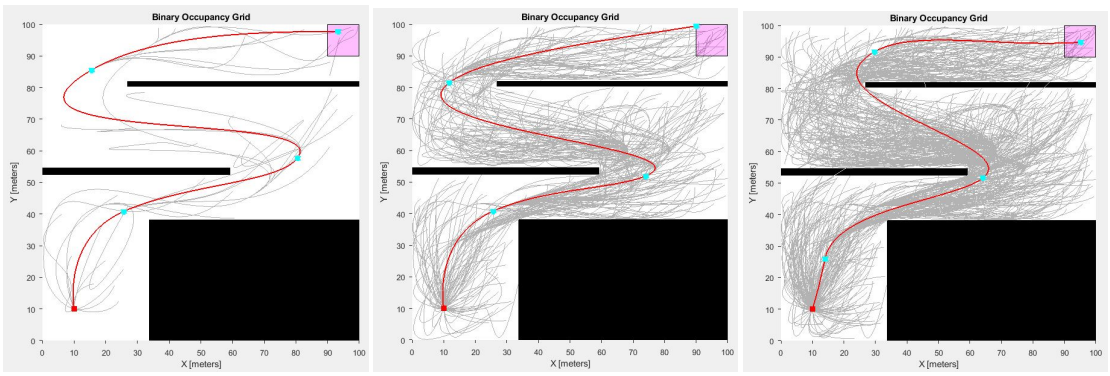
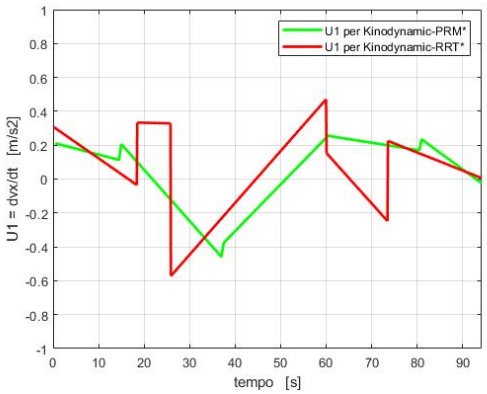
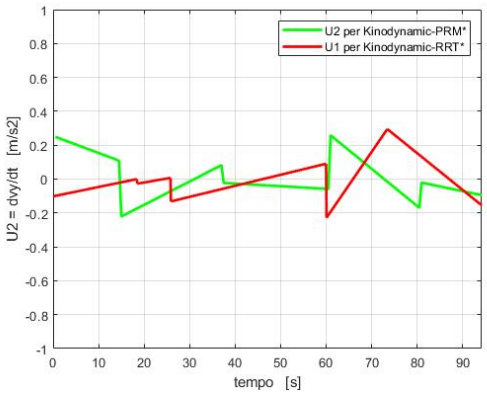


Figura 6.22: Kinodynamic-RRT* per 50, 500 e 1000 nodi

E' possibile notare come l'algoritmo Kinodynamic-PRM* pianifichi traiettorie connettendo gli archi generati singolarmente nella fase di *Learning*. Infatti,



(a) $U_1 = \dot{V}_x$



(b) $U_2 = \dot{V}_y$

Figura 6.23: Variabili di controllo

specialmente nelle prime iterazioni, è evidente che le soluzioni vengono ricavate unendo diverse traiettorie locali dando l'impressione di una linea spezzata. Questo fatto non si presenta nel caso di Knodynamic-RRT* data la diversa logica con cui quest'ultimo pianifica la traiettoria.

Per queste simulazioni sono state osservate anche le variabili di controllo u_1 e u_2 che ricordiamo essere le derivate temporali delle due componenti Cartesiane del vettore velocità:

$$\begin{cases} u_1 = \dot{V}_x \\ u_2 = \dot{V}_y \end{cases} \tag{6.1}$$

In figura 6.23 vengono riportati i grafici delle due variabili per una simulazione ottenuta fissando il numero di nodi a 500.

In questi casi non è stata applicata nessuna saturazione alle variabili di controllo. D'altro canto, per come viene definita la funzione di costo, al progredire delle iterazioni, verranno scartate le soluzioni che sollecitano maggiormente le variabili di controllo.

Dal grafico dei costi riportato in figura 6.24 si è potuto concludere che l'algoritmo Kinodynamic-PRM*, apparentemente meno adatto a sistemi di questo tipo, calcola soluzioni di ottime qualità.

Quello che si è constatato infatti, è che per un numero di iterazioni sufficientemente elevato, esso pianifica traiettorie di costi simili a quelle trovate da Kinodynamic-RRT*.

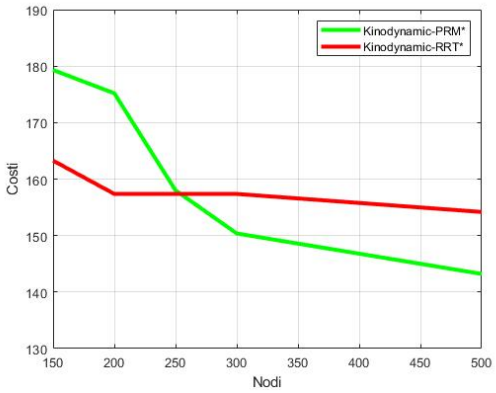


Figura 6.24: Costi traiettorie

Conclusioni

In questo lavoro di tesi si è voluto dare una panoramica generale sui principali algoritmi della classe *Sampling-based*.

In primo luogo sono stati presentati nel loro contesto globale, ovvero all'interno del macro problema relativo alla Pianificazione di Traiettoria.

In secondo luogo, suddividendoli nelle due categorie *Probabilistic RoadMap* e *Rapidly-exploring Random Tree*, ci si è focalizzati sull'analisi specifica delle sequenze logiche che li caratterizzano. Per una maggiore chiarezza, è stato deciso di riportare le tecniche adottate per l'implementazione pratica nell'ambiente *Matlab*.

Nei primi capitoli gli algoritmi sono stati applicati ad un modello di robot astratto rappresentato dalle sole variabili di posizione nel piano Cartesiano. In questo modo è stato possibile comprendere in modo generale e chiaro le sequenze logiche che li caratterizzano.

I risultati ottenuti dalle analisi teoriche e pratiche mostrano alcuni vantaggi e svantaggi relativi ai diversi algoritmi: si è potuto constatare per esempio che l'algoritmo RRT*, apparentemente migliore in tutti gli aspetti rispetto a PRM*, a parità del numero di nodi, è meno efficiente in termini di copertura dello spazio libero. Un'altra osservazione degna di nota è la differenza dal punto di vista del carico computazionale tra RRT e RRT*. Si è notato infatti come l'aggiunta della fase di *Rewiring* abbia notevolmente incrementato i tempi necessari alla computazione dei codici pur migliorando in modo consistente la qualità delle traiettorie calcolate.

In seguito il modello del robot è stato esteso ad un caso kinodinamico aggiungendo allo stato anche le componenti Cartesiane della velocità. E' stato quindi implementato l'algoritmo Kinodynamic-RRT* utilizzando una tecnica particolare di linearizzazione che non prevede alcun genere di approssimazione.

Utilizzando lo stesso Pianificatore Locale adottato per Kinodynamic-RRT*, è stato implementato il codice relativo all'algoritmo Kinodynamic-PRM*. Quest'ultimo si fonda sulle sequenza logiche dettate da PRM*, ma connette gli archi minimizzando un'apposita funzione di costo.

Sebbene l'algoritmo PRM* non sia nato con lo scopo di essere applicato a modelli di questo genere, l'algoritmo ottenuto risulta essere piuttosto soddisfacente rispetto alle previsioni. Infatti oltre al fatto che i tempi di computazione non risultano proibitivi l'algoritmo pianifica soluzioni di ottime qualità.

In questa tesi non sono state applicate limitazioni alle variabili di stato e di controllo del robot. Com'è possibile intuire però, all'aumentare del numero di iterazioni degli algoritmi kinodinamici, le soluzioni trovate presentano variabili sempre meno sollecitate grazie all'inclusione nella cifra di merito della variabile di controllo.

Nei possibili sviluppi futuri però sarebbe opportuno includere nelle implementazioni pratiche anche le saturazioni dovute a tali limitazioni.

Inoltre vi è da aggiungere che questa tesi si è limitata ad un'analisi teorica degli algoritmi, per questo non sono stati inclusi i modelli dei sensori di posizione e di rilevazione degli ostacoli. In questo senso si è assunto di conoscere l'esatta geometria e posizione degli ostacoli all'interno della mappa. Un futuro aggiustamento potrebbe quindi essere quello di includere negli algoritmi delle logiche di mappatura dell'ambiente sulla base di esperimenti preliminari.

Sebbene in questa tesi vengano proposti modelli di robot terrestri capaci di muoversi unicamente nelle due dimensioni, tali algoritmi risultano estremamente vantaggiosi anche per robot manipolatori di svariati gradi di libertà. In questi casi lo stato verrebbe definito come l'insieme delle variabili dei giunti e di conseguenza i nodi sarebbero vettori n-dimensionali. Per questa ragione, un terzo sviluppo futuro potrebbe riguardare l'applicazione di questi algoritmi alla classe dei robot antropomorfi.

Appendice A

Manuale utente per l'utilizzo dei codici Matlab

In questa Appendice si vuole guidare l'utente all'utilizzo dell'ambiente per le simulazioni degli algoritmi trattati.

Gli algoritmi sono stati implementati in linguaggio *Matlab* utilizzando la versione *2018a*.

La stesura dei codici è stata eseguita attraverso l'utilizzo di diverse *Matlab function* già presenti nei toolbox della *Mathworks* e da diverse altre create appositamente per questo scopo.

All'interno dei codici si è cercato di riportare in modo chiaro ma allo stesso tempo sintetico, i commenti relativi alle varie sezioni. Si è ritenuto però di dover dare delle indicazioni generali sull'utilizzo dei codici in modo da aiutare l'utente a condurre simulazioni per ambiti diversi da quelli di questa tesi.

Per ogni algoritmo è stata creata un'apposita directory contenente il codice principale e le diverse *Matlab function*. in particolare però, essendo l'algoritmo Kinodynamic-RRT* strutturato su una logica del tutto analoga a quella di RRT*, si è deciso di utilizzare uno stesso codice principale in cui è possibile selezionare inizialmente quale dei due algoritmi si vuole utilizzare (figura A.1). Alla voce `algoritmo`, all'inizio dell'inizializzazione, si può assegnare il valore **1** qualora si voglia eseguire RRT* oppure **2** se si voglia usare Kinodynamic-RRT*. Grazie all'utilizzo di puntatori a funzioni, si andranno dunque a definire le funzioni relative al primo o al secondo algoritmo.

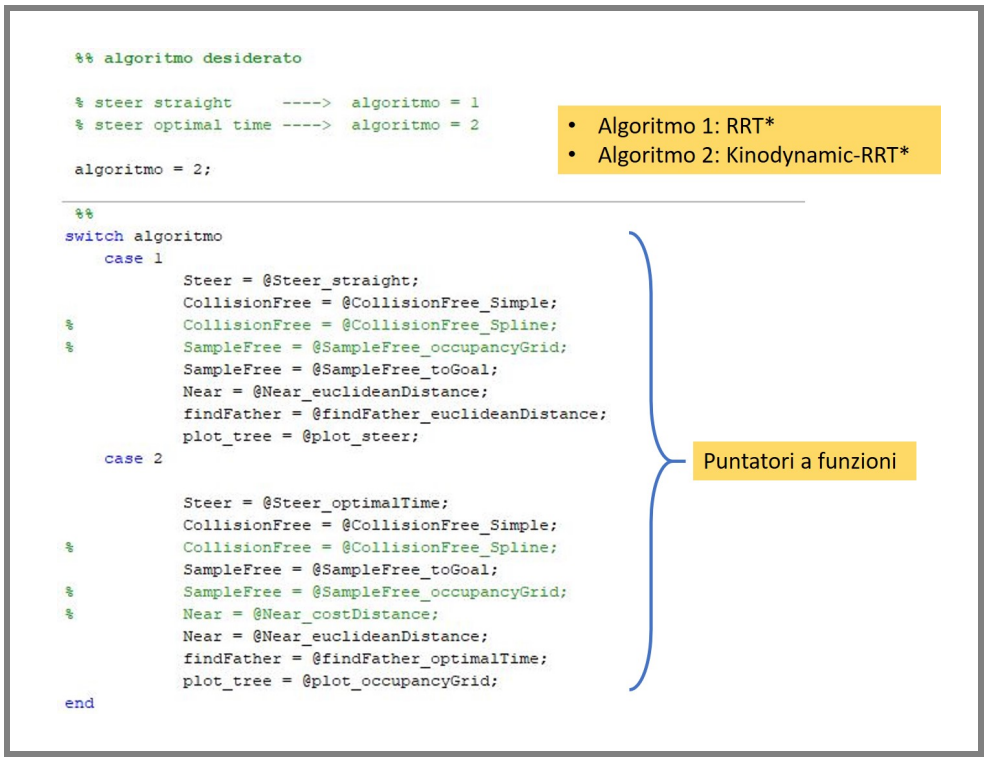


Figura A.1: Scelta dell'algoritmo

Le funzioni implementate vengono spesso utilizzate per diverse tipologie di algoritmi, per questa ragione in alcune di esse si dovrà passare come argomenti diversi parametri non utilizzati per quel caso specifico.

Nel codice principale si possono osservare quattro sezioni fondamentali: inizializzazione, fase *Learning*, fase *Quering* e visualizzazione grafica.

Inizializzazione

In questa fase l'utente può definire le più importanti specifiche del problema come: il modello del robot con le relative matrici della dinamica, il numero di nodi (e quindi delle iterazioni) che si vogliono utilizzare per campionare l'ambiente, il passo temporale che intercorre tra i diversi nodi della traiettoria, lo stato iniziale del robot, la regione di Goal, la mappa (definita come griglia di valori binari) e i parametri per il calcolo del raggio d'azione (figure A.2 e A.3).

In ogni directory si possono trovare delle possibili configurazioni degli ostacoli, tra le quali vi sono quelle utilizzate per le simulazioni di questa tesi. Degno di nota è il parametro **cellDim**, il quale esprime in metri la lunghezza del lato di una singola cella della griglia (figura A.4).

Diverse configurazioni si possono ottenere tramite la generazione di un'immagine in formato jpg. E' importante però sottolineare che qualora si voglia ottenere un ambiente con delle specifiche dimensioni, è necessario caricare un'immagine che abbia il giusto numero di pixels.

A titolo d'esempio: se si volesse ottenere un ambiente di dimensioni 100x100 metri con una risoluzione di 0.01 cm a cella, sarebbe necessario caricare un'immagine di 10000x10000 pixels.

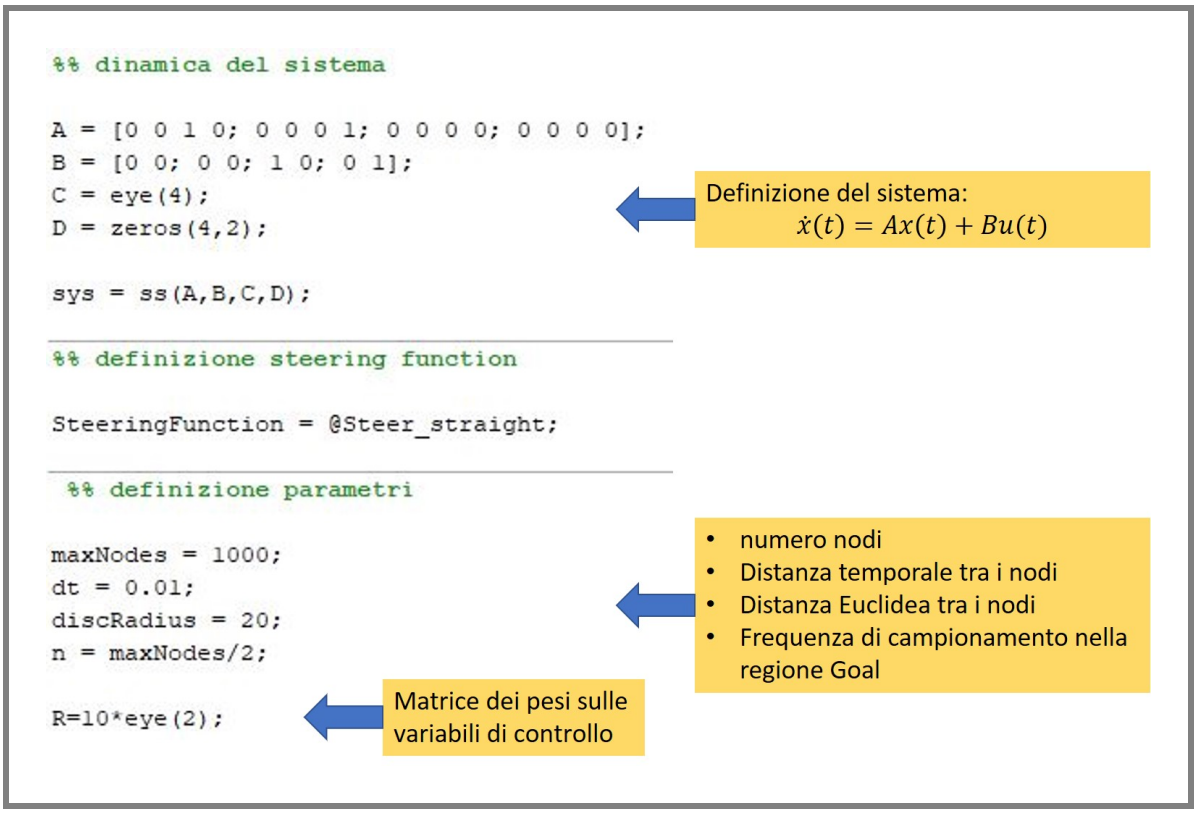


Figura A.2: Inizializzazione sistema e definizione variabili generali

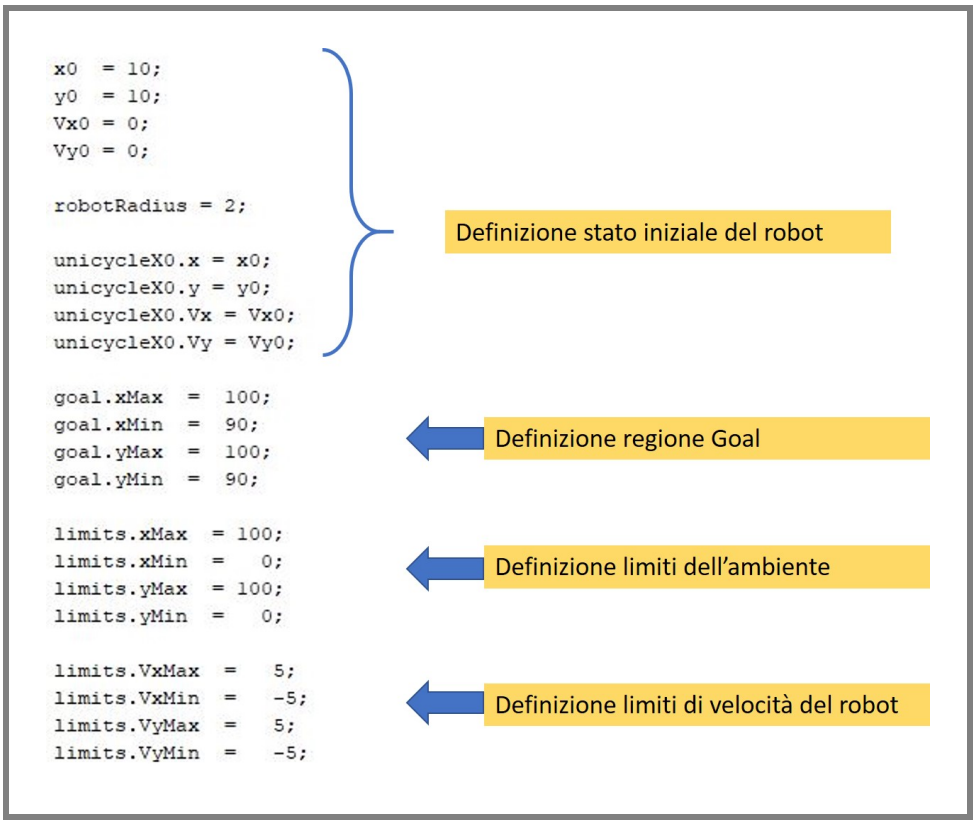


Figura A.3: Definizione stato iniziale, limiti ambiente e regione Goal

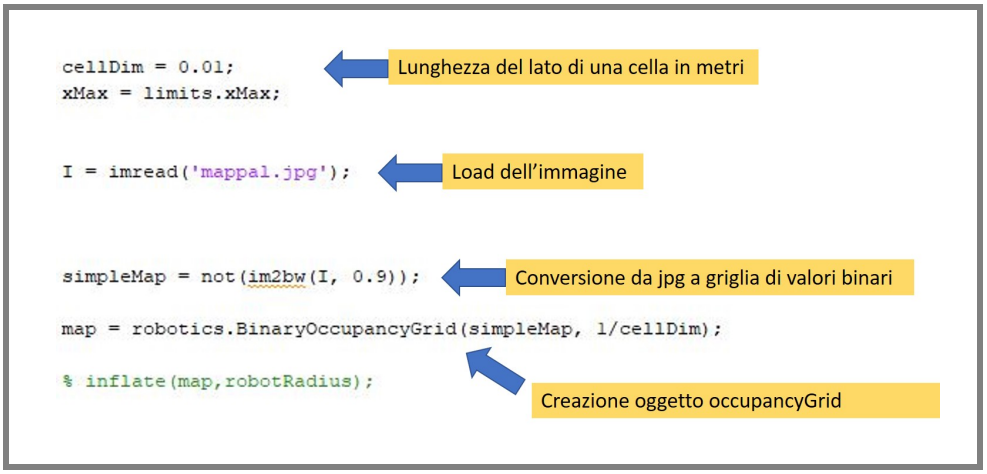


Figura A.4: Inizializzazione mappa

Learning

Questa sezione è quella più versatile in quanto varia in base all'algoritmo. In generale però si può affermare che si tratta di un unico ciclo in cui viene iterato lo stesso procedimento per n volte, dove n è il numero di nodi inserito.

La RoadMap per gli algoritmi PRM e l'Albero negli RRT vengono definiti tramite un oggetto di tipo `struct`(figure A.5, A.6 e A.7). Questo tipo di oggetto permette una definizione gerarchica. In esso infatti vengono aggiunti progressivamente tutti i nodi con i relativi attributi che ne descrivono le connessioni e i costi.

Nodes(i)	.State	.x
		.y
	.goalReach	0/1
	.componentNumber	Scalare
	.idPrec	[id1, id2, id3, ...]
	.idSucc	[id1, id2, id3, ...]

Figura A.5: `struct` PRM, sPRM e PRM*

Nodes(i)	.State	.x					
		.y					
	.goalReach	0/1					
	.componentNumber	Scalare					
	.idPrec	Array					
	.idSucc	.Id	Scalare				
		.Trajectory	.X	x(1)	x(2)	x(3)	
				y(1)	y(2)	y(3)	...
			.t	[t1, t2, t3, t4, ...]			
			.U	u1(1)	u1(2)	u1(3)	...
				u2(1)	u2(2)	u2(3)	
			.C	Scalare			

Figura A.6: `struct` Kynodynamic-PRM*

Nodes(i)	.State	.x					
		.y					
	.goalReach	0/1					
	.componentNumber	Scalare					
	.idPrec	Array					
	.idSucc	.Id	Scalare				
		.Trajectory	.X	x(1)	x(2)	x(3)	
				y(1)	y(2)	y(3)	...
			.t	[t1, t2, t3, t4, ...]			
			.U	u1(1)	u1(2)	u1(3)	...
				u2(1)	u2(2)	u2(3)	
			.C	Scalare			

Figura A.7: `struct` RRT, RRT* e Kynodynamic-RRT*

Quering

La fase di *Quering* consiste essenzialmente nella ricerca, all'interno della struttura dei nodi, della traiettoria ottima.

Negli algoritmi **PRM** si utilizzano le funzioni relative all'oggetto `digraph`, un grafo orientato definito da nodi e archi pesati. Per poter creare un digrafo è necessario prima definire la matrice della Adiacenze (Adj in figura A.8). Con il comando `plot(digraph)` sarà possibile visualizzare il modello semplificato del grafo come mostrato in figura A.9 (per soli 30 nodi).

```
Adj = zeros(length(Nodes),length(Nodes));

for i=1:length(Nodes)
    for j=i:length(Nodes)
        ind = find([Nodes(i).idSucc(1:end)] == j);
        if (length(ind) ~= 0) && (i ~= j)
            distance=sqrt((Nodes(i).state.x - Nodes(j).state.x)^2 + ...
                (Nodes(i).state.y - Nodes(j).state.y)^2);
            Adj(i,j) = distance;
            Adj(j,i) = distance;
        end
    end
end

G = digraph(Adj);
```

Ad ogni arco viene assegnato un peso pari alla distanza Euclidea percorsa

Figura A.8: Matrice adiacenze

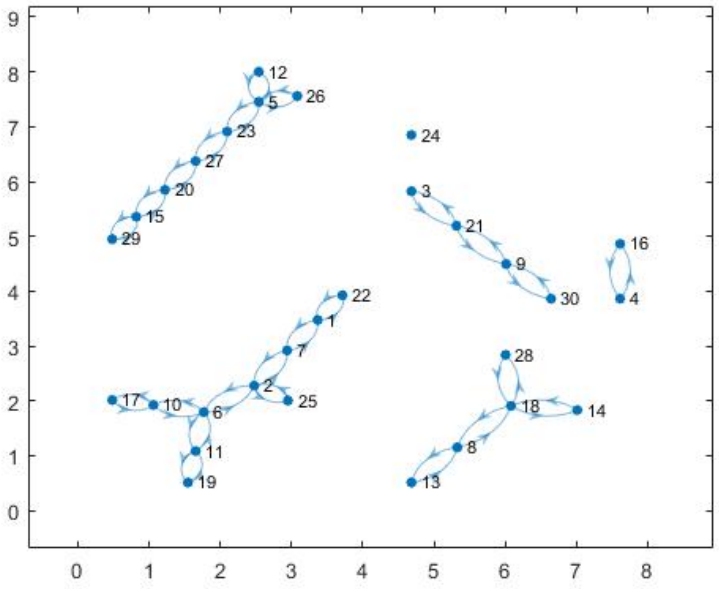


Figura A.9: Digrafo per 30 nodi

In un secondo tempo, grazie al comando `shortestpathtree`(figura A.10) si ottiene la traiettoria desiderata (qualora essa esista).

```
% trovo best nodo

ind = find([Nodes(1:end).goalreach]==1);
indBest = [];
for i=1:length(ind)
    if Nodes(ind(i)).componentNumber == 1
            indBest(end+1) = ind(i);
    end
end

% trovo tutti i cammini che partono da start
% e arrivano in uno dei bestNodes

if length(indBest) ~= 0
    s = 1;
    t = indBest;
    [TR, D] = shortestpathtree(G,s,t);

    [minCost indNewT] = min (D);
    newT = indBest(indNewT);
    [TR, D] = shortestpathtree(G,s,newT);
    K = table2array(TR.Edges);
end
```

All'interno della RoadMap vengono identificati i nodi connessi al nodo **start** e appartenenti alla regione goal

Questo comando permette di trovare il cammino ottimo tra **start** e **goal**

Figura A.10: Ricerca cammino ottimo

Per quanto riguarda l'algoritmo **Kinodynamic-PRM***, la fase di *Quering* rimane la stessa. L'unica differenza sta nell'assegnare agli archi della matrice delle

adiacenze i costi relativi alla cifra di merito.

Negli algoritmi **RRT**, la ricerca del cammino ottimo viene affidata alla funzione `Sbest_ path`. Tale funzione, dopo aver identificato il nodo finale all'interno della regione di goal, ripercorre al contrario la traiettoria fino al nodo radice (l'unico senza un nodo padre).

Visualizzazione Grafica

In questa fase vengono richiamate le funzioni dedicate alla visualizzazione grafica della traiettoria e dell'intera rete delle connessioni. Inoltre, per quanto riguarda il plottaggio del cammino ottimo vengono salvati e sommati i pesi dei singoli archi in modo da restituire il costo relativo all'intero percorso finale.

Per quanto riguarda gli algoritmi più semplici, ovvero quelli che utilizzano la funzione `steerStraight`, le traiettorie vengono tracciate unendo i singoli nodi del percorso. Nei casi più complessi che utilizzano la cifra di merito e che quindi presentano traiettorie curve, le funzioni di plottaggio andranno a unire tutti i nodi campione di ogni singola traiettoria locale e per questo richiedono tempi computazionali maggiori.

Bibliografia

[1] *Sampling-based algorithms for optimal motion planning*, Sertac Karaman and Emilio Frazzoli, Jun 22, 2011.

[2] *Kinodynamic RRT*: Optimal Motion Planning for Systems with Linear Differential Constraints*, Dustin J. Webb, Jur van den Berg, 23 May 2012.

[3] *Poli-RRT*: Optimal RRT-based Planning for constrained and Feedback Linearisable Vehicle Dynamics*, Matteo Ragaglia, Maria Prandini and Luca Bascetta July 2015.

[4] *Planning Algorithms*, Steven M. LaValle, 2006.

[5] *Control of Industrial Robots: Motion planning*, Dispense del professore Paolo Rocco, Politecnico di Milano.

[6] *Robotics Toolbox for Matlab, Release 10*, Peter Corke, Novembre 2017.

[7] *Sito ufficiale della Mathworks*, `https://www.mathworks.com`.

[8] *Humanoid robot path planning with fuzzy Markov decision processes*, Fakoor, Mahdi; Kosari, Amirreza; Jafarzadeh, Mohsen, 2016.

[9] *Revision on fuzzy artificial potential field for humanoid robot path planning in unknown environment* , Mahdi Fakoor, Amirreza Kosari and Mohsen Jafarzadeh, 2015.

[10] *A new algorithm for computing Visibility Graphs of polygonal obstacles in the plane*, Danny Z. Chen, Haitao Wang, 2015.

[11] *Advanced and Multivariable Control*, Lalo Magni e Riccardo Scattolini, 2014.

[12] *Optimal Control*, Frank L. Lewis, Draguna Vrabie, Vassilis L. Syrmos, 2011.

[13] *Teoria dei Sistemi e del Controllo*, L. Biagiotti, R. Zanasi, 2010/2011.

[14] *Rapidly-Exploring Random Trees: A New Tool for Path Planning*, Steven M. LaValle.

[15] *A Comparison of RRT, RRT* and RRT*-Smart Path Planning Algorithms*, Iram Noreen, Amna Khan, Zulfiqar Habib, October 2016.

[16] *Principles of Robot Motion*, H. Choset et. al. MIT Press.

[17] *Motion Planning and Feedback Control for a Unicycle in a Way Point Following Task: the VFO Approach*, Maciej Michaelek, Krzysztof Kozlowski, 2009.