POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Meccanica

Tesi di Laurea Magistrale

Analisi di algoritmi di path-planning e obstacle avoidance in uso nei sistemi AGV

Politecnico
di Torino

Relatori
Prof. Aurelio Somà
Dott. Francesco Mocera

Candidato
Daniele Biolè

Anno Accademico 2020/2021

Indice generale

ABSTRACT

Il seguente lavoro ha lo scopo di mostrare quelli che sono ad oggi alcuni dei metodi più utilizzati ed efficienti per quella che è la pianificazione e l'inseguimento della traiettoria di robot a guida autonoma. Verranno illustrate quelle che sono le macrocategorie di tali algoritmi scendendo più nello specifico nel caso dei principali esponenti di queste ultime. Sarà brevemente illustrato quali sono i vantaggi e svantaggi delle varie strategie e qual è il loro funzionamento.

Si è poi provato ad estendere il discorso toccando anche quelli che possono essere gli eventuali sviluppi del Machine Learning nel campo con particolare enfasi sul Reinforcement Learning cercando di valutarne le prestazioni in confronto alle tecniche più "classiche" sopra citate.

ABSTRACT

The purpose of this thesis is to show some of the most used and most efficient methods for the path-planning and path following of the trajectory of an automatically guided robot. The macro categories of algorithms will be shown and more specifically will be discussed some of the main exponents of each category. It will be briefly discussed which are the various advantages and disadvantages of each strategy and how they work.
Later has also been discussed the possible involvement of Machine Learning in such topic, particularly for the sub-branch called Reinforcement Learning trying to determine how it performs and compares near the more “classic” technique shown above.

Capitolo 1

1.1 Introduzione

L'avvento della quarta evoluzione industriale anche detta Industria 4.0, basata sull' integrazione di nuove tecnologie concernenti l'informazione e la connessione al fine di rendere sempre più efficienti ed autonomi i processi di produzione ha permesso negli ultimi decenni di assistere ad un'evoluzione tecnologica senza pari. In questo contesto sempre più votato all'efficienza ed alle prestazioni trovano spazio una miriade di soluzioni fino a pochi anni fa impensabili.
Una branca in particolare che verrà esplorata nel corso di questa tesi è quella concernente i veicoli a guida autonoma anche detti AGV, ovvero veicoli in grado di essere guidati senza operatore o comunque di lavorare in maniera attiva e sinergica con quest'ultimo. Caratterizzati da innegabili vantaggi in termini di precisione ed efficienza questi sistemi sono già largamente in uso in quelle che possiamo definire come applicazioni al chiuso, ovvero magazzini e fabbriche e più recentemente anche in applicazioni all'aperto come cantieri o campi agricoli stanno prendendo piede.

Figura 1: Veicoli AGV industriali

Particolare attenzione nel seguente lavoro verrà posta sull'ultima casistica presentata, ovvero mezzi in grado di muoversi in ambienti esterni. Vi sono infatti sostanziali differenze di ambiente e condizioni di lavoro. Se la routine di un robot industriale infatti è particolarmente rigida e sicura dal momento che i compiti che deve svolgere sono sempre identici ed in ambienti ad hoc, così non è per una macchina ad uso esterno. Una macchina che lavora in un campo agricolo o in un cantiere deve potersi muoversi in un ambiente ricco di ostacoli, oltre in generale a svolgere routine di movimento non altrettanto regolari come la controparte industriale.

Figura 2: Mezzo da lavoro in cantiere

Per la soluzione del problema legato al movimento vi è un generale consenso che gli step da affrontare per la risoluzione sono tre:

- **Path-planning:** Ovvero la ricerca di quello che è il percorso migliore dal punto iniziale al punto finale. Importante notare che nella ricerca del percorso da seguire oltre alla geometria sono incluse le velocità fisicamente possibili del robot.
- **Path-following:** Una volta trovato un percorso da seguire è necessario renderlo "seguibile" in accordo con quelli che sono i limiti dinamici della macchina reale.
- **Obstacle avoidance:** Capita spesso in ambienti come un campo o un cantiere di trovare dossi buche o altri ostacoli, è necessario quindi che possano essere evitati senza fermare la produzione e senza che ci si allontani eccessivamente dall'obbiettivo da raggiungere. Anche se di fatto può essere visto come una problematica a sé stante solitamente questo punto viene risolto e sviluppato in modo parallelo al path-following ed al path-planning.

1.2 Path-planning

Primo passo nella realizzazione di un sistema AGV funzionante è sicuramente il path planning o più semplicemente "pianificazione della traiettoria" in italiano. È infatti fondamentale la scelta della traiettoria da seguire per andare dal punto di partenza al punto di arrivo, non solo per un mero discorso di efficienza, ma anche e soprattutto per aggirare a monte quelli che sono gli ostacoli fissi dell'ambiente di lavoro quali muri, buche, dossi e simili. Se dalla prospettiva di un essere umano si tratta di un compito molto semplice se non addirittura banale per una macchina invece è un problema estremamente complesso. Non è per nulla immediato trovare quella che è una traduzione del problema nel linguaggio di basso livello usato per programmare. È a questo punto che entrano in gioco quelli che sono gli algoritmi di path-planning per dare alla macchina la capacità di risolvere in maniera autonoma tali problemi.

Condizione fondamentale di cui quasi tutti gli algoritmi odierni è la conoscenza a monte di quello che è l'ambiente geometrico in cui è inserito il robot, il punto di partenza, il punto di arrivo e quelli che sono i limiti cinematici della macchina (gradi di libertà). Lo spazio di tutte le configurazioni fisicamente realizzabili dal robot e che non dà come risultato una collisione con un ostacolo è detto *free configuration space* e scopo dell'algoritmo è trovare un percorso che unisca la configurazione iniziale alla configurazione finale rimanendo sempre all'interno del suddetto spazio.

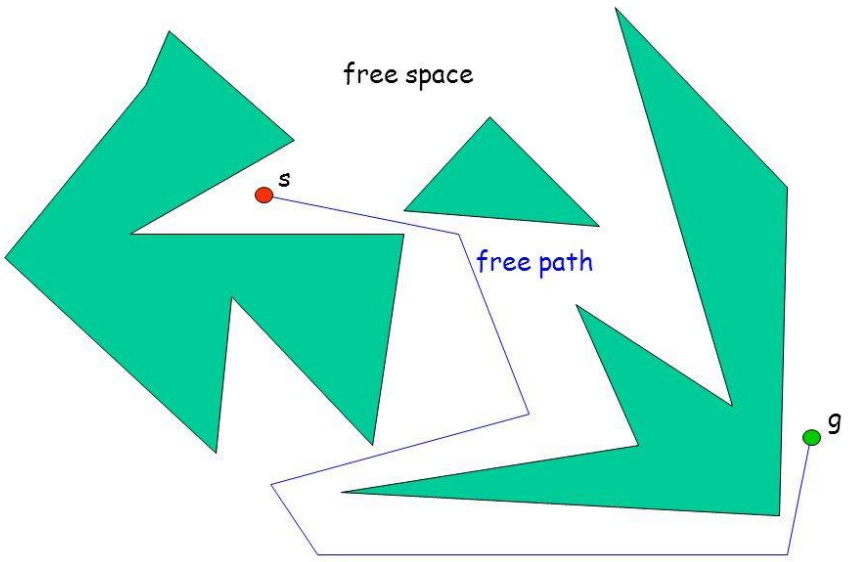


Figura 3: Esempio di problema di path-planning

Le strategie con cui sviluppare l'algoritmo sono molteplici e differenti, in particolare possono essere di natura locale o globale. I planner locali non cercano una soluzione di tipo generale al problema che si pone, ma si limitano a dare indicazioni su quelli che devono essere i movimenti nelle immediate vicinanze del robot, mentre quelli globali risolvono il problema nella sua interezza, similmente a come se dovessero uscire da un labirinto. In generale al giorno d'oggi si cerca di adottare una strategia ibrida, ovvero globale per la ricerca del percorso ideale da seguire e locale per la gestione di eventuali ostacoli imprevisti.

Di seguito sono elencate alcune delle principali classi di algoritmi[2] utilizzati e che verranno trattate in maniera più approfondita più avanti nella tesi:

- **Grid-based search algorithms:** È una delle categorie di algoritmi più semplici possibili. Si basa sulla suddivisione dello spazio libero in celle raggiungibili solo da un'altra cella adiacente. Sono molto semplici, ma è immediato intuire che il costo computazionale cresca esponenzialmente con la risoluzione della griglia e la complessità del robot (maggior numero di gradi di libertà) perciò sono generalmente limitati ad applicazioni ridotte per dimensioni e complessità.
- **Artificial Potential Fields:** Sono più che altro utilizzati a livello locale per la rilevazione e l'aggiramento di ostacoli. Si basano sull'assegnazione di un campo potenziale di forze fittizie che tendono ad attrarre il robot nella regione di arrivo e ad allontanarlo dagli ostacoli. In applicazioni globali possono essere pericolosi in quanto c'è il rischio che intrappolino il robot in dei minimi locali.
- **Visibility Graph algorithms:** Gli algoritmi di questa categoria si basano sul concetto di *Visibility graph.* I nodi del grafico rappresentano le posizioni raggiungibili nel piano Euclideo e possono essere collegati da un segmento solo se non attraversa un ostacolo. I principali esponenti di tale categoria sono sicuramente l'algoritmo di Dijkstra e l'algoritmo A*.
- **Sample-based algorithms:** Sono sicuramente la classe di algoritmi più popolare al momento prevalentemente per il loro ridotto costo computazionale rispetto alle precedenti famiglie. Infatti, anzi che discretizzare e connettere l'intero spazio delle configurazioni libere gli algoritmi di questa categoria si basano sul campionamento stocastico dello spazio delle configurazioni libere detto *Roadmap.* Si può dire che in generale questi algoritmi funzionano in due fasi. Una fase di *Learning* in cui i nodi vengono generati e connessi tra di loro ed una fase

di *Quering* in cui il percorso ottimale viene ricavato.

Quelle illustrate sono solo alcune delle innumerevoli strategie possibili per un algoritmo di path-planning, ma è importante notare che al giorno d'oggi l'argomento è in continua evoluzione oggetto di una costante ricerca visti gli evidenti vantaggi che presenta una fruttuosa applicazione. In particolare, negli ultimi anni argomenti sono stati sollevati a favore di quello che potrebbe essere un futuro sviluppo grazie alle tecnologie di Machine Learning con particolare attenzione su quello che è definito Reinforcement Learning. Più avanti nella tesi è illustrato un tentativo che è stato fatto in tal senso, ma per il momento sembra evidente che quest'ultimo tipo di AGV è ancora decisamente indietro rispetto alle categorie di algoritmi fin ora elencati.

1.3 Path-following e obstacle avoidance

Una volta stabilita la traiettoria da seguire è fondamentale riuscire ad implementarla e realizzarla in quella che è la realtà pratica tramite opportuni comandi. Questa fase è detta fase di path-following. Non vi è un algoritmo o una classe di algoritmi che possano chiaramente definire come effettuare questa fase anche e soprattutto perché è intrinsecamente legata a quella che è la realizzazione fisica del robot stesso. In base a quelle che sono le proprietà inerziali ed ai sistemi di controllo a bordo ci possono essere una miriade di approcci differenti. In generale, comunque, almeno per quel che riguarda una tipologia di macchine come possono essere delle navette o dei trattori i comandi sono sostanzialmente due; la coppia motrice alle ruote ed un comando di sterzata che può avvenire tramite un vero è proprio sistema di sterzata come può essere un Ackermann oppure differenziando quelle che sono le velocità delle singole ruote. Anche in questa particolare casistica non è comunque possibile dire molto altro che abbia una valenza generale dal momento che anche tra veicoli tutto sommato simili è possibile avere innumerevoli possibilità su come impartire i comandi di accelerazione e sterzata.
Per quanto riguarda quelle che invece sono le strategie di obstacle avoidance è invece possibile espandere il discorso in maniera più generale dal momento che, come per il path-planning, la problematica è molto ben risolvibile tramite algoritmi. Come già accennato in precedenza per la risoluzione dei problemi di obstacle avoidance si predilige un approccio di carattere locale da inserire in parallelo all'approccio globale del path-planning. Più precisamente le strategie che verranno illustrate si prefiggono principalmente come obbiettivo la risoluzione di ostacoli imprevisti, come pozzanghere, dossi o altre asperità del terreno che in fase di path-planning non sono e, nella quasi totalità dei casi, non possono essere previste. Alcuni degli algoritmi più comuni e più utilizzati e che verranno trattati più esaustivamente nella debita sezione sono:

- **Bug algorithms:** È concettualmente molto semplice. Una volta incrociato l'ostacolo sul proprio cammino il robot ne segue il contorno finché non trova il punto più vicino all'obbiettivo finale e a quel punto riprende la marcia nella direzione di arrivo. Seppur di una semplicità disarmante concettualmente sono lenti e molto onerosi dal punto di vista computazionale.
- **Potential fields algorithms:** Come già precedentemente detto nel riassunto delle tecniche di path-planning questo tipo di algoritmi rende di fatto l'obstacle

avoidance un sottoprocesso, ma presenta diversi inconvenienti nell'affrontare passaggi stretti o nel caso di ambienti molto complessi di eventuali minimi locali in cui il robot rischia di "impantanarsi".

- **Vector field histogram (VFH) algorithms:** Sicuramente quella che ad oggi è una delle tecniche più utilizzate ed efficienti. Tale algoritmo si basa sulla creazione di un diagramma sulle cui ascisse sono riportati gli angoli relativi alle letture del sonar e sulle ordinate la possibilità di imbattersi in degli ostacoli. Da tale diagramma è poi possibile identificare passaggi sufficientemente larghi da permettere il passaggio del robot.
- **Follow the gap method (FGM):** Altro algoritmo estremamente semplice a livello di concetto. Come suggerisce il nome si basa sulla ricerca di quello che è il passaggio più largo nelle immediate vicinanze del robot. Per applicazioni semplici è particolarmente veloce ed efficiente, ma è poco robusto in particolare se deve effettuare manovre più complesse come un'inversione di marcia a U.

Capitolo 2

In questo capitolo si andranno a sviscerare in maniera più completa ed esaustiva quelle che sono le categorie di algoritmi di path-planning presentate in precedenza. Prima di proseguire è però importante dare ancora alcune nozioni che saranno fondamentali per determinare la bontà dei vari algoritmi per quelle che sono applicazioni di tipo generale. In particolare, è fondamentale la nozione di "completo". Un algoritmo si definisce completo se trova una soluzione esistente oppure ritorna un messaggio di errore in caso contrario. Più in particolare il secondo caso è detto "a risoluzione completa". Se invece l'algoritmo pur non riuscendo a convergere ad una soluzione continua a lavorare all'infinito è detto "semi-completo". Ultima casistica e anche la più comune è quella definita come algoritmi a risoluzione "probabilisticamente completa", ovvero algoritmi che per un numero tendente infinito di iterazioni tende a trovare quella che è la soluzione completa con probabilità tendente a 1.

2.1 Grid-based search algorithms

Forse la categoria più semplice possibile di algoritmi. Si basa sulla discretizzazione del dominio libero F in celle elementari. Tale decomposizione può essere esatta se riesce a comprendere l'intero dominio o approssimata qualora sia semplicemente un sottoinsieme di esso. Ottenuta la decomposizione il movimento viene determinato con un grafo di connettività che unisce tutte le celle adiacenti e che aiuta a trovare quello che viene definito come canale attraverso le celle prive di collisioni. Nel seguente sotto capitolo saranno analizzati alcuni metodi con cui tale decomposizione può avvenire.

2.1.1 Cell decomposition methods

La decomposizione in celle di un ambiente di lavoro può avvenire con varie metodologie e criteri.
Una decomposizione in celle per poter essere utilizzabile efficacemente da un algoritmo deve possedere determinate caratteristiche che ne rendano agevole e veloce la

manipolazione matematica. Per esempio, le singole celle non devono essere troppo grandi o complicate perché se così fosse si potrebbe arrivare a definire lo spazio come un'unica cella e ciò sarebbe intuitivamente inutile, oppure e opportuno che non abbiano buchi al loro interno in quanto renderebbe quasi impossibile una discretizzazione efficace. Da un punto di vista del path-planning però non è ancora una condizione del tutto sufficiente e quindi starà ai vari algoritmi aggiungere altri vincoli di forma e dimensione.
In generale una decomposizione in celle per essere considerata accettabile deve necessariamente soddisfare le seguenti tre proprietà:

1. Trovare un percorso che permetta l'unione di due punti qualsiasi all'interno di una singola cella deve essere facile. Per esempio, se la cella è convessa qualsiasi coppia di punti potrà essere unita da una linea o un segmento
2. L'informazione di adiacenza tra le varie celle deve essere facilmente ricavabile in modo da poter costruire una roadmap.
3. Data una coppia di punti q_I e q_G deve poter essere determinato in modo efficiente quali celle li contengono.

Se la decomposizione soddisfa tutte le proprietà sopra elencate il problema si riduce alla risoluzione di un grafo. I metodi più popolari per tale scopo al momento risultano gli algoritmi di Dijkstra e A* che verranno discussi più approfonditamente più avanti.
Uno dei metodi di decomposizione più utilizzati è la cosiddetta decomposizione verticale. Di seguito sono illustrati i passaggi che la compongono[3]:

- **Definizione di una decomposizione verticale:** Il metodo divide lo spazio delle configurazioni libere $\boldsymbol{C_{free}}$ in celle 1D (segmenti) e 2D (triangoli e trapezoidi) ed è definito nella seguente maniera. Sia P l'insieme dei vertici che rinchiudono lo spazio delle configurazioni con ostacoli $\boldsymbol{C_{obs}}$. Per ogni $p \in P$ si estendano, quando possibile, segmenti verticali verso l'alto e verso il basso all'interno di $\boldsymbol{C_{free}}$ finché non si arriva nuovamente a $\boldsymbol{C_{obs}}$. Si avranno quattro possibili scenari illustrati nella figura sottostante e se $\boldsymbol{C_{free}}$ risulterà diviso in celle al termine di questa operazione si avrà una decomposizione verticale come mostrato in Figura 5. Da notare come accennato in precedenza che la decomposizione è formata solamente da triangoli e trapezoidi, è per questo motivo, infatti, che un altro nome della decomposizione

verticale è anche *Decomposizione trapezoidale.*

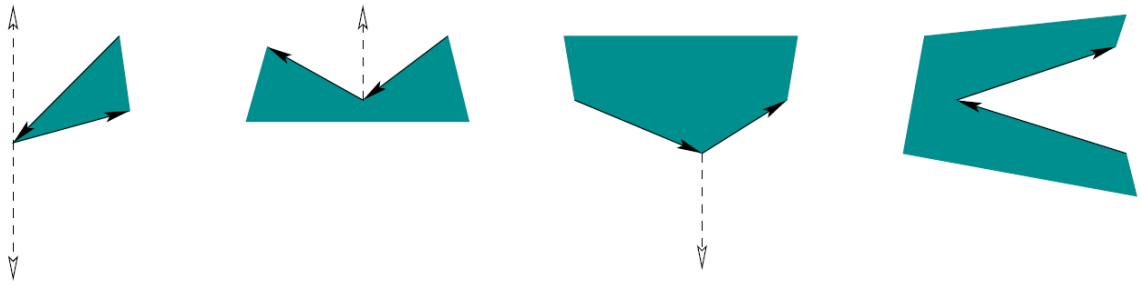


Figura 4: 1)Estensione bidirezionale; 2)Estensione verso l'alto; 3)Estensione verso il basso; 4)Estensione impossibile

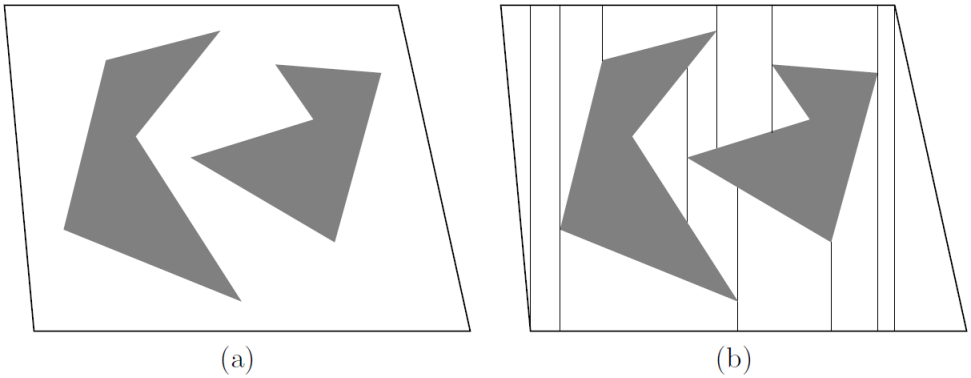


Figura 5: Decomposizione verticale di un generico ambiente

- **Risoluzione di eventuali problemi di posizione:** Un problema che potrebbe capitare usando questo tipo di decomposizione sarebbe quello di due vertici che cadono sulla stessa verticale oppure se direttamente l'intero spigolo dell'ostacolo è verticale in quanto renderebbe la decomposizione molto meno efficiente e performante. In generale per risolvere questo problema si utilizza un metodo molto semplice, ovvero si aggiunge un leggero rumore ai vari vertici facendo si che la possibilità che si allineino su una posizione verticale sia praticamente nulla. Tale metodo non è tuttavia da utilizzare a cuor leggero in quanto intrinsecamente causa un cambio di posizione e quindi necessita soluzioni ad hoc per compensare eventuali errori. Nella stragrande maggioranza dei casi, tuttavia, è possibile tenere sotto controllo tali problemi nella decomposizione verticale.
- **Creazione della roadmap:** Una volta creata la decomposizione è necessario definire quella che sarà la roadmap da risolvere. Per ogni cella C_i sia q_i il punto di campionamento designato al suo interno tale che $q_i \in C_i$. Il punto di campionamento può essere scelto a piacere in base a quelle che sono le esigenze

di partenza ed arrivo, in generale si predilige quello che è il centroide della cella 1D piuttosto che 2D. Sia *G(V,E)* il grafo topologico definito nel seguente modo. Per ogni cella C_i si definisca un vertice $q_i \in V$ in modo che ogni cella indipendentemente dalla dimensione ne abbia uno. Per ogni punto di campionamento all'interno di una cella 2D si faccia partire uno spigolo che lo connetta ad ogni punto di campionamento delle celle 1D che risiedono sul suo confine. Gli spigoli così creati rappresentano dei percorsi che uniscono tra loro tutte le celle adiacenti. La roadmap così creata soddisfa sia le condizioni di accessibilità, dal momento che ogni punto può essere raggiunto da un segmento grazie alla convessità delle celle, che di connettività dal momento che G è stato creato dalla decomposizione che intrinsecamente mantiene la connettività di $\boldsymbol{C_{free}}$. Un esempio di roadmap è illustrato nella figura seguente.

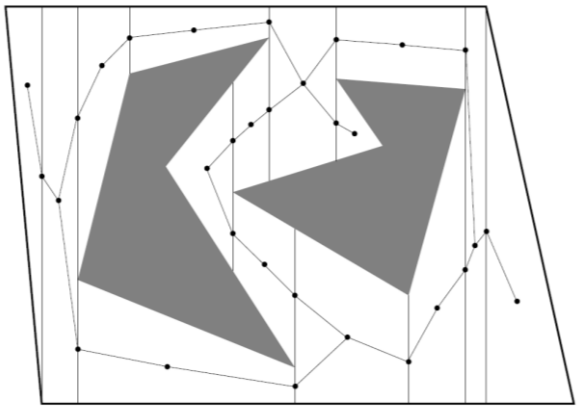


Figura 6: Esempio di roadmap

- **Risoluzione del grafo:** Una volta avuta la roadmap diventa estremamente semplice risolvere il grafo ottenuto per ricavare quello che è il percorso migliore e per fare ciò è possibile utilizzare l'algoritmo che più si desidera. Come già accennato più volte più avanti in questo lavoro di tesi verranno discussi i metodi di Dijkstra e A* che sono sicuramente tra i più diffusi.

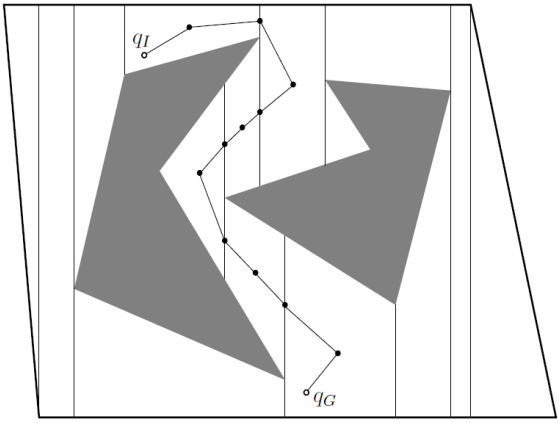


Figura 7: Esempio risoluzione percorso

- **Implementazione algoritmo:** Se da un lato l'algoritmo proposto risulta estremamente semplice dal punto di vista concettuale è anche necessario tenere in considerazione quella che è l'implementazione vera e propria su un calcolatore. Dovendo infatti connettere e far intersecare spigoli generati da n vertici in $\boldsymbol{C_{obs}}$ si avrà un tempo di calcolo nell'ordine $O(n^2)$ e questo ignorando quella che è la struttura dei dati e la localizzazione dei punti di campionamento all'interno delle celle. E' possibile dimostrare che con un attenta gestione del codice si può portare la crescita del tempo nell'ordine $O(n\ log(n))$, ma rimane comunque il fatto che questa come in generale le tecniche grid-based non sono eccessivamente performanti nell'applicazione di problemi più complessi in quanto il costo computazionale cresce comunque troppo rapidamente in relazione al modello.

2.2 Visibility graph algorithms

Non sono quasi mai utilizzati in maniera a sé stante, ma quasi sempre utilizzati per risolvere dei grafi ottenuti con altri metodi come per esempio cell-decomposition o sample-based. Sono quindi una parte fondamentale di quelle che sono le logiche di path-planning odierne. I due principali esponenti di questa categoria e che verranno approfonditi nella sezione seguente sono sicuramente l'algoritmo di Dijkstra e l'algoritmo A*.

2.2.1 Algoritmo di Dijkstra

Come già accennato si tratta di un algoritmo che risolve i percorsi da intraprendere come dei grafi.

L'algoritmo di Dijkstra[4] visita i nodi nel grafo, in maniera simile a una ricerca in ampiezza o in profondità. In ogni istante, l'insieme N dei nodi del grafo è diviso in tre parti: l'insieme dei nodi visitati V, l'insieme dei nodi di frontiera F, che sono successori dei nodi visitati, e i nodi sconosciuti, che sono ancora da esaminare. Per ogni nodo z, l'algoritmo tiene traccia di un valore d_z, inizialmente posto uguale a 1, e di un nodo u_z, inizialmente indefinito.

L'algoritmo consiste semplicemente nel ripetere il seguente passo: si prende dall'insieme F un qualunque nodo z con d_z minimo, si sposta z da F in V, si spostano tutti i successori di z sconosciuti in F, e per ogni successore w di z si aggiornano i valori d_w e u_w. L'aggiornamento viene effettuato con la regola

$$d_w \leftarrow \min\{d_w, d_z + p_a\}$$

dove a è l'arco che collega z a w. Se il valore di d_w è stato effettivamente modificato, allora u_w viene posto uguale z.

La regola segue un'idea piuttosto naturale: se sappiamo che con peso d_z possiamo arrivare fino a z, allora arrivare a w non può costare più di arrivare a z e spostarsi lungo un arco fino a w. L'aggiornamento di u_w ci permette di ricordare che, al momento, il cammino di peso minimo che conosciamo per arrivare da x in w ha come penultimo nodo z.

L'algoritmo parte con $V = \emptyset$, $F = \{x\}$, $dx = 0$ e prosegue finché y non viene visitato, o finché $F = \emptyset$: in questo caso, y non è raggiungibile da x lungo un arco orientato. Se usciamo solo nel secondo caso, alla fine dell'algoritmo d_z contiene, per ogni nodo z, il peso di un cammino minimo da x a z; inoltre, il vettore u permette di ricostruire l'*albero dei cammini minimi con origine in x*.

Per meglio far comprendere il funzionamento è opportuno riportare un esempio[4] svolto passa per passo. L'obbiettivo è trovare il cammino minimo dal nodo 0 al nodo 5 e la configurazione iniziale è la seguente:

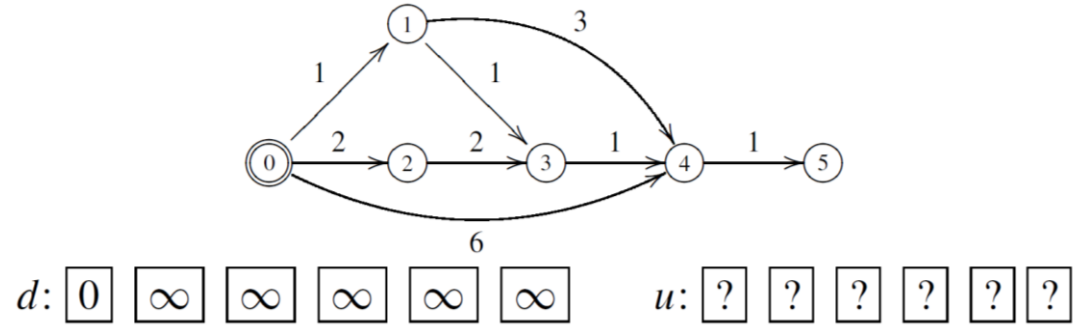


Figura 8: Configurazione iniziale

Al primo passo, il nodo 0 viene visitato (nella figura, viene barrato), mentre 1, 2 e 4 passano nella frontiera (denotata da un ulteriore cerchio attorno al nodo).

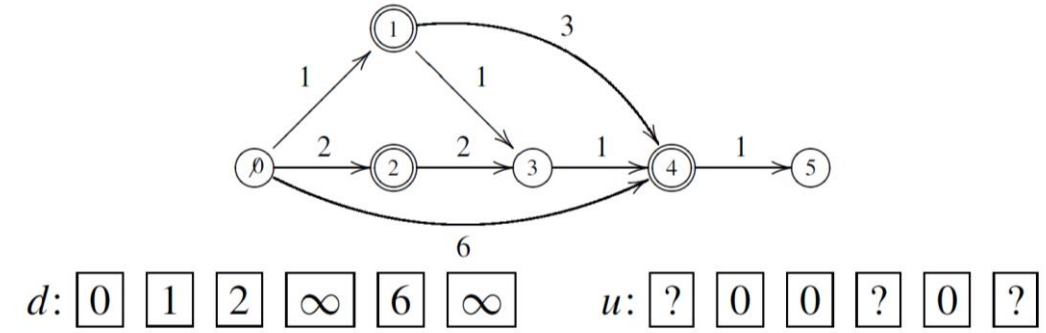


Figura 9: Passo 1

Ora dobbiamo andare a prendere il nodo della frontiera che ha distanza minima da 0, e possiamo vedere come l'algoritmo di Dijkstra risolva i problemi posti da una visita in

ampiezza: essendo i nodi 3 e 4 molto distanti da 0 (in termini di cammini di peso minimo), prima visitiamo il nodo 1, il che ha l'effetto di aggiornare d_3, d_4, u_3 e u_4:

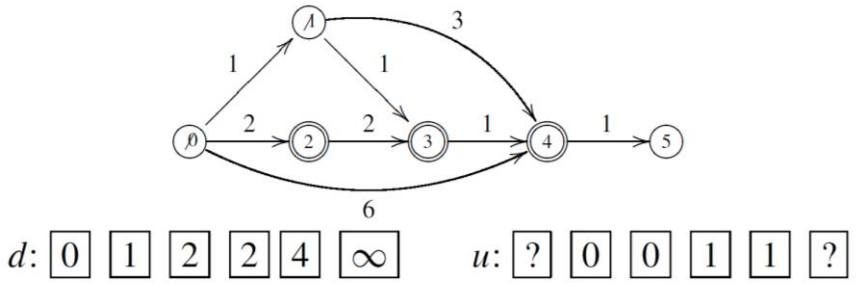


Figura 10: Passo 2

Ancora una volta, invece di scegliere i nodi aggiunti inizialmente alla frontiera, proseguiamo con quello che ha d minimo, cioè 2. L'effetto è solo quello di spostare il nodo tra quelli visitati, dato che per l'unico successore (il nodo 3) conosciamo già un cammino di peso minimo migliore di quello che passerebbe per il nodo 2:

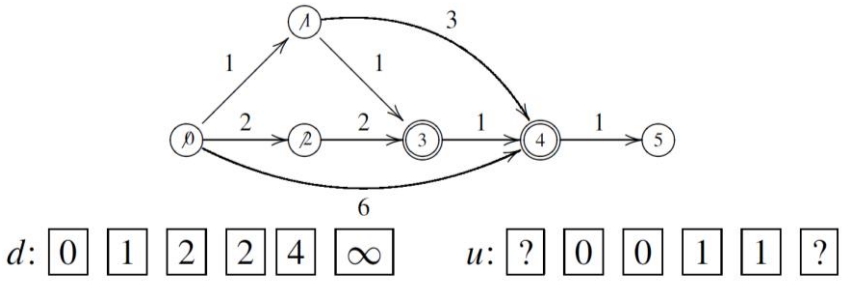


Figura 11: Passo 3

Visitiamo ora il nodo 3, aggiornando nuovamente d_4 e u_4:

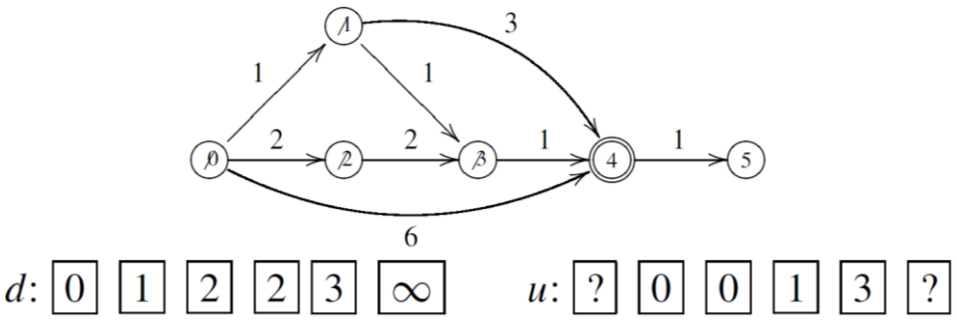


Figura 12: Passo 4

A questo punto visitiamo il nodo 4:

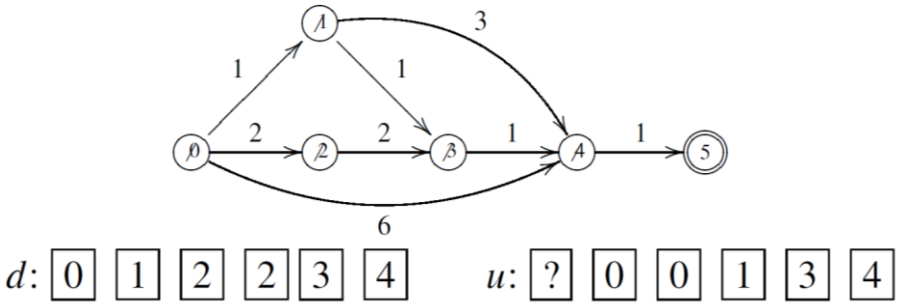


Figura 13: Passo 5

La visita successiva ci dirà che esiste un cammino da 0 a 5 di distanza 4: inseguendo i puntatori di *u* all'indietro è immediato leggere che il cammino sarà 0, 1, 3, 4, 5.

Per dimostrare la correttezza dell'algoritmo è necessario dimostrare che le seguenti proprietà vengano rispettate ad ogni passo:

1. I nodi in *F* sono tutti e soli i successori non ancora visitati di nodi visitati (cioè sono i successori di nodi in *V,* a parte quelli già in *V*).
2. Per ogni nodo *z* in *V*, d_z è il peso di un cammino minimo da *x* in *z;* inoltre, uno dei cammini minimi ha come penultimo nodo u_z.
3. Per ogni nodo *z* in *F*, d_z è il peso minimo di un cammino di *x* in *z* che passa solo per nodi in *V,* ad esclusione di *z* stesso; inoltre, tale cammino ha come penultimo nodo u_z.

Le proprietà sono vere in maniera ovvia al primo passo, ed è anche banale che la proprietà (1) rimanga sempre vera dopo il primo passo.

Dobbiamo innanzitutto mostrare che quando spostiamo un nodo *z* con d_z minimo da *F* in *V*, d_z è il peso di un cammino minimo da *x* a *z*. Infatti, se per assurdo ci fosse un cammino di peso inferiore *d*, esso dovrebbe partire da *x*, viaggiare entro *V*, passare per qualche nodo *z*' in *F* diverso da *z* (altrimenti la proprietà (3) sarebbe violata), e infine arrivare in *z*. Ma allora il cammino da *x* a *z*' avrebbe peso inferiore a d_z , e quindi $d_{z'} < d_z$, contrariamente a quanto assunto. Quindi la proprietà (2) è ancora verificata.

Dobbiamo ora dimostrare che (3) rimane vera. Per i nodi in *F* che non sono successori di *z*, la proprietà rimane ovviamente vera, perché l'aggiunta di *z* a *V* non ha generato nuovi cammini che li raggiungono entro *V*. Per gli altri nodi, la regola di aggiornamento di *d* e *u* fa sì che la proprietà (3) rimanga vera.

Concludiamo che non appena *y* cade dentro *V*, possiamo fermare l'algoritmo e restituire d_y come peso di un cammino minimo. Inoltre, per costruire un cammino minimo ci basta inseguire all'indietro i puntatori u_y, u_{uy}, ecc., fino a *x*. Dato che qualunque porzione di un cammino di peso minimo è a sua volta un cammino di
peso minimo (o potremmo sostituire la porzione con una di peso minore, ottenendo un cammino complessivo più breve), in questo modo ricostruiremo un cammino minimo da *x* a *y*.
Infine, se prima che *y* cada dentro *V* succede che $F = \emptyset$, allora la proprietà (1) garantisce che non ci sono altri nodi raggiungibili da *x*, e che, in particolare, *y* non è raggiungibile da *x*.

2.2.2 Algoritmo A*

Per applicazioni di AGV può essere quasi visto come un'evoluzione naturale del già noto algoritmo di Dijkstra. L'algoritmo A*[5] si basa su quella che è una ricerca "euristica", ovvero, focalizza la risoluzione del grafo nella direzione corretta attraverso l'aggiunta di una funzione euristica. Più precisamente la funzione euristica è una funzione che stima quello che è il costo di un percorso attraverso un dato nodo. In particolare, per essere considerata valida tale funzione deve essere "ottimistica", ovvero, sia *h(s,x)* la funzione euristica che stima il costo per andare dal nodo *x* al nodo soluzione *s*, tale funzione sarà considerata ottimistica e quindi valida se e solo se:

$$\forall x \in V: h(s,x) \leq g(s,x)$$

dove *V* è l'insieme dei nodi e *g* è l'effettivo costo per spostarsi da *x* a *s.* Dalla funzione euristica l'algoritmo procede poi alla ricerca con un approccio detto "best-first" ovvero dando precedenza ai percorsi con euristica minore. Per poter funzionare l'algoritmo ha bisogno dei seguenti requisiti[6]:

- La soluzione deve essere esprimibile e rappresentabile sotto forma di grafi;
- Devono essere noti a priori nodo iniziale e nodo finale oppure talvolta sono sufficienti le regole per arrivare alla soluzione finale (oltre ad applicazioni di AGV l'algoritmo A* è molto efficiente a risolvere problemi come cubi di Rubik e Sudoku);
- Deve essere nota e valida la funzione euristica da utilizzare nell'algoritmo;

- Deve essere noto il costo effettivo associato ad ogni transizione tra nodi (condizione banalmente implicita poiché necessaria per la validità della condizione precedente).

Rispettate queste condizioni l'algoritmo va ad utilizzare due strutture per la risoluzione:

- Una lista di nodi già visitati
- Una coda di priorità contenente i nodi da visitare.

Durante l'esecuzione ad ogni nodo vengono associati i seguenti valori:

- *g(n):* È il minimo costo reale e necessario per arrivare dal nodo di partenza al nodo *n*;
- *h(n):* È la funzione euristica che stima il costo mancante per poter raggiungere il nodo di arrivo dal nodo *n*.
- *f(n) = g(n)+h(n):* È la funzione che definisce la priorità del nodo. Altro non è che la stima per difetto di quello che sarà il costo totale del percorso qualora si decidesse di passare per il nodo *n*. Ovviamente l'algoritmo espanderà prima i nodi con *f* minore.

Prima di introdurlo in maniera più formale come per i precedenti algoritmi risulta comodo utilizzare un esempio[5] per capire in maniera più semplice e intuitiva come l'algoritmo A* lavori. Si prenda come esempio il grafo sottostante:

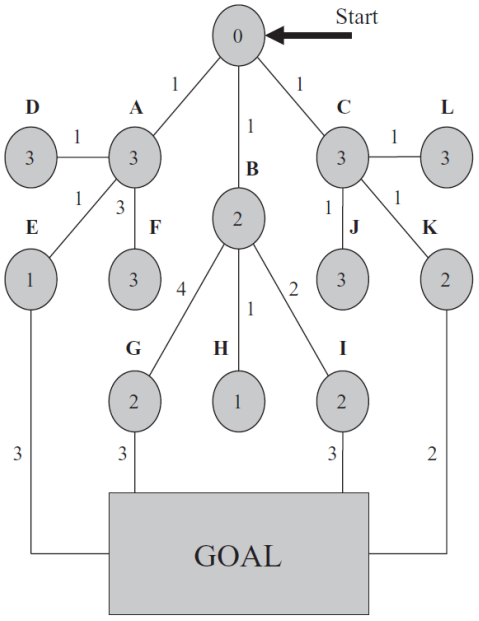


Figura 14: Esempio di grafo

Il primo nodo a dover essere inserito nella coda di priorità è ovviamente il nodo start. Al passo successivo si espande l'algoritmo incorporando nella coda i nodi non ancora visitati

confinanti con il nodo start e sistemandoli in ordine di priorità (sui rami si legge il costo *g* reale mentre all'interno dei nodi si può leggere l'euristica *h* del nodo stesso).

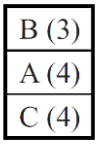


Figura 15: Espansione nodo start

Visto che il nodo B è quello con la priorità più alta (*f* minore di tutti) viene espanso per primo. Viene perciò rimosso dalla coda e sostituito dai suoi nodi confinanti non ancora visitati che vengono inseriti in base al loro ordine di priorità.

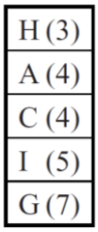


Figura 16: Espansione nodo B

Al passo successivo si procede con l'espansione del nodo H che però essendo un vicolo cieco viene semplicemente rimosso dalla coda. Si passa quindi al nodo A che, come in precedenza, verrà sostituito dai suoi vicini non ancora visitati in ordine di priorità. Successivamente verrà infine espanso E che porterà ad un percorso completo di costo 5.

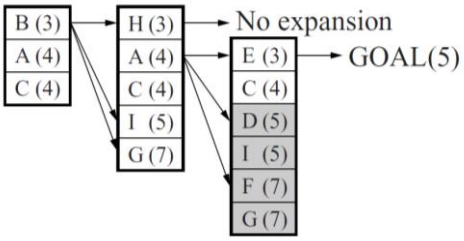


Figura 17: Prima serie di espansioni

Anche se un percorso è stato trovato non è detto che sia il migliore possibile quindi la ricerca non può considerarsi ancora conclusa. È tuttavia possibile effettuare forti semplificazioni tra i nodi rimasti evitando di controllare quelli con $f > 5$ (celle scure). Ciò è possibile grazie alla definizione di funzione euristica data in precedenza; infatti, *f* altro non è che una stima del costo totale effettuata per difetto per cui, se la funzione *h* è correttamente definita, tutti i nodi con $f >= 5$ porteranno per forza ad un percorso più costoso di quello già trovato. D'altro canto, è opportuno controllare i rimanenti nodi con

$f < 5$ dal momento che potrebbero portare a soluzioni migliori. A questo punto l'unico nodo espandibile è il nodo C che porta ad una successiva espansione di K ed infine a quello che effettivamente è il percorso migliore possibile.

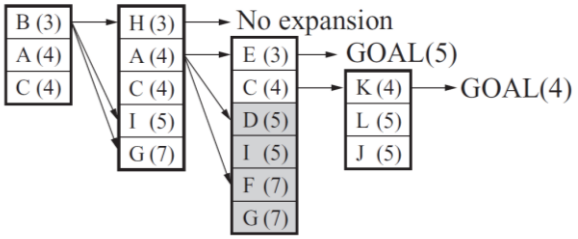


Figura 18: Espansione completa

L'algoritmo risulta estremamente efficiente per la risoluzione dei grafi e volendo può essere sintetizzato dai seguenti 8 punti[7]:

1. Inserimento nella coda del nodo di partenza con priorità pari a *f*.
2. Se la coda è vuota, l'algoritmo termina: soluzione non trovata.
3. Estrazione del miglior nodo da visitare (priorità con valore più basso).
4. Se il nodo estratto ha *h* nullo, l'algoritmo termina: soluzione trovata.
5. Costruzione dei nodi figli.
6. Eliminazione dei nodi figli già visitati e sub-ottimi.
7. Inserimento dei nodi rimanenti nella coda con priorità pari a *f*.
8. Tornare al punto 2.

Oppure in forma di pseudo-codice[5]:

Input: A graph
Output: A path between start and goal nodes

1: **repeat**
2: Pick n_{best} from O such that $f(n_{best}) \leq f(n), \forall n \in O$.
3: Remove n_{best} from O and add to C.
4: If $n_{best} = q_{goal}$, EXIT.
5: Expand n_{best}: for all $x \in \text{Star}(n_{best})$ that are not in C.
6: **if** $x \notin O$ **then**
7: add x to O.
8: **else if** $g(n_{best}) + c(n_{best}, x) < g(x)$ **then**
9: update x's backpointer to point to n_{best}
10: **end if**
11: **until** O is empty

*Figura 19: Pseudo-codice algoritmo A**

In sintesi, l'algoritmo A* è estremamente potente, l'assenza di cicli chiusi da indagare garantisce la completezza, mentre il sistema della funzione euristica garantisce priorità di

ricerca di quella che è la soluzione migliore. Garantisce inoltre risultati ottimizzati in quanto non si ferma alla risoluzione di un singolo percorso, ma continua la ricerca finché non svuota la coda di tutti i nodi con *f* minore del costo dell'attuale percorso migliore.
Piccola nota aggiuntiva riguarda la scelta di una pessima euristica come può esserlo una funzione costante. In quel caso, infatti, si avrebbe un algoritmo che basa le decisioni esclusivamente su quello che è il costo effettivo dei rami riportandoci di fatto nel caso precedente di algoritmo, ovvero Dijkstra.

2.3 Sample-based algorithms

Sicuramente la categoria di algoritmi più popolare al giorno d'oggi prevalentemente grazie alle innegabili doti di efficienza e velocità che li distinguono. La loro forza sta nel fatto che a differenza delle tecniche precedenti, dove l'intero spazio delle configurazioni è esplorato, negli algoritmi sample-based si analizza il problema in maniera stocastica permettendo di avere ottimi risultati con costi computazionali estremamente ridotti. I due principali algoritmi di questa categoria sono sicuramente il metodo Probabilistic Roadmap (PRM) e Rapidly-exploring random tree (RRT). Nel seguente paragrafo verranno analizzati questi due algoritmi oltre alle loro versioni ottimizzate PRM* e RRT*. Verrà infine anche analizzato l'algoritmo Kinodynamic-RRT* che permette l'applicazione di queste tecniche anche in presenza di vincoli anolonomi e quindi cinodinamici come suggerisce appunto il nome.
Gli algoritmi appartenenti a questa categoria per funzionare correttamente si basano su due passaggi fondamentali comuni a tutti i metodi più popolari ed alle loro varie mutazioni:

- **Fase di Learning:** È la fase in cui vengono selezionati stocasticamente i nodi appartenenti allo spazio delle configurazioni libere C_{free} e collegati tramite opportuni archi.
- **Fase di Quering:** È la fase in cui tra tutti i percorsi disponibili si sceglie il migliore possibile. È in questa fase che ritornano particolarmente utili alcuni dei metodi di risoluzione dei grafi visti in precedenza.

Gli algoritmi PRM e RRT hanno entrambi queste due fasi, la vera differenza tra i due e le loro seguenti mutazioni sta nel come vengono svolte le due fasi con i conseguenti vantaggi e svantaggi che ne possono derivare.

2.3.1 Algoritmo PRM (Probabilistic Roadmap)

È la versione di algoritmo sample-based più "generale". Tra le due famiglie proposte si tratta infatti dell'algoritmo che meglio si presta ad applicazioni multi-query, ovvero alla risoluzione più generale appunto dell'ambiente rendendo più efficiente la creazione di più percorsi con più regioni di partenza e di arrivo piuttosto che un singolo percorso. Il motivo

risulterà evidente non appena verrà spiegato meglio come entrambe le metodologie affrontano la così detta fase di Learning. Come già detto in precedenza infatti ogni algoritmo sample-based è caratterizzato da questi due macro-fasi.

2.3.1.1 Learning

Probabilmente la macro-fase più importante essendo quella che determina il tipo di algoritmo di cui si parla, ma soprattutto quella che sarà l'efficienza della soluzione finale per la data applicazione. Questa, infatti, è la fase in cui viene costruita quella che si definisce la *Roadmap* da risolvere per trovare il percorso finale. La Roadmap altro non è che un grafo $G(V,E)$ formato da nodi V uniti da archi e linee E. Esistono vari modi di crearlo e in questo paragrafo verrà brevemente illustrata quella che è la costruzione basilare per l'algoritmo PRM.
All' inizio il grafo è ovviamente vuoto e andrà riempito man mano estraendo casualmente nuovi nodi dallo spazio delle configurazioni libere C_{free}. Il primo nodo per ovvie ragioni non sarà soggetto a particolari vincoli da rispettare oltre, ovviamente all'appartenenza a C_{free}, con i nodi successivi invece inizierà la vera e propria costruzione del grafo dal momento che andranno progressivamente connessi tra di loro. Le congiunzioni dei nuovi nodi con quelli già esistenti sono gestite da un pianificatore locale che può seguire diverse regole in base agli scopi desiderati e alla complessità dello spazio delle configurazioni. Per un algoritmo PRM generico il pianificatore locale funziona nella seguente maniera. Si crea una bolla di raggio r attorno al nuovo nodo q_{rand} generato, a questo punto per poter connettere il nodo ai nodi già presenti è necessario che entrambi i nodi siano connessi a due componenti diverse del grafo in modo da evitare cicli che specialmente nel caso di spazi molto complessi o con archi percorribili solo in un senso potrebbero causare problemi di inefficienza non banali, ma soprattutto l'arco che eventualmente andrebbe a connetterli deve cadere anch'esso all'interno di $C_{free,}$ per il banale motivo secondo cui la traiettoria deve essere priva di collisioni come il nodo stesso. Infine, si uniscono i nodi tra loro tramite segmenti, archi o qualsivoglia metodo il pianificatore locale preveda.
Risulta chiaro dopo la breve introduzione dell'algoritmo che l'efficacia o meno di quest'ultimo può essere condotta principalmente a due fattori:

- **Numero N di nodi da scegliere casualmente:** Gli algoritmi PRM come tutti gli algoritmi sample-based sono probabilisticamente completi, ovvero tendono ad avere la proprietà di completezza per un numero di nodi e archi che tende ad

infinito; perciò il numero massimo di nodi con cui costruire il grafo deve essere grande abbastanza da garantire una buona copertura dell'ambiente e possibilmente l'assenza di più componenti, ma non deve comunque eccedere in quanto risulta palese che oltre un certo punto il costo computazionale non giustificherebbe un aumento di precisione irrisorio.

- **Il pianificatore locale:** È quasi banale dirlo, ma il pianificatore locale è quello che può definirsi il cuore dell'algoritmo stesso; infatti, spesso il nome della variante dell'algoritmo originale deriva dalle diverse policy del pianificatore locale. Un pianificatore locale con delle policy correttamente adeguate a quella che è l'applicazione desiderata possono fare una differenza abissale in termini di efficienza computazionale e risultato finale. Rispetto alle condizioni basilari illustrate precedentemente alcuni metodi di valutazione e connessione dei nodi utilizzati dal pianificatore locale possono essere:
    - ***k*-Nearest PRM:** Particolarmente utilizzato per quelli che possono essere grafi molto fitti. Semplicemente oltre alla condizione di "vicinanza" data dal raggio *r* della bolla locale aggiunge una condizione di "affollamento", ovvero verifica solamente i *k* nodi più vicini all'interno della bolla.
    - **Bouded-degree PRM:** Condizione simile a quella appena accennata, semplicemente anzi che limitare il numero di nodi sondati limita il numero di tentativi di connessione.
    - **Variable-radius PRM:** Anch'essa variante che similmente alle precedenti tenta di mantenere ridotto il numero di vertici e di nodi analizzati ad ogni iterazione. Come dice abbastanza chiaramente il nome in questo caso l'idea è di cambiare progressivamente il raggio all'aumentare dei nodi *n* nel grafo. Questo approccio verrà discusso meglio più avanti quando si parlerà di algoritmo PRM*.

In sintesi, la fase di learning con tutti i suoi accorgimenti può essere sintetizzata nei seguenti sette punti[2]:

1. Inizializzazione del grafo $G(V,E)$ come insieme vuoto ed inserimento del primo vertice q_1 dallo spazio delle configurazioni libere C_{free}.
2. Generazione di un nodo casuale q_{rand} anch'esso proveniente da $C_{free.}$
3. Annessione di tutti i nodi contenuti all'interno della bolla di raggio r e centro q_{rand} all'insieme U dei nodi limitrofi.
4. Annessione di q_{rand} all'insieme dei vertici V.

5. Tutti i nodi di U vengono connessi a q_{rand} se rispettano le due condizioni precedentemente specificate; ovvero il percorso congiungente i due nodi deve essere interamente in C_{free} e i due nodi devono appartenere a due componenti diverse.
6. Qualora le connessioni risultassero possibili gli archi che la permetterebbero verranno aggiunti all'insieme E del grafo.
7. Raggiunto il numero massimo N di iterazioni desiderate l'algoritmo termina restituendo la roadmap composta dal grafo $G(V,E)$.

Di seguito l'algoritmo di learning è rappresentato in figura come pseudo-codice:

$V \leftarrow \emptyset$; $E \leftarrow \emptyset$;
for $i = 0, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $U \leftarrow \texttt{Near}(G = (V, E), x_{\text{rand}}, r)$;
  $V \leftarrow V \cup \{x_{\text{rand}}\}$;
  foreach $u \in U$, *in order of increasing* $\|u - x_{\text{rand}}\|$, **do**
    if x_{rand} *and* u *are not in the same connected component of* $G = (V, E)$ **then**
      if $\texttt{CollisionFree}(x_{\text{rand}}, u)$ **then** $E \leftarrow E \cup \{(x_{\text{rand}}, u), (u, x_{\text{rand}})\}$;
return $G = (V, E)$;

Figura 20: Pseudo-codice algoritmo di learning PRM

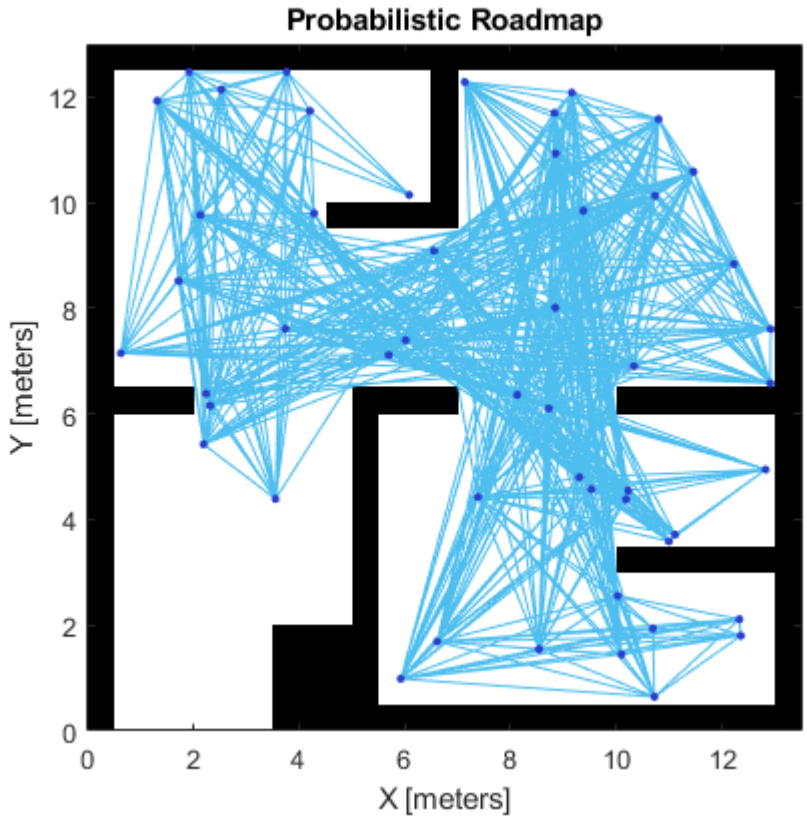


Figura 21: Esempio di generica Roadmap

2.3.1.2 Quering

Una volta completata la fase di learning risulta non meno importante la fase appunto di quering in cui si risolve il grafo creato in precedenza per trovare il percorso ottimale. Come prima cosa e necessario aggiungere i nodi q_{start} e q_{goal} all'insieme V in modo da poterli connettere al resto del grafo. Per quanto riguarda il nodo di arrivo nella pratica è raro si usi un nodo solo, ma è solitamente preferibile indicare una zona di arrivo dove qualunque nodo al suo interno può essere ritenuto valido come percorso. Ciò è fatto anche in vista dell'eventuale presenza di ostacoli imprevisti in prossimità del punto di arrivo in modo da avere più flessibilità, per esempio, sarebbe ridicolo considerare una simulazione fallita qualora un robot fosse costretto a fermarsi qualche centimetro più in là rispetto al punto di arrivo previsto per colpa di un dosso o di una pozzanghera. Scelti quindi la partenza e l'arrivo e ipotizzando di avere un grafo robusto ed efficiente dal passaggio precedente e possibile passare alla risoluzione vera e propria. Essa è appunto la risoluzione di un grafo e viene per tanto effettuata con i metodi già elencati in precedenza come l'algoritmo di Dijkstra o A*.

2.3.2 Algoritmo PRM* (PRM ottimizzato)

Si tratta di una delle versioni più recenti dell'algoritmo PRM ed è stata proposta per la prima volta nell'articolo scientifico "*Sampling-based Algorithms for Optimal Motion Planning"[8]* di S. Karaman ed E. Frazzoli. Si tratta di un metodo ottimizzato che sfrutta un raggio di prossimità mutevole con quella che è la dimensione attuale del grafo.
Prima di spiegare più dettagliatamente come effettivamente avvenga il cambio di raggio con il proseguire delle iterazioni e quindi della dimensione del grafo è opportuno fare un passo indietro per riprendere quello che è l'algoritmo di learning classico del metodo PRM. L'algoritmo PRM* infatti si basa su una versione leggermente semplificata dell'algoritmo classico, ovvero l'algoritmo sPRM (simplified PRM). Il funzionamento è molto simile a quello dell'algoritmo standard, l'unica vera differenza e che l'algoritmo sPRM ammette la presenza di cicli mentre nella versione originale non erano permesse più di una sola connessione tra un nodo e una componente del grafo. Se da un lato risulta ovvia la semplificazione in termini computazionali che ne deriva avendo una condizione in meno dall'altro sembrerebbe ovvia una performance peggiore o addirittura fallimentare

vista la possibile presenza di cicli. Tuttavia, la pratica sperimentale ha mostrato che in ambienti adeguati e soprattutto grazie alla modifica PRM* risulta estremamente più efficiente e performante della versione classica.

Come già accennato l'algoritmo PRM* fa della sua forza il fatto di usare un raggio di prossimità per definire l'insieme *U* mutevole. Più precisamente caratterizzato da quella che è una dipendenza esponenzialmente decrescente con il numero di iterazioni *n* dell'algoritmo e con i gradi di libertà del sistema che si desidera controllare. Il ragionamento alla base risulta estremamente potente nonostante la sua semplicità, è infatti immediato rendersi conto che le prime iterazioni caratterizzate da pochi punti non connessi potranno permettersi di utilizzare un raggio più ampio per monitorare lo spazio libero mentre con il procedere delle iterazioni arrivando quindi ad avere uno spazio pieno di punti probabilmente già connessi risulterebbe inutile controllare regioni eccessivamente ampie che quasi sicuramente non darebbero comunque una ricerca fruttuosa.

La relazione utilizzata nell'articolo è la seguente:

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

Dove:

- ***n:*** Numero di iterazioni e quindi di nodi in *V*.
- ***d:*** Dimensione dello spazio delle configurazioni considerato.
- **γ_{PRM}:** È un fattore moltiplicativo che tiene conto dei gradi di libertà che danno la dimensione dello spazio delle configurazioni e la misura di Lebesgue dello spazio C_{free}. È definito come:

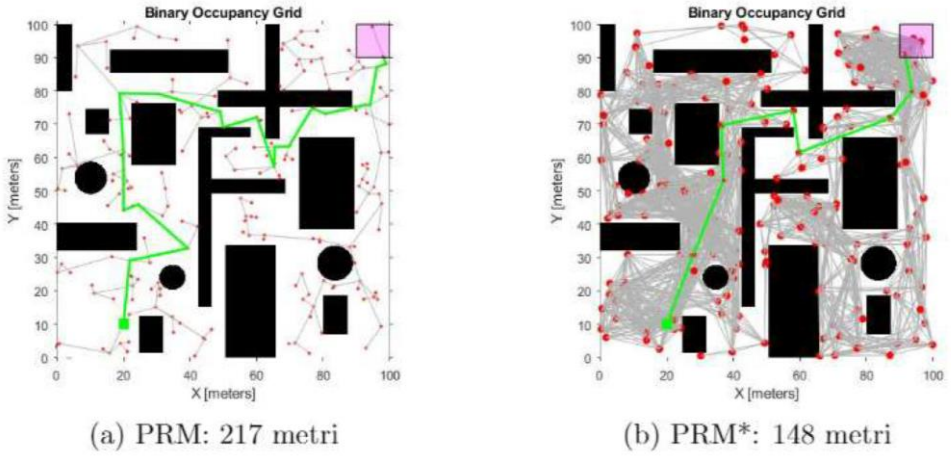
$$\gamma_{PRM} > \gamma_{PRM^*} = 2\left(1 + \frac{1}{d}\right)^{\frac{1}{d}} \left[\frac{\mu(C_{free})}{\zeta_d}\right]^{\frac{1}{d}}$$

  - **$\mu(C_{free})$:** Misura di Lebesgue dello spazio delle configurazioni libere C_{free}. È la massima misura definibile nello spazio di riferimento, per esempio area nello spazio 2D, volume nello spazio 3D e via di seguito.
  - **C_{free}:** Spazio delle configurazioni libere da ostacoli o comunque fisicamente realizzabili.
  - **ζ_d:** È il volume della bolla di raggio unitario nello spazio Euclideo di dimensione *d.* È definita come:

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

- $\Gamma(x) = (x-1)!$ È la funzione gamma.

Quello che all'apparenza può sembrare un cambiamento relativamente semplice in realtà è dimostrato nella pratica che da risultati molto più efficienti e veloci a livello computazionale. La figura sottostante[2] mostra la chiara superiorità dell'algoritmo PRM* a parità di ambiente e nodi:



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

*Figura 22: Confronto tra PRM e PRM**

2.3.3 Algoritmo RRT (Rapidly-exploring random tree)

È l'altro grande esponente della categoria degli algoritmi sample-based. Come già accennato l'unica vera differenza con l'algoritmo PRM e le sue versioni sta nella fase di learning. Se infatti l'algoritmo PRM creava e connetteva i nodi in maniera completamente stocastica e solo in seguito annetteva i nodi di partenza e di arrivo l'algoritmo RRT invece inizia a costruire il grafo inserendoli già nella fase di learning. Inoltre, se nell'algoritmo PRM i nodi randomici venivano annessi al grafo e poi in seguito avevano la possibilità di essere connessi nell'algoritmo RRT non è così. Nell'algoritmo RRT i nodi generati in maniera stocastica servono più che altro a dare la direzione in cui si svilupperà il grafo; infatti, più che il nodo stesso ad ogni iterazione è più probabile annettere un nodo che ne è "figlio". La generazione di questo nodo derivato nella pratica si realizza tramite una funzione chiamata *Steering function* il cui scopo e sostanzialmente espandere il grafo attuale verso il nuovo nodo di una distanza "ragionevole". Il termine ragionevole non è usato a sproposito in quanto gioca un ruolo determinante in quella che può essere

l'efficienza globale dell'algoritmo. La ragione di ciò è dovuta alla possibilità di scartare un nodo anche qualora si trovi nello spazio C_{free}, questo perché come già accennato il nodo più che una posizione valida deve restituire una direzione valida. Per stabilire la validità di una direzione la steering function altro non fa che tentare di creare una traiettoria per unire il nodo più vicino nel grafo con il nuovo nodo creato, e se la traiettoria calcolata dovesse attraversare un qualsivoglia ostacolo farebbe si che l'intero nodo venga scartato. Ciò rischierebbe chiaramente di rendere l'algoritmo incredibilmente oneroso ed inefficiente in quanto in ambienti complessi sarebbe quasi impossibile riuscire in maniera consistente a trovare percorsi fattibili anche lunghi in una sola iterazione. Inserendo invece una condizione di distanza massima e facendo diventare quindi il problema di natura locale è possibile a quasi ogni iterazione ricavare una direzione corretta. In un linguaggio con una matematica leggermente più rigorosa possiamo quindi affermare che; dati due nodi q_y, nodo randomico esterno al grafo e q_x, generico nodo del grafo la steering function creerà un nodo q_z con le seguenti caratteristiche[2]

$$\nabla \| q_z - q_y \| = 0$$

$$\| q_z - q_x \| < \eta$$

Dove η è il parametro che indica la distanza ammissibile tra i vari parametri che caratterizzano i nodi e può passare dall'essere il semplice raggio di un cerchio in un caso bidimensionale fino ad assumere valori sempre più complessi a seconda dei gradi di libertà del sistema.

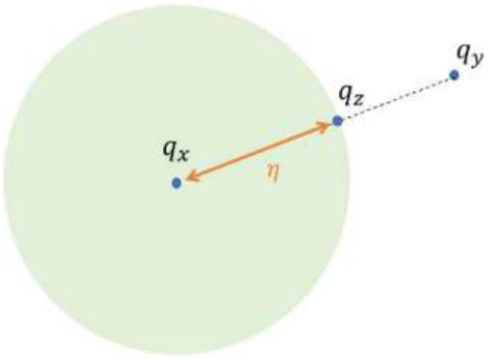


Figura 23: Esempio funzionamento Steering function

Altra caratteristica interessante che viene spontaneo notare dal funzionamento della steering function è l'asimmetria della soluzione che si avrebbe partendo da *x* piuttosto che da *y*. Questa proprietà risulta estremamente vantaggiosa nella successiva fase di quering in quanto tiene implicitamente conto del senso di percorrenza dei rami del grafo ed è quindi uno strumento estremamente potente per evitare cicli interni e soprattutto garantire

una soluzione chiara.

In sintesi, è possibile riassumere l'algoritmo di learning di un generico RRT nei seguenti punti:

1. Creazione del grafo *G(V,E)* come insieme vuoto ed inserimento del nodo di partenza q_{start} dallo spazio delle configurazioni C_{free} in V.
2. Generazione di un nodo q_{rand} proveniente anch'esso da C_{free}.
3. Identificazione del nodo $q_{nearest}$ all'interno del grafo G più vicino a q_{rand},
4. Calcolo del nuovo nodo q_{new} tramite *Steering function*
5. Validazione del percorso tra $q_{nearest}$ e q_{new} in modo che non produca collisioni.
6. Annessione di q_{new} a V e dell'arco a E.
7. Raggiunto il numero massimo di iterazioni n l'algoritmo termina e restituisce *G(V,E)*.

Sotto è riportato l'algoritmo anche in forma di pseudo codice.

```
1: V ← {q_start}; E ← ∅;
2: for i = 1, ··· , maxNodes do
3:    q_rand ← SampleFree;
4:    q_nearest ← Nearest(G = (V, E)), q_rand)
5:    q_new ← Steer(q_nearest, q_rand)
6:    if CollisionFree(q_nearest, q_new) then
7:       V ← V ∪ {q_new}; E ← E ∪ {(q_nearest, q_new)}
8:    end if
9: end for
10: return G = (V, E);
```

Figura 24: Pseudo codice algoritmo RRT

Da notare che l'algoritmo proposto finora è sempre stato fatto espandere dal nodo start, ma una variante dell'algoritmo prevede che sia possibile far crescere contemporaneamente un albero che parte dal nodo q_{start} e uno che parte dal nodo q_{goal} per poi unirli a metà strada. Questa particolare variante è chiamata Rapidly-exploring Dense Trees (RDT).

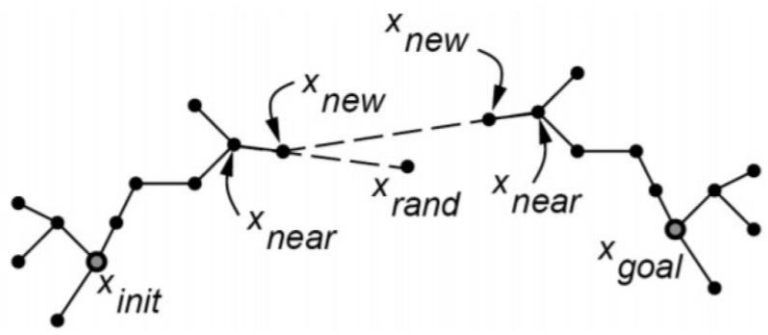


Figura 25: Esempio algoritmo RDT

2.3.4 Algoritmo RRT* (RRT ottimizzato)

Proposto anch'esso nell'articolo "*Sampling-based algorithms for optimal motion planning"[8]* di S. Karaman ed E. Frazzoli tratta una versione ottimizzata dell'algoritmo RRT appena visto.

Come per il caso dell'algoritmo PRM* anche l'algoritmo RRT* si basa su una versione semplificata o comunque all'apparenza meno robusta dell'algoritmo originale. In questo caso la versione non ottima dell'algoritmo proposta si chiama *Rapidly-exploring Random Graph* (RRG).

L'algoritmo RRG si differenzia dall'algoritmo RRT per un semplice passaggio in più, più precisamente ogni volta che verrà aggiunto un nodo con annesso arco al grafo G il nuovo nodo q_{new} verrà anche collegato a tutti gli altri nodi nelle sue vicinanze (purché la cosa non generi collisioni ovviamente) oltre a q_{new}. La regola con cui si stabilisce la "vicinanza" di un nodo è molto simile a quella usata per l'algoritmo PRM*; in particolare la formula che dà la dimensione del raggio è identica, è semplicemente necessario sostituire i pedici PRM con RRG per chiarezza, però si pone un limite superiore pari a η, ovvero il parametro utilizzato nell'algoritmo RRT originale per definire una ragionevole distanza da $q_{nearest}$. In termini più matematicamente rigorosi

$$r\big(card(V)\big) = min\left\{\gamma_{RRG}\left(\frac{\log\left(card(V)\right)}{card(V)}\right)^{\frac{1}{d}}, \eta\right\}$$

Il risultato finale è un grafo con gli stessi vertici che avrebbe l'algoritmo RRT, ma con molti più archi e segmenti che possono anche permettere dei cicli. Di seguito è riportato l'algoritmo in forma di pseudo codice con il semplice scopo di chiarire meglio il funzionamento dell'algoritmo RRG.

```
V ← {x_init}; E ← ∅;
for i = 1, ..., n do
    x_rand ← SampleFree_i;
    x_nearest ← Nearest(G = (V, E), x_rand);
    x_new ← Steer(x_nearest, x_rand) ;
    if ObtacleFree(x_nearest, x_new) then
        X_near ← Near(G = (V, E), x_new, min{γ_RRG(log(card(V))/ card(V))^(1/d), η}) ;
        V ← V ∪ {x_new}; E ← E ∪ {(x_nearest, x_new), (x_new, x_nearest)} ;
        foreach x_near ∈ X_near do
            if CollisionFree(x_near, x_new) then E ← E ∪ {(x_near, x_new), (x_new, x_near)}
return G = (V, E);
```

Figura 26: Pseudo codice algoritmo RRG

Anche se può sembrare folle appesantire in maniera così deliberata l'algoritmo originale dopo tutti i pregi che sono stati attribuiti alle strutture ad albero l'ultima operazione risulta necessaria per poter effettuare quella che è la vera peculiarità dell'algoritmo ottimizzato RRT*, ovvero il *rewiring*. In parole semplici è letteralmente ciò che la parola significa ovvero un ricablaggio dell'albero a mano a mano che vengono aggiunti nuovi vertici. È un processo molto simile agli algoritmi di risoluzione dei percorsi visti negli algoritmi del sotto capitolo precedente. Inizialmente come già detto si passa per quello che è l'appena definito algoritmo RRG collegando il nuovo nodo ottenuto a tutti i suoi vicini. Prima di passare oltre però viene fatto il rewiring vero e proprio. Ovvero molto semplicemente il nuovo nodo q_{new} viene ricollegato in modo non da minimizzare il costo del segmento locale che lo unisce al grafo, ma bensì da minimizzare quello che è il costo del percorso totale (partendo da q_{start}). Non si può ancora considerare concluso qui il rewiring, tuttavia, la stessa operazione appena descritta viene infatti effettuata anche per tutti gli altri nodi limitrofi appartenenti all'insieme X_{near} considerando però anche il nodo q_{new} che ovviamente non c'era nelle iterazioni precedenti. Il processo appena descritto garantisce pertanto le comodità computazionali di un grafo ad albero e la garanzia di avere una soluzione asintoticamente ottima con il procedere dell'algoritmo.

Complessivamente è quindi possibile riassumere l'intero algoritmo RRT* nei seguenti punti[2]:

1. Creazione del grafo $G(V,E)$ come insieme vuoto ed inserimento del nodo di partenza q_{start} dallo spazio delle configurazioni C_{free} in V.
2. Generazione di un nodo q_{rand} proveniente anch'esso da C_{free}.
3. Identificazione del nodo $q_{nearest}$ più vicino a q_{rand}.
4. Calcolo del nuovo nodo q_{new} tramite *Steering function.*
5. Validazione del percorso tra $q_{nearest}$ e q_{new} in modo che non produca collisioni.
6. Calcolo dell'insieme X_{near} contenente tutti i nodi entro una distanza r da q_{new}.
7. Annessione q_{new} a V.
8. Inizializzazione variabili ausiliarie utili per la fase di *Rewiring:*
    - q_{min}: Inizialmente posta 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 dell'intero percorso $q_{start} \rightarrow q_{min} \rightarrow q_{new.}$
9. Inizio della fase di *Rewiring.* L'algoritmo cerca all'interno di X_{near} il nodo che, se considerato come 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, ad

ogni iterazione l'algoritmo ottimizza i costi delle connessioni valutando se nelle vicinanze di q_{new} vi sia un possibile nodo precedente migliore di $q_{nearest}$. L'aggiornamento ovviamente avviene solo in caso la traiettoria non vada a creare collisioni

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} nell'insieme X_{near} possano essere raggiunti con un costo minore considerando q_{new} come nodo precedente.

Per capirne ancora meglio il funzionamento di seguito sono riportati l'algoritmo in forma di pseudo codice e alcune immagini di come viene effettuato il rewiring[8].

---

1 $V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
2 **for** $i = 1, \ldots, n$ **do**
3 $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
4 $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
5 $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
6 **if** $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
7 $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
8 $V \leftarrow V \cup \{x_{\text{new}}\}$;
9 $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
10 **foreach** $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
11 **if** $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
12 $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
13 $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
14 **foreach** $x_{\text{near}} \in X_{\text{near}}$ **do** // Rewire the tree
15 **if** $\texttt{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}})) < \texttt{Cost}(x_{\text{near}})$ **then** $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
16 $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
17 **return** $G = (V, E)$;

*Figura 27: Pseudo codice algoritmo RRT**

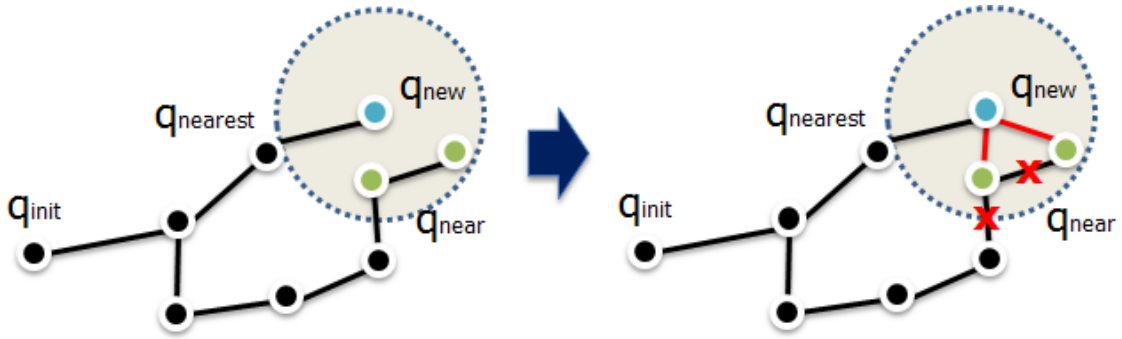


Figura 28: Esempio rewiring

2.3.5 Algoritmo Kinodynamic RRT*

Ultimo algoritmo del capitolo e finora probabilmente il più completo. Sicuramente l'algoritmo più vicino a quella che è la realtà di un sistema dinamico. Proposto per la prima volta nell'articolo scientifico *"Kinodynamic RRT*: Optimal Motion Planning for Systems with Linear Differential Constraints"[9]* di D. J. Webb e J. van den Berg si tratta di un adattamento del precedentemente citato algoritmo RRT* ad applicazioni esprimibili come sistemi dinamici lineari o linearizzabili.

Per sistemi dinamici lineari intendiamo tutti i sistemi che possono essere definiti come:

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

Dove $x[t] \in X$ è lo stato del robot, $u[t] \in U$ sono i controlli del robot e $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$ e $c \in \mathbb{R}^n$ sono delle costanti note del sistema.

L'algoritmo differisce dalla versione RRT* per un solo, ma fondamentale punto, ovvero la steering function. Se fino ad ora, infatti, la steering function ipotizzava traiettorie assimilabili a dei segmenti adesso non è più così. La condizione differenziale alla giunzione di due stati non può infatti essere soddisfatta da dei semplici segmenti che si muovono a zig-zag. È anche vero però che il problema non risulta affatto banale da risolvere dal momento che per quanto siano molto efficienti anche le versioni ottimizzate degli algoritmi PRM ed RRT tenderebbero comunque ad aumentare il costo computazionale in maniera esponenziale con la complessità dei sistemi rendendo di fatto un approccio troppo improntato alla forza bruta poco efficacie. L'approccio utilizzato nell'articolo è stato quello di attribuire ad ogni percorso π fisicamente percorribile una funzione di costo che tenga conto del controllo utilizzato e del tempo impiegato per raggiungere l'obbiettivo. La funzione di costo è quindi esprimibile nella forma:

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

Dove $R \in \mathbb{R}^{m\times m}$ è una matrice definita positiva con delle costanti che attribuiscono il costo ad ogni input del sistema di controllo e τ è la durata dell'intero percorso.

Per la soluzione dimostrata in maniera approfondita si invita a far riferimento all'articolo vero e proprio nella bibliografia[9]. I risultati ottenuti nell'articolo risultano eccellenti e infatti ad oggi la soluzione di controllo proposta rimane al giorno d'oggi forse la più popolare grazie all'efficienza ed alla robustezza che la caratterizzano riuscendo a coniugare una risposta completa e precisa della dinamica reale ad una velocità ed

efficienza tipiche della sfera più prettamente matematica e astratta degli algoritmi presentati. Di seguito è riportato lo pseudo codice dell'algoritmo:

KINODYNAMICRRT*$[\mathbf{x}_{\text{start}} \in \mathcal{X}_{\text{free}}, \mathbf{x}_{\text{goal}} \in \mathcal{X}_{\text{free}}]$

1: $\mathcal{T} \leftarrow \{\mathbf{x}_{\text{start}}\}$.
2: **for** $i \in [1, \infty)$ **do**
3: Randomly sample $\mathbf{x}_i \in \mathcal{X}_{\text{free}}$.
4: $\mathbf{x} \leftarrow \text{argmin}\{\mathbf{x} \in \mathcal{T} \mid c^*[\mathbf{x}, \mathbf{x}_i] < r \wedge$ COLLISIONFREE$[\pi^*[\mathbf{x}, \mathbf{x}_i]]\}$ $(\text{cost}[\mathbf{x}] + c^*[\mathbf{x}, \mathbf{x}_i])$.
5: $\text{parent}[\mathbf{x}_i] \leftarrow \mathbf{x}$.
6: $\text{cost}[\mathbf{x}_i] \leftarrow \text{cost}[\mathbf{x}] + c^*[\mathbf{x}, \mathbf{x}_i]$.
7: **for all** $\{\mathbf{x} \in \mathcal{T} \cup \{\mathbf{x}_{\text{goal}}\} \mid c^*[\mathbf{x}_i, \mathbf{x}] < r \wedge \text{cost}[\mathbf{x}_i] + c^*[\mathbf{x}_i, \mathbf{x}] < \text{cost}[\mathbf{x}] \wedge$ COLLISIONFREE$[\pi^*[\mathbf{x}_i, \mathbf{x}]]\}$ **do**
8: $\text{cost}[\mathbf{x}] \leftarrow \text{cost}[\mathbf{x}_i] + c^*[\mathbf{x}_i, \mathbf{x}]$.
9: $\text{parent}[\mathbf{x}] \leftarrow \mathbf{x}_i$.
10: $\mathcal{T} \leftarrow \mathcal{T} \cup \{\mathbf{x}_i\}$.

*Figura 29: Pseudo codice algoritmo Kinodynamic RRT**

Capitolo 3

In questo capitolo verranno affrontati quelli che sono alcuni dei più popolari algoritmi di obstacle-avoidance in uso oggi. L'importanza di questa particolare categoria di algoritmi non è affatto da sottovalutare in quelle che sono le applicazioni moderne. Se da un lato infatti è fondamentale avere una buona strategia di path-planning a monte per tenere conto di quelli che sono gli ostacoli certi sul percorso altrettanto importante è avere un'efficacie strategia della gestione di quelli che sono gli ostacoli imprevisti.

Quasi tutti gli algoritmi che verranno presentati hanno una natura locale nell'affrontare il problema principalmente per due motivi: l'obstacle-avoidance è un problema locale in quanto legato ad un imprevisto che può avvenire in una zona ristretta e poi essendoci per ovvi motivi una forte correlazione con quella che è la sensoristica a bordo non è possibile espandere troppo le informazioni da dare al solutore.

Per sua natura il problema è qualcosa che va gestito parallelamente a quelle che sono le strategie di path-following, ma in questo capitolo ci si limiterà a vedere la cosa da un punto di vista più algoritmico senza scendere eccessivamente in quelli che potrebbero essere i dettagli dinamici dell'interruzione del percorso originale, in quanto sarebbe un argomento fin troppo legato alla singola applicazione e quindi rischierebbe di appesantire inutilmente la soluzione di carattere più generale.

In maniera più formale il problema di obstacle-avoidance può essere espresso nel seguente modo[10]. Sia A il robot che si muove nello spazio W e sia il suo spazio delle configurazioni C. Sia q una configurazione, q_t la configurazione al tempo t e $A(q_t) \in W$ lo spazio occupato dal robot nella configurazione. Sul robot c'è un sensore che nella configurazione q_t misura una porzione di spazio $S(q_t) \subset W$ identificando un insieme di ostacoli $O(q_t) \subset W$.

Sia ora u un vettore di controllo costante e sia $u(q_t)$ il vettore in q_t al tempo δt. Dato $u(q_t)$ il robot descriverà una traiettoria

$$q_{t+\delta t} = f(u, q_t, \delta t)$$

Con $\delta t \geq 0$. Sia $Q_{t,T}$ l'insieme delle configurazioni della traiettoria seguita partendo da q_t con $\delta t \in [0, T]$, un generico intervallo temporale. $T > 0$ è chiamato *periodo di campionamento*. Sia $F : C$ x C una funzione che valuta la progressione da una configurazione ad un'altra. Date queste definizioni è ora possibile definire il problema di obstacle avoidance in maniera precisa.

Sia q_{goal} la configurazione di arrivo. Allora, al tempo t_i il robot è nella configurazione q_{ti}, dove il sensore di misura ottiene lo spazio $S(q_{ti})$ dove sono presenti ostacoli $O(q_{ti})$. L'obbiettivo è dare un segnale di controllo u_i tale che:

I. La traiettoria generata sia priva di collisioni con degli ostacoli

$$A(Q_{ti,T}) \cap O(q_{ti}) = \emptyset$$

II. Faccia progredire il robot verso la configurazione di arrivo

$$F(q_{ti}, q_{goal}) < F(q_{ti+T}, q_{goal})$$

Da notare come l'utilizzo di sensori che campionano costantemente l'ambiente circostante al robot aiutino soluzioni intrinsecamente locali (Figura 30.a) e che quindi rispettano il primo punto a risolvere anche il secondo di natura globale (Figura 30.b). Il campionamento costante unito alla richiesta che ogni intervallo deve essere privo di collisioni dà quindi come risultato la soluzione globale del problema. Di seguito verranno descritti alcuni dei principali algoritmi utilizzati per risolvere il problema.

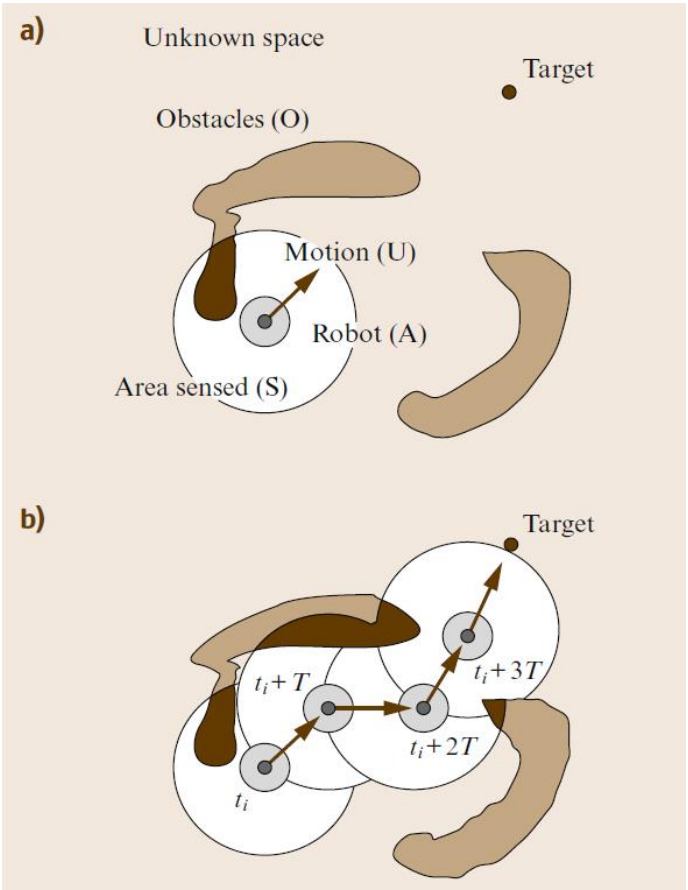


Figura 30: Generico esempio di ostacle-avoidance

3.1 Bug Algorithms

Sicuramente uno dei metodi di obstacle avoidance più semplici, ma in grado comunque di dare alcune garanzie. Gli algoritmi in questa categoria ipotizzano il robot come un punto immerso nell'ambiente e, almeno le prime due versioni, si basano sull'assunzione che i sensori siano a raggio nullo; ovvero si attivino a contatto similmente ad un pulsante. È possibile, tuttavia, anche una versione in cui il sensore abbia una percezione in profondità dell'ambiente rilevando eventuali ostacoli prima di arrivare toccarli. Il loro funzionamento in linea generale può essere ricondotto a due tipi di comportamento:

- **Motion-to-goal:** In maniera abbastanza intuitiva si limitano a seguire il percorso indicato da un pianificatore di grado più alto oppure ancor più semplicemente puntano in linea retta al punto di arrivo, assumendo ovviamente che sia noto.
- **Follow a boundary:** La fase di obstacle avoidance vera e propria. Consiste, con le dovute differenze di policy in base allo specifico algoritmo, nel seguire il contorno dell'ostacolo rilevato per trovare una soluzione di percorso e tornare alla fase precedente.

Com'è facilmente intuibile nonostante una forte semplicità di concetto e di implementazione questi algoritmi presentano difetti non trascurabili in applicazioni realistiche in quanto non tengono conto della reale cinematica del robot e in generale risultano poco robusti e molto onerosi a livello computazionale. Nelle seguenti sezioni verranno esplorate quelle che sono i tre principali algoritmi di questa categoria.

3.1.1 Algoritmo Bug 1

Forse la versione più primitiva in assoluto di quello che può essere un algoritmo di obstacle avoidance. Come già accennato si basa anch'esso su due assunti molto semplici; ovvero il robot di natura puntiforme e la sensoristica di rilevamento ostacoli funzionante per contatto.
In questo algoritmo il robot si limita a procedere in maniera diretta in direzione del punto di arrivo e ogni volta che incontra un ostacolo gli gira intorno per poi ripartire in direzione dell'arrivo dal punto più vicino ad esso sul perimetro dell'ostacolo. La procedura viene ripetuta ogni volta che viene rilevato un nuovo ostacolo finché non viene trovato un

percorso o un'impossibilità di soluzione. Più formalmente può essere espresso nel seguente modo[5]. Come già detto, sia il robot puntiforme e il sensore rilevi l'ostacolo tramite contatto. Si immagini poi che il robot possa misurare la distanza *d(x,y)* tra un punto *x* ed un punto *y*. Si assuma infine che lo spazio di lavoro sia *vincolato.* Formalmente uno spazio si definisce vincolato nel seguente modo. Sia $B_r(x)$ una palla di raggio *r* centrata in *x;* $B_r(x) = \{y \in \mathbb{R}^2 | d(x,y) < r\}$. Il fatto che lo spazio di lavoro sia vincolato implica che per ogni $x \in W$ esiste almeno un raggio $r < \infty$ tale per cui $W \subset B_r(x)$. Detto più semplicemente quest'ultima condizione impone che il problema sia finito, ovvero che si possa trovare una soluzione o un'impossibilità di essa in un tempo finito.

Definite le basi è possibile descrivere meglio l'algoritmo vero e proprio. La partenza e l'arrivo sono nominati rispettivamente q_{start} e q_{goal}. Sia ora $q_0^L = q_{start}$ il primo punto di "distacco" e sia la linea *m* il segmento che unisce il punto q_i^L al punto q_{goal} con $i = 0$ inizialmente. A questo punto l'algoritmo inizia la fase di path following o motion to goal dove si dirige verso il punto q_{goal} seguendo la linea *m* finché non lo raggiunge oppure non si imbatte in un ostacolo. Se il robot ha effettivamente colpito un ostacolo, sia q_1^H il punto dove è avvenuto il primo contatto. A questo punto il robot circumnavigherà l'ostacolo finché non tornerà nel punto q_1^H. Campionato quindi il perimetro dell'ostacolo il robot determinerà su di esso il punto più vicino a q_{goal} e viaggerà fino ad esso. Questo sarà il nuovo punto di distacco q_1^L. Da qui riprenderà la fase di motion to goal verso l'arrivo. Se la nuova traiettoria viene nuovamente arrestata dallo stesso ostacolo vuol dire che non esiste soluzione, altrimenti il procedimento appena descritto verrà reiterato per ogni nuovo ostacolo finché non si otterrà una soluzione oppure un'impossibilità di quest'ultima.

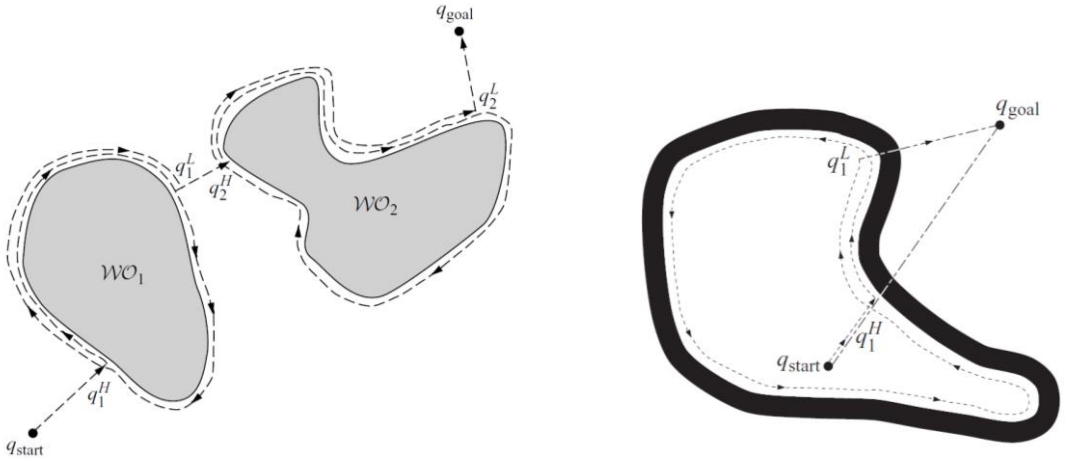


Figura 31: Esempi di algoritmo Bug 1

Per ulteriore chiarezza di seguito è riportato l'algoritmo in versione di pseudo codice.

Input: A point robot with a tactile sensor
Output: A path to the q_{goal} or a conclusion no such path exists

while Forever **do**
  repeat
    From q_{i-1}^{L}, move toward q_{goal}.
  until q_{goal} is reached **or** an obstacle is encountered at q_i^H.
  if Goal is reached **then**
    Exit.
  end if
  repeat
    Follow the obstacle boundary.
  until q_{goal} is reached **or** q_i^H is re-encountered.
  Determine the point q_i^L on the perimeter that has the shortest distance to the goal.
  Go to q_i^L.
  if the robot were to move toward the goal **then**
    Conclude q_{goal} is not reachable and exit.
  end if
end while

Figura 32: Pseudo codice algoritmo Bug 1

3.1.2 Algoritmo Bug 2

Versione alternativa dell'algoritmo Bug 1 proposto nel sotto capitolo precedente[5]. Si basa essenzialmente sulle stesse ipotesi e gli stessi principi di funzionamento. Anche in questo caso troviamo un comportamento di motion to goal e di obstacle avoidance vero e proprio e anche in questo caso si lavora con un robot puntiforme in uno spazio limitato in cui gli ostacoli vengono riconosciuti per contatto.

La vera differenza sta nelle linee *m* che di volta in volta connettono il punto di distacco q_i^L attuale al punto q_{goal}. Nell'algoritmo è infatti una sola e parte dal nodo q_{start} fino al nodo q_{goal}. A questo punto infatti il punto di distacco q_i^L da ogni ostacolo non sarà più determinato dalla circumnavigazione del suddetto e dalla minimizzazione della distanza rispetto all'arrivo, ma bensì dal primo punto appartenente alla linea *m* originale incontrato dopo aver iniziato a seguire il bordo dell'ostacolo.

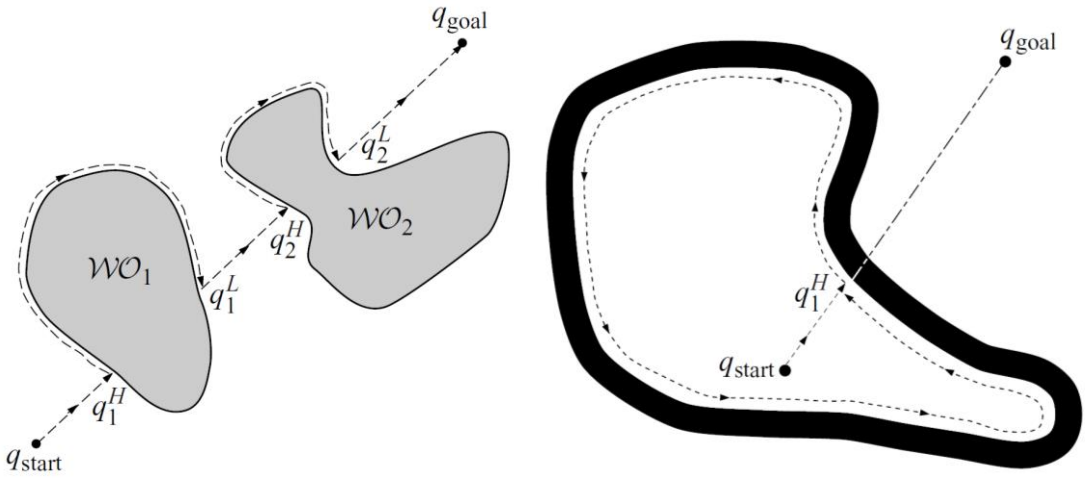


Figura 33: Esempio algoritmo Bug 2

Ad una prima analisi più superficiale può sembrare che l'algoritmo Bug 2 sia sempre e comunque più efficiente dell'algoritmo Bug 1 dal momento che a differenza di quest'ultimo non perderà tempo a circumnavigare tutti gli ostacoli incontrati nella loro interezza; tuttavia, la cosa non è da ritenersi vera a priori. Per capire meglio il perché l'ultima affermazione sia falsa è opportuno analizzare più attentamente la lunghezza dei percorsi intrapresi da entrambi gli algoritmi a parità di percorso.

L'algoritmo Bug 1 raggiunto l'*i*-esimo ostacolo lo circumnavigherà completamente e poi ripartirà verso l'arrivo dopo aver raggiunto il punto di distacco stabilito. Nel peggiore dei casi per ritornare al punto di distacco scelto dovrà percorrere al massimo metà del perimetro, p_i, dell'ostacolo incontrato. Sempre nel peggiore dei casi poi, incontrerà tutti gli n ostacoli presenti lungo il percorso. Se non ci sono ostacoli infine la lunghezza da percorrere sarà $d(q_{start}, q_{goal})$. Si può quindi affermare che la massima lunghezza percorsa sarà

$$L_{Bug1} \leq d(q_{start}, q_{goal}) + 1.5 \sum_{i=1}^{n} p_i$$

Per l'algoritmo Bug 2 la lunghezza del tragitto è un po' più complessa. Si supponga che la linea che va da q_{start} a q_{goal} incroci l'*i*-esimo ostacolo n_i volte. Ci saranno quindi al massimo n_i punti di distacco lungo l'ostacolo. Risulta immediato capire, inoltre, che metà di questi punti non saranno utilizzabili dal momento che si trovano sul lato "sbagliato" dell'ostacolo e proseguire verso q_{goal} partendo da essi porterebbe ad una collisione. Nel peggiore dei casi è quindi possibile che per ogni punto il robot attraversi la quasi interezza del perimetro. La lunghezza finale massima sarà quindi

$$L_{Bug2} \leq d(q_{start}, q_{goal}) + \frac{1}{2}\sum_{i=1}^{n} n_i p_i$$

Già da un'analisi superficiale si può intuire che è possibile avere un percorso più lungo in un algoritmo Bug 2 rispetto ad un algoritmo Bug 1. La discriminante perché ciò avvenga è la complicatezza degli ostacoli presenti. Se infatti l'algoritmo Bug 2 risulta chiaramente superiore in presenza di ostacoli semplici perde drammaticamente di efficacia in presenza di ostacoli di tipo labirintico.

La cosa è spiegabile da quelle che sono le policy dei due algoritmi. Se infatti l'algoritmo Bug 1 può essere visto come un algoritmo più "esaustivo", dal momento che per ogni ostacolo incontrato si assicurerà di trovare sempre il percorso migliore, l'algoritmo Bug 2, è invece, un algoritmo di tipo *greedy.* Come suggerisce abbastanza bene il termine; infatti, anzi che analizzare l'ostacolo nella sua interezza si limiterà ad intraprendere la prima strada che offra un miglioramento rispetto al percorso attuale, indipendentemente che questa rappresenti la scelta migliore in senso assoluto. Un paio di esempi dei limiti di tale strategia sono ben illustrato nelle figure sottostanti.

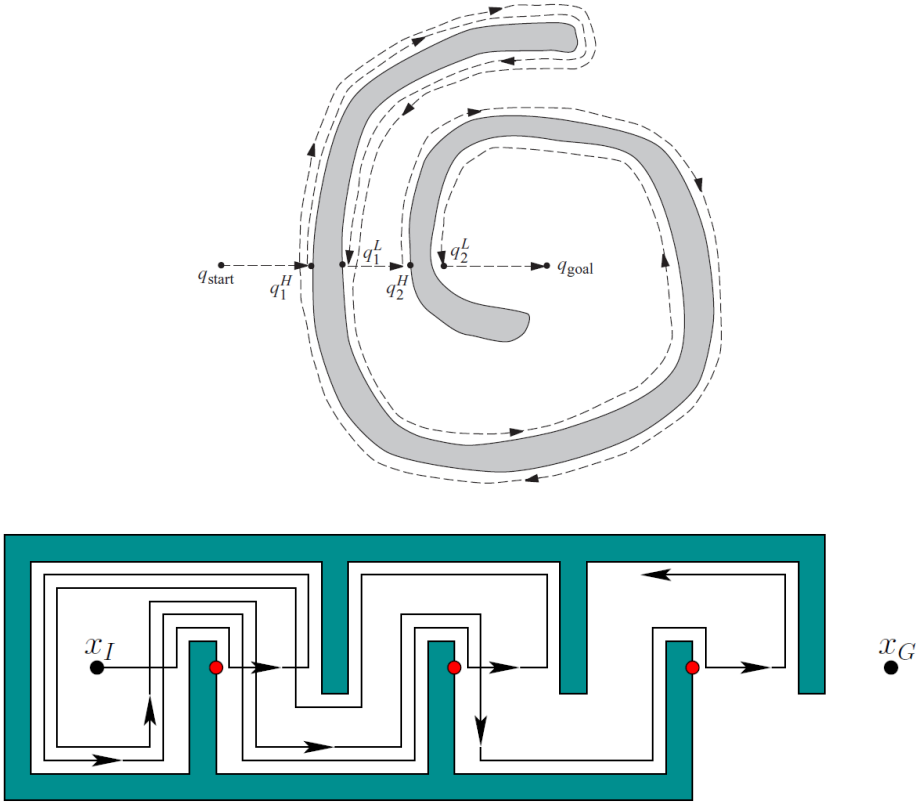


Figura 34: Esempi di pessimo uso dell'algoritmo Bug 2

Per ulteriore chiarezza di seguito è riportato l'algoritmo in forma di pseudo codice.

Input: A point robot with a tactile sensor
Output: A path to q_{goal} or a conclusion no such path exists

while True **do**
  repeat
    From q_{i-1}^{L}, move toward q_{goal} along m-line.
  until
  q_{goal} is reached **or**
  an obstacle is encountered at *hit point* q_i^H.
  Turn left (or right).
  repeat
    Follow boundary
  until
   q_{goal} is reached **or**
    q_i^H is re-encountered **or**
    m-line is re-encountered at a point m such that
      $m \neq q_i^H$ (robot did not reach the hit point),
      $d(m, q_{goal}) < d(m, q_i^H)$ (robot is closer), and
      If robot moves toward goal, it would not hit the obstacle
  Let $q_{i+1}^{L} = m$
  Increment i
end while

Figura 35: Pseudo codice algoritmo Bug 2

3.1.3 Algoritmo Tangent Bug

Può essere considerato come un miglioramento dell'algoritmo Bug 2. A differenza degli algoritmi precedenti[5] si basa su un tipo di sensoristica in grado di rilevare la presenza di ostacoli a distanza invece che per contatto. Da questo cambiamento è quindi possibile estrapolare un algoritmo che rispetto alle precedenti versioni permette un approccio più verosimile.

La versione più generale prevede l'utilizzo di una sensoristica in grado di rilevare la presenza di ostacoli a 360 gradi attorno al robot per una distanza infinita. Questo sensore viene modellato dalla funzione $\rho : \mathbb{R}^2 \times S^1 \longrightarrow \mathbb{R}$. Si consideri un robot puntiforme situato nel punto $x \in \mathbb{R}^2$ che emani dei raggi di lunghezza infinita in ogni direzione intorno ad esso. Per ogni $\theta \in S^1$ il valore $\rho(x, \theta)$ è la distanza minima tra x e l'ostacolo più vicino lungo la direzione θ. Più formalmente

$$\rho(x,\theta) = \min_{\lambda \in [0,\infty]} d(x, x + \lambda[\cos\theta, \sin\theta]^T), tale\ che\ x + \lambda[\cos\theta, \sin\theta]^T \in \bigcup_i WO_i$$

La definizione appena presentata si basa su una risoluzione di θ infinita e perciò irrealizzabile. È comunque possibile accettare un'approssimazione ottenuta da diversi

sensori lungo la circonferenza del robot.

Similmente a quanto detto per la direzione non è possibile neanche avere sensori in grado di leggere a distanze infinite. Viene perciò adottata la funzione *satura* $\rho_R: \mathbb{R}^2 \times S^1 \to \mathbb{R}$. Molto semplicemente questa nuova funzione assume il valore di ρ entro il valore della distanza di range, R, mentre va a infinito se l'ostacolo si trova al di fuori di questo intervallo. Più formalmente

$$\rho_R = \begin{cases} \rho(x,\theta), & \rho(x,\theta) < R \\ \infty \quad , & \rho(x,\theta) \geq R \end{cases}$$

Altra assunzione fondamentale fatta dal pianificatore è la capacità del robot di rilevare le discontinuità di ρ_R. Per una posizione $x \in \mathbb{R}^2$ fissata, un *intervallo di continuità* è definito come un insieme di punti $x + \rho(x,\theta)[\cos\theta, \sin\theta]^T$ connesso lungo il confine dello spazio libero dove $\rho_R(x,\theta)$ è continua è finita rispetto a θ.

I punti di transizione di questi intervalli si trovano quando $\rho_R(x,\theta)$ diventa discontinua. Le discontinuità possono essere generate sia da un ostacolo che ne blocca un altro sia dal sensore che raggiunge il range limite. Questi punti vengono chiamati O_i. Nelle seguenti figure viene illustrato più chiaramente il concetto mostrando come effettivamente il robot riesce a vedere l'ambiente circostante.

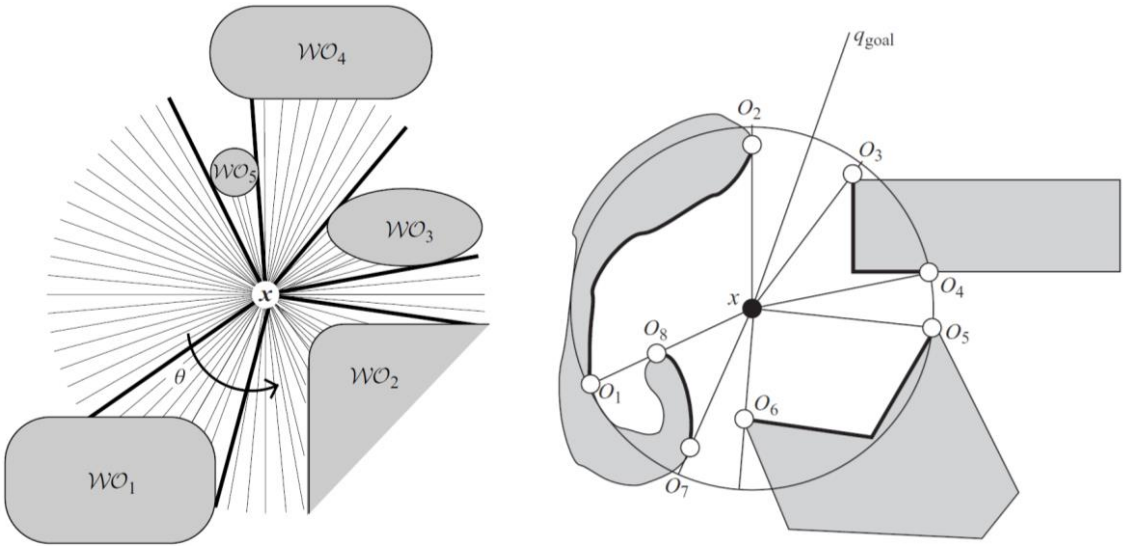


Figura 36: Esempio di rilevamento delle discontinuità nel Tangent Bug

Dalle immagini risulta chiaro quanto detto prima. Gli intervalli in grassetto rappresentano gli intervalli di continuità e i punti O_i rappresentano ogni volta l'inizio e la fine di dati intervalli. Da notare, come già detto, che l'interruzione di continuità oltre che per la presenza di altri ostacoli può avvenire anche con il raggiungimento del limite del sensore (figura a destra, punto O_4).

Come nelle versioni precedenti è possibile notare anche nell'algoritmo Tangent Bug due diversi tipi di comportamento, e anche in questo caso si tratta del motion to goal e del boundary following. Anche se si tratta delle stesse fasi già viste nelle prime due versioni dell'algoritmo in realtà entrambe queste fasi presentano un tipo di comportamento ibrido tra di loro.

Inizialmente come già visto negli altri algoritmi Bug il robot comincia con un tipo di comportamento motion-to-goal che, come verrà discusso a breve, può essere a sua volta diviso in due parti. Dapprima il robot si dirige in maniera diretta verso l'arrivo come sempre fatto e questo finché non rileva la presenza di un ostacolo a distanza *R* che si trovi direttamente sul suo percorso. Questo significa semplicemente che la linea che porta il robot all'arrivo cade in un intervallo di continuità. Ciò viene rappresentato molto bene dalla seguente figura dove si può notare che WO_2 anche se percepito non blocca il passaggio mentre l'ostacolo WO_1 si. Durante la prima lettura il cerchio di raggio *R* entro cui percepisce l'ambiente il robot diviene tangente all' ostacolo in un punto e successivamente con il procedere del movimento il punto si divide in quelli che saranno di volta in volta i punti O_i che determinano l'intervallo di continuità. Se la linea congiungente tra punto di partenza e punto di arrivo cade all'interno dell'intervallo l'ostacolo è considerato davanti al robot.

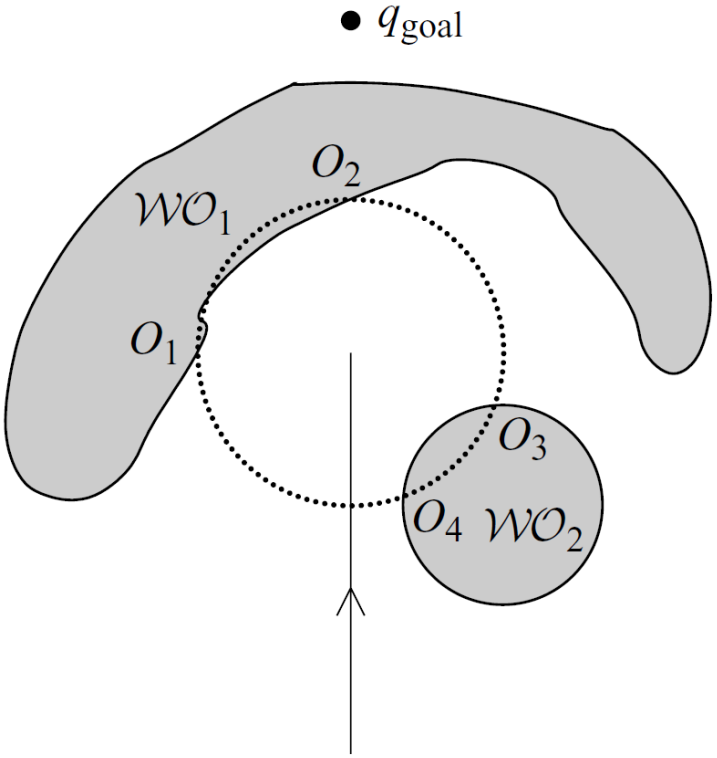


Figura 37: Esempio di rilevamento ostacoli sulla traiettoria da percorrere

Una volta rilevato l'ostacolo il robot si muove verso il punto O_i che ne minimizzi la distanza euristica $d(x, O_i) + d(O_i, q_{goal})$. Come già detto più volte nel documento la funzione euristica può variare molto in base all'applicazione ed è strettamente legata alle informazioni che si hanno ed alle grandezze che si vogliono tenere sotto controllo. Nelle figure seguenti è possibile capire meglio questo concetto. Nell'immagine a sinistra il robot vede WO_1 e si dirige verso O_2 perché $i = 2$ è il punto che minimizza la distanza $d(x, O_i) + d(O_i, q_{goal})$. È interessante notare come nel punto x il robot ancora non sappia che l'ostacolo WO_2 effettivamente blocchi il passaggio. Nell'immagine a destra invece il robot ha informazioni sufficienti per determinare che WO_2 blocca il passaggio e quindi decide di dirigersi verso O_4.

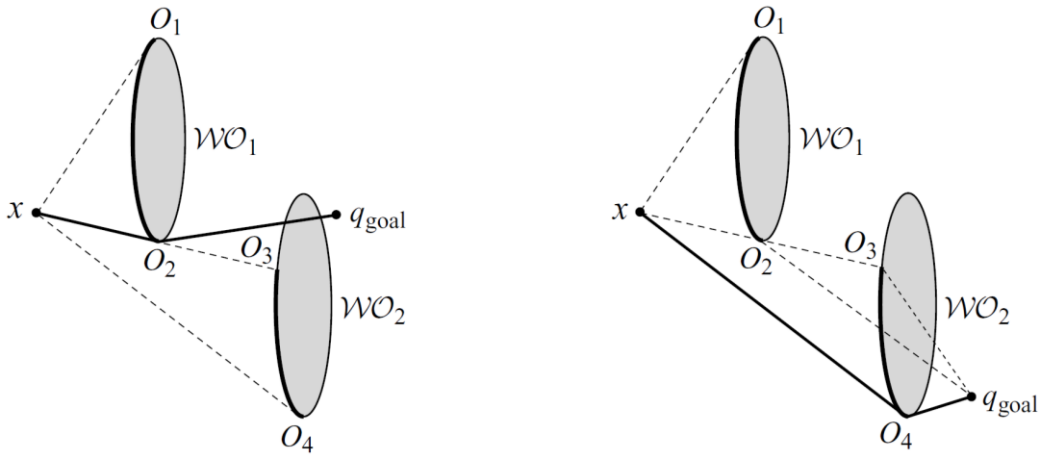


Figura 38: Esempio di deviazione della traiettoria basato sull'euristica

L'insieme $\{O_i\}$ viene aggiornato costantemente a mano a mano che il robot procede verso un particolare O_i. La seguente figura illustra chiaramente questo processo durante l'avvicinamento dell'ostacolo.

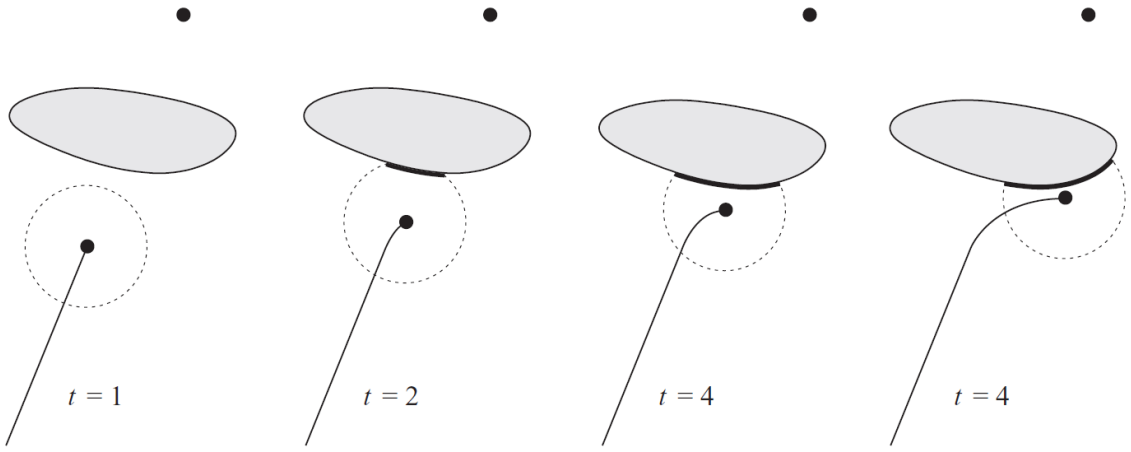


Figura 39: Dimostrazione dell'aggiornamento dell'euristica

In maniera simile a quanto visto nelle varianti precedenti questo tipo di movimento continua finché non è più possibile minimizzare il valore della funzione euristica. In altri termini la strategia continua finché la funzione $d(\cdot, O_i) + d(O_i, q_{goal})$ non si trova ad un minimo locale dettato dall'attuale percorso verso l'arrivo. A questo punto l'algoritmo passa in modalità boundary following. Per prima cosa trova nella porzione di ostacolo percepita il punto *M* che minimizza in senso assoluto la distanza dal punto di arrivo. La parte percepita dell'ostacolo è anche detta *ostacolo seguito*. A questo punto, però, è necessario definire quella che è la differenza tra l'ostacolo seguito e l'*ostacolo bloccante*. Sia *x* l'attuale posizione del robot. L'ostacolo bloccante è il più vicino ostacolo nel raggio del sensore che interseca la retta $(1-\lambda)x + \lambda q_{goal}\ \forall\lambda \in [0,1]$. Inizialmente l'ostacolo bloccante e l'ostacolo seguito sono la stessa entità.

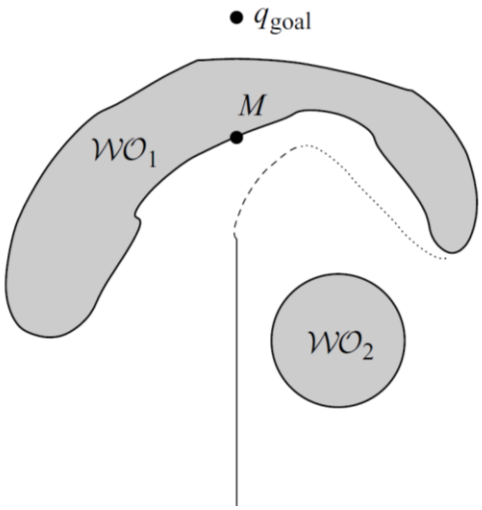


Figura 40: Esempio di boundary following

A questo punto il robot continua a muoversi nella stessa direzione di prima come se fosse ancora in modalità motion to goal. Continua a muoversi verso il punto O_i selezionato seguendo la direzione intrapresa prima di passare al boundary following. Contemporaneamente tiene però conto di due distanze: $d_{followed}$ e d_{reach}. La distanza $d_{followed}$ è la minima distanza tra il confine dell'ostacolo finora percepito e il punto di arrivo. La distanza d_{reach} è la distanza tra il punto percepito sull'ostacolo più vicino all'arrivo e l'arrivo stesso. Più formalmente; sia Λ l'insieme di tutti i punti sull'ostacolo WO_f visibili dalla posizione *x* nel range di raggio *R*, ovvero

$$\Lambda = \{y \in \partial WO_f : \lambda x + (1-\lambda)y \in Q_{free}\ \forall\lambda \in [0,1]\}$$

A questo punto d_{reach} sarà quindi

$$d_{reach} = \min_{c\in\Lambda} d(q_{goal}, c)$$

Quando si avrà $d_{reach} < d_{following}$ l'algoritmo riprenderà un comportamento di tipo motion to goal. Come nelle precedenti versioni le due modalità di movimento continueranno ad alternarsi finché non verrà raggiunto q_{goal} o finché non si avrà impossibilità di soluzione.

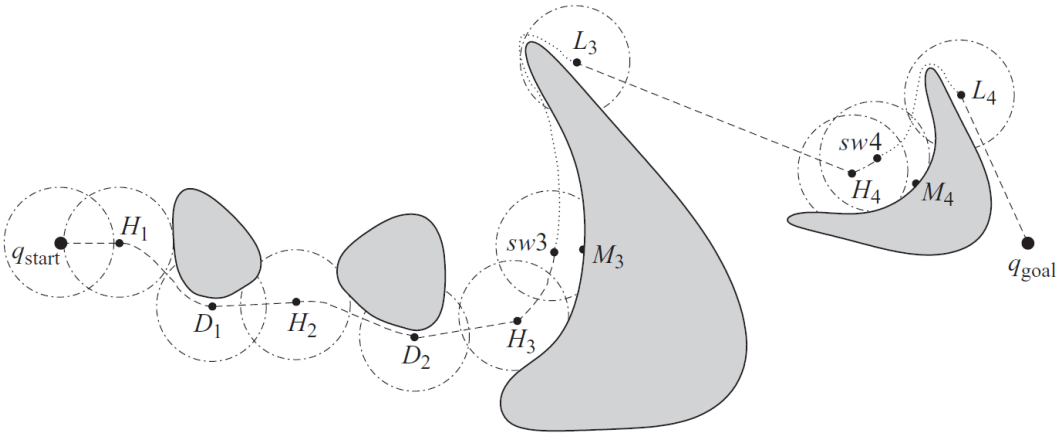


Figura 41: Esempio completo Tangent Bug Algorithm

Per una migliore comprensione di seguito è riportato lo pseudo codice dell'algoritmo Tangent Bug

Input: A point robot with a range sensor
Output: A path to the q_{goal} or a conclusion no such path exists

1: **while** True **do**
2: **repeat**
3: Continuously move toward the point $n \in \{T, O_i\}$ which minimizes $d(x, n) + d(n, q_{goal})$
4: **until**
- the goal is encountered **or**
- The direction that minimizes $d(x, n) + d(n, q_{goal})$ begins to increase $d(x, q_{goal})$, i.e., the robot detects a "local minimum" of $d(\cdot, q_{goal})$.

5: Chose a boundary following direction which continues in the same direction as the most recent motion-to-goal direction.
6: **repeat**
7: Continuously update d_{reach}, $d_{followed}$, and $\{O_i\}$.
8: Continuously moves toward $n \in \{O_i\}$ that is in the chosen boundary direction.
9: **until**
- The goal is reached.
- The robot completes a cycle around the obstacle in which case the goal cannot be achieved.
- $d_{reach} < d_{followed}$

10: **end while**

Figura 42: Pseudo codice algoritmo Tangent Bug

3.2 Follow the gap method (FGM)

Altro algoritmo molto semplice. Proposto per la prima volta nell'articolo scientifico *"A novel obstacle avoidance algorithm: "Follow the Gap Method""*[11] di V. Sezer e M. Gokasan.

Come suggerisce abbastanza intuitivamente il nome si basa sulla scelta del passaggio più ampio per la prosecuzione della traiettoria. L'algoritmo è di natura locale e reattiva dal momento che, come buona parte degli algoritmi di obstacle avoidance, subentra al pianificatore locale solo in caso di ostacoli improvvisi. Le ipotesi da cui l'algoritmo è stato sviluppato sono le seguenti:

1. **Campo visivo del robot:** Il campo visivo del robot può essere racchiuso in due raggi con angolo destro ϕ_{fov_r}, angolo sinistro ϕ_{fov_l} e una distanza dal bordo dell'ostacolo d_{fov}. Il robot non ha informazioni precedenti riguardo agli ostacoli.
2. **Vincoli del robot:** Il robot si muove obbedendo a dei vincoli di tipo non olonomo e la cosa può essere semplificata da un raggio minimo di sterza ta di quest'ultimo r_{min}.
3. **Coordinate:** Tutte le coordinate e i confini dei vari oggetti sono misurabili e i valori dei vincoli ϕ_{fov_i}, ϕ_{fov_r}, d_{fov} e r_{min} sono calcolati prima di procedere con l'algoritmo stesso in accordo con la posizione dei sensori e la geometria del robot.
4. **Approccio puntiforme:** Si suppone che la geometria del robot e degli ostacoli possa essere schematizzata da dei cerchi e per semplicità di programmazione si aggiunge il raggio del robot a quello degli ostacoli in modo da poter trattare il robot come un oggetto puntiforme.

La natura puntiforme del robot e del seguente rilevamento delle distanze è ben rappresentata nelle seguenti figure.

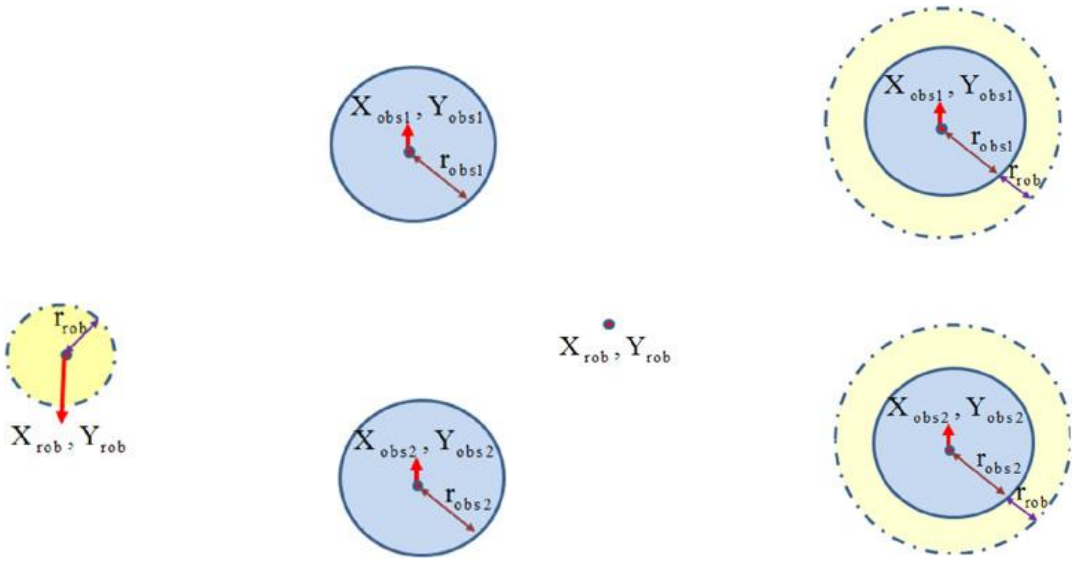


Figura 43: Rappresentazione approccio puntiforme al robot

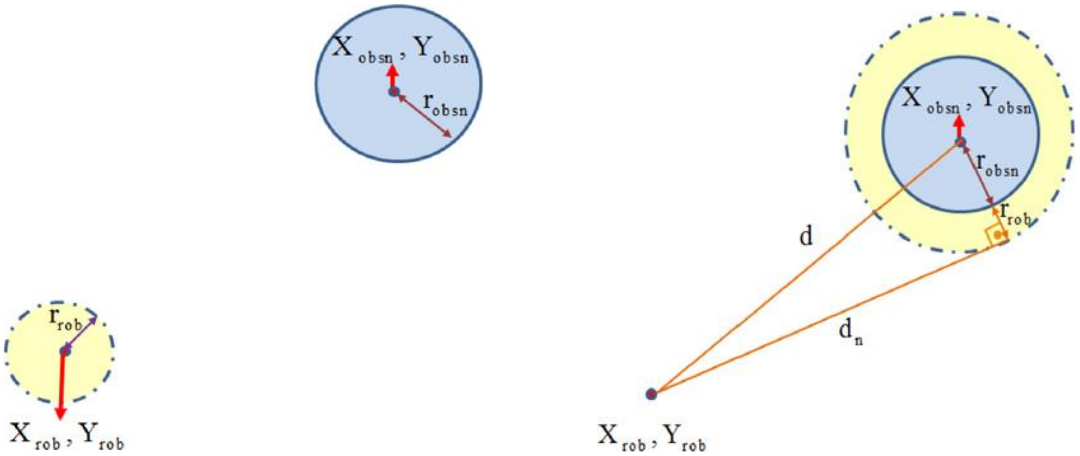


Figura 44: Calcolo delle distanze tra robot e ostacoli

Fatte le precedenti premesse a questo punto l'algoritmo vero e proprio può essere sintetizzato in tre fasi distinte. Da queste tre fasi lo scopo è ottenere l'angolo finale della traiettoria che intraprenderà il robot. Per fare ciò si tiene conto del più grande varco attraversabile (maximum gap) e del punto di goal da raggiungere indicato dal pianificatore globale. Le fasi sono indicate nel seguente schema a blocchi e vengono meglio illustrate nel resto del sotto capitolo.

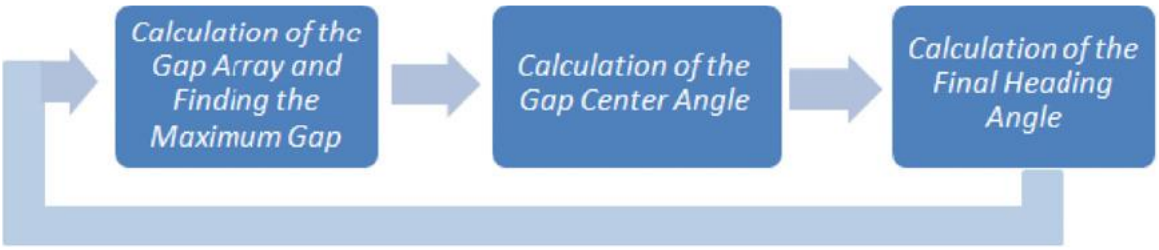


Figura 45: Fasi del Follow the Gap Method

3.2.1 Calcolo del vettore dei gap e ricerca gap massimo

Grazie all'ipotesi di natura puntiforme del robot ogni ostacolo nel suo intorno può essere definito da due parametri, ϕ_{obs_l} e ϕ_{obsi_r}, come mostrato nella figura seguente.

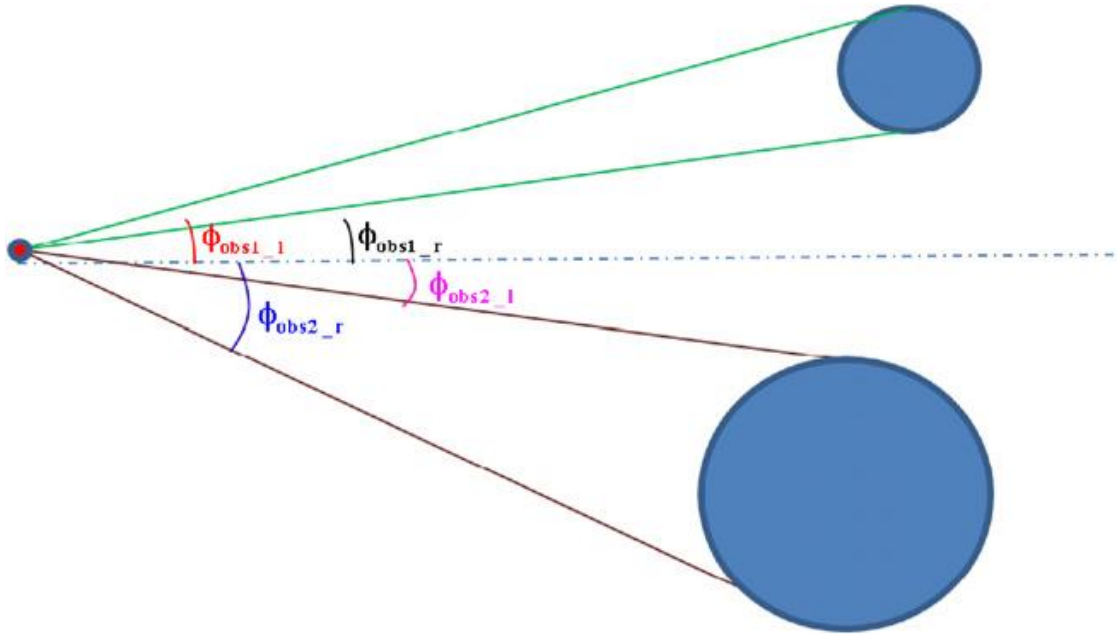


Figura 46: Rilevazione ostacoli da robot puntiforme

È necessario, tuttavia, ancora un passaggio prima di poter calcolare il vettore dei gap. Per poter procedere è infatti necessario tenere in conto della natura non olonoma dei vincoli in gioco in modo da definire correttamente lo spazio osservato e lo spazio raggiungibile. Detto in maniera semplificata il robot, essendo di tipo veicolistico, non può semplicemente spostarsi lateralmente, ma deve bensì muoversi necessariamente su degli archi con un raggio minimo r_{min}. Ricapitolando il robot può agire solo nello spazio circostante che riesce sia a vedere che a raggiungere. Più formalmente il criterio di vincolo è espresso come segue:

$$d_{nhol} < d_{fov} \Rightarrow \phi_{lim} = \phi_{nhol}$$
$$d_{nhol} \geq d_{fov} \Rightarrow \phi_{lim} = \phi_{fov}$$

Il risultato finale è rappresentato nelle seguenti figure.

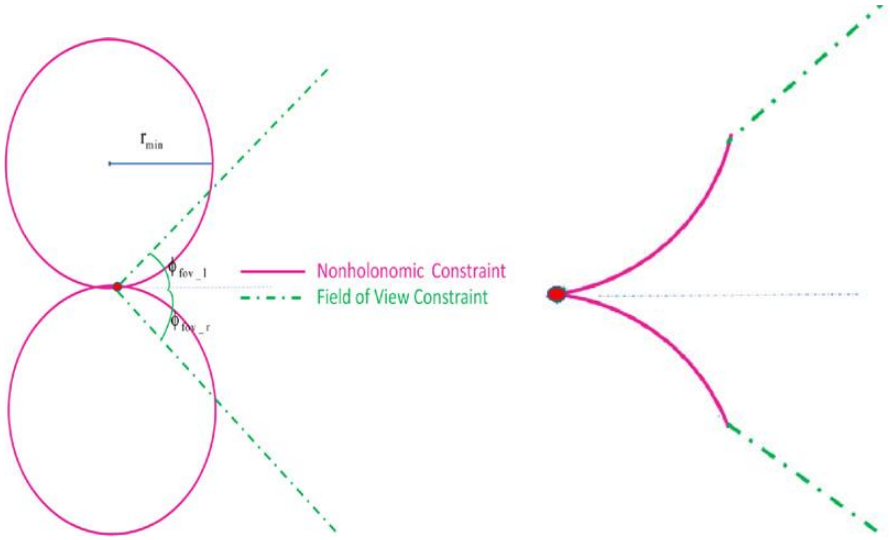


Figura 47: Dominio del robot. A sinistra l'incrocio tra vincoli olonomi e campo visivo. A destra dominio risultante

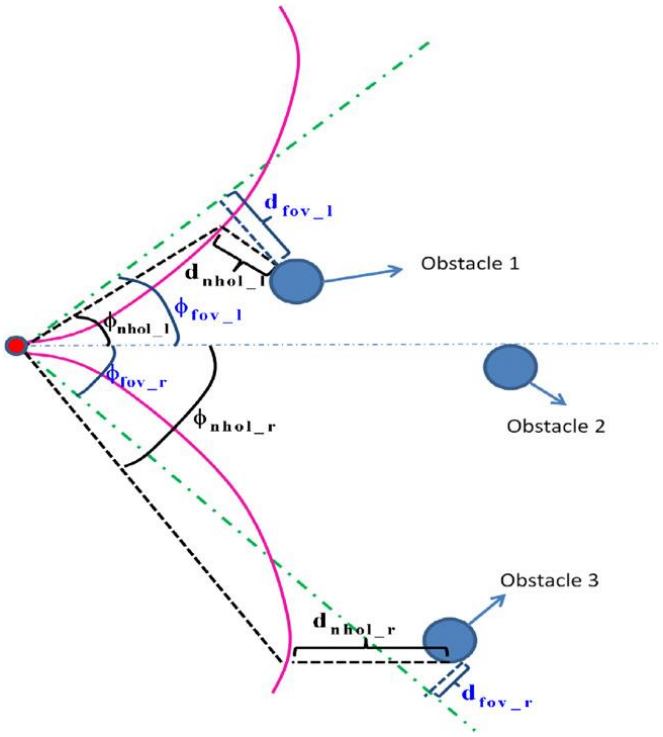


Figura 48: Parametri del dominio

Opportunamente definito il dominio con tutti i suoi parametri è ora possibile andare finalmente a costruire il vettore dei gap. I gap altro non sono che la differenza angolare tra la fine di un ostacolo e l'inizio del successivo. Più formalmente il vettore è espresso così

$$Gap[N+1] = \left[\left(\phi_{\text{lim }\_l} - \phi_{obs1\_l}\right)\left(\phi_{\text{obs1 }\_r} - \phi_{obs2\_l}\right)\cdots\left(\phi_{\text{obs(n-1) }\_r} - \phi_{obs(n)\_l}\right)\left(\phi_{\text{obs(n) }\_r} - \phi_{lim\_r}\right)\right]$$

Ottenuto il vettore dei gap l'algoritmo si limita a scegliere l'elemento maggiore.

3.2.2 Calcolo dell'angolo del centro del gap

Ottenuto il gap maggiore si passa al calcolo di quello che è l'angolo necessario al robot per raggiungerne il centro, ϕ_{gap_c}. Il significato geometrico di quest'affermazione è facilmente intuibile dall'immagine sottostante.

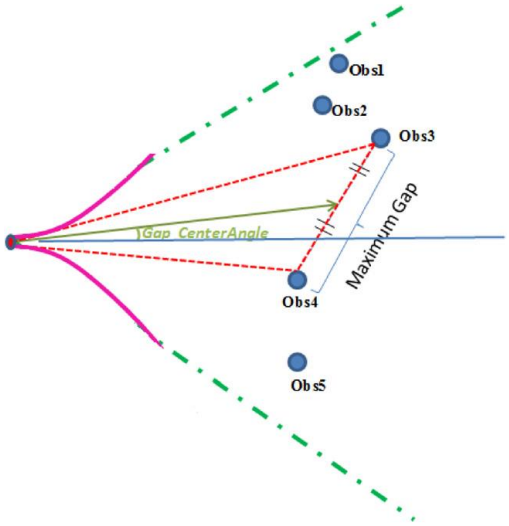


Figura 49: Angolo del gap centrale

Il calcolo di tale angolo è molto semplice e si può agevolmente ricavare utilizzando i temi del coseno e di Apollonio. Per semplicità in questa sezione verrà riportato solo il risultato finale in termine di grandezze note assieme ad un paio di figure che meglio rendono l'idea. Per il procedimento nella sua interezza si rimanda al già citato articolo in cui è stato illustrato per la prima volta l'algoritmo[11].

Dopo alcuni semplici passaggi l'angolo può essere quindi espresso come:

$$\phi_{gap\_c} = \cos^{-1}\left(\frac{d_1 + d_2\cos(\phi_1 + \phi_2)}{\sqrt{d_1^2 + d_2^2 + 2d_1 d_2\cos(\phi_1 + \phi_2)}}\right) - \phi_1$$

Il significato dei simboli utilizzati è descritto nell'immagine seguente in cui è schematizzata la geometria del triangolo d'interesse da risolvere.

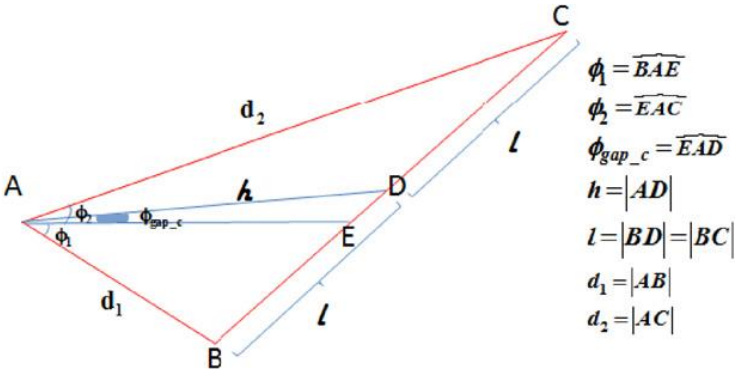


Figura 50: Parametri angolo del centro del gap

3.2.3 Calcolo della traiettoria finale

Ottenuta la direzione verso il gap massimo, che altro non rappresenta se non la direzione con minor rischio di collisioni possibili, non resta che calcolare l'effettiva traiettoria da seguire per poter anche arrivare al punto di arrivo desiderato.

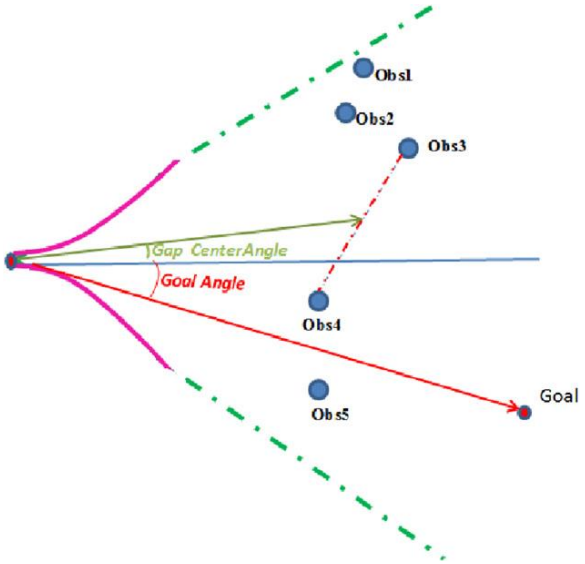


Figura 51: Direzioni verso il centro del gap e verso l'arrivo

La traiettoria finale, ϕ_{final}, altro non è se non una media pesata tra l'angolo verso il centro del gap, ϕ_{gap_c}, e l'angolo di arrivo, ϕ_{goal}. La funzione proposta per il calcolo è quindi la seguente

$$\phi_{final} = \frac{\frac{\alpha}{d_{min}}\phi_{gap\_c} + \beta\phi_{goal}}{\frac{\alpha}{d_{min}} + \beta} \quad dove\ d_{min} = \min_{i=1:n}(d_i)$$

Dove α e β sono parametri di controllo per decidere se prediligere una logica che dia priorità alla sicurezza o alla velocità di arrivo. L'equazione può essere semplificata ulteriormente ponendo $\beta = 1$ in modo da poter regolare il comportamento del robot tramite un singolo parametro α. Il risultato diventerà quindi:

$$\phi_{final} = \frac{\frac{\alpha}{d_{min}}\phi_{gap\_c} + \phi_{goal}}{\frac{\alpha}{d_{min}} + 1} \quad dove\ d_{min} = \min_{i=1:n}(d_i)$$

Da notare anche come nel calcolo influisca anche la distanza dall'ostacolo a cui si trova il robot facendo si che quando ci si avvicina troppo la logica di obstacle avoidance prenda, giustamente, il sopravvento sulla logica di path following.

Nel grafico seguente è mostrato un tipico andamento della traiettoria finale al cambiare del parametro α. La linea blu ($\alpha = 0$) rappresenta ovviamente l'angolo ϕ_{goal}.

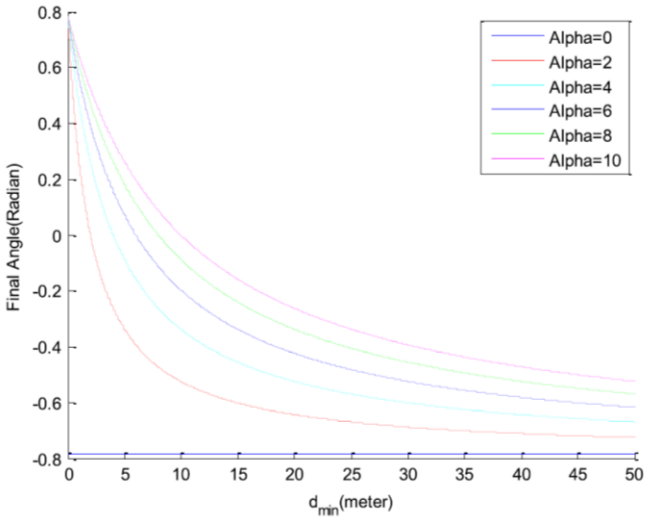


Figura 52: Andamento traiettoria finale al variare del parametro Alpha

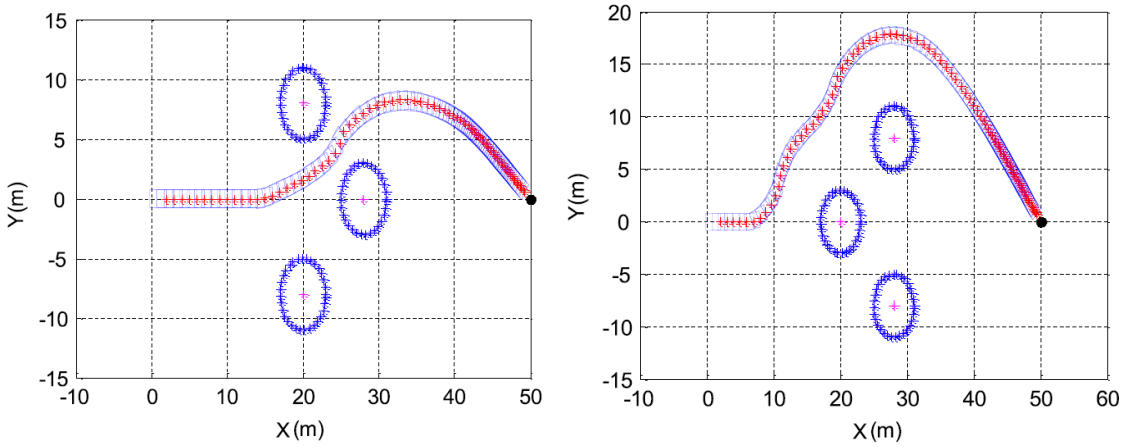


Figura 53: Esempio riassuntivo Follow the Gap Method

Riassumendo si può affermare che nonostante l'estrema semplicità di concetto e di regolazione l'algoritmo risulta piuttosto potente ed efficiente. Oltre a non presentare minimi locali in cui potrebbe rimanere intrappolato le ipotesi su cui si basa lo rendono facilmente applicabile a modelli realistici con vincoli non olonomi.

3.3 Potential Fields Algorithm (PFA)

Altro metodo molto semplice e molto usato. Il robot viene approcciato come una particella puntiforme immersa in un campo di forze[10] in modo da essere attirato verso il traguardo e lontano dagli ostacoli.

Oltre che per applicazioni di obstacle avoidance questo tipo di algoritmo può anche essere usato come pianificatore globale. In generale, tuttavia, vengono preferite molte delle soluzioni proposte nel capitolo precedente in quanto questa classe di algoritmi nonostante un'apparente semplicità di concetto risulta avere difetti non trascurabili. In particolare, proprio per come è definito, questo tipo di pianificatore può portare al rischio di avere minimi locali che di fatto intrappolerebbero il robot; specialmente in presenza di ostacoli posti in maniera simmetrica alla traiettoria. Rimane comunque una valida risorsa per la risoluzione di ostacoli disposti in maniera non eccessivamente labirintica.

Come già detto, si ipotizzi il robot come una particella puntiforme nello spazio di lavoro W, ed il cui spazio delle configurazioni C sia sottoposto a delle forze da quest'ultimo. E' intuitivo capire che le forze saranno di natura attrattiva, F_{att}, per l'arrivo e repulsiva, F_{rep}, per gli ostacoli. Ad ogni istante t_i viene calcolata la risultante di queste forze e conseguentemente viene aggiornata la configurazione q_i,. Tale relazione è più formalmente esprimibile come:

$$F_{tot}(q_{t_i}) = F_{att}(q_{t_i}) + F_{rep}(q_{t_i})$$

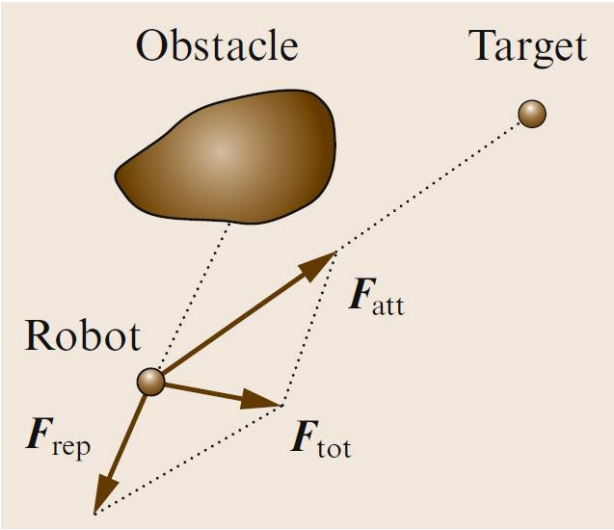


Figura 54: Esempio calcolo della risultante

La forma generica delle varie forze di attrazione e repulsione può essere definita in diversi modi a seconda dello scopo e dei dati a disposizione. Una delle forme più semplici e di

carattere generale è la seguente.

$$F_{att}(q_{t_i}) = K_{att} n_{q_{target}}$$

$$F_{rep}(q_{t_i}) = \begin{cases} K_{rep} \sum_j \left(\frac{1}{d(q_{t_i}, p_j)} - \frac{1}{d_0} \right) n_{p_j}, & d(q_{t_i}, p_j) < d_0 \\ 0 \quad , & d(q_{t_i}, p_j) \geq d_0 \end{cases}$$

Dove:

- ***K_{att}* e *K_{rep}*:** Sono delle costanti che si limitano a controllare il modulo di quelle che sono la forza attrattiva e repulsiva rispettivamente.
- ***n_{qtarget}* e *n_{pj}*:** Sono i vettori unitari che danno la direzione che dal robot si dirige verso il traguardo e che dal robot si dirige verso ogni ostacolo *j*.
- ***d(q_{ti}, p_j):*** E' la distanza tra la configurazione attuale del robot e l'ostacolo p_j.
- ***d₀*:** È la distanza d'influenza dell'ostacolo. Detto semplicemente oltre questa distanza il robot si comporta come se non ci fosse alcun ostacolo.

Questa semplice equazione si limita a descrivere il campo di forze che guida il robot come se fosse composto da una componente attrattiva costante ovunque ed una componente repulsiva che varia inversamente rispetto alla distanza da un qualsivoglia ostacolo.

Sebbene l'equazione mostrata sembri ad un primo sguardo dare un campo soddisfacente e sia di indubbia semplicità in realtà presenta non pochi problemi. Primo fra tutti è il già citato pericolo di minimi locali. Questa condizione porterebbe il robot localmente in equilibrio impedendogli di fatto di raggiungere la meta. Ciò si può verificare facilmente con degli ostacoli posti simmetricamente rispetto al robot o con ostacoli ampi che tendono a circondarlo su più fronti. Un paio di esempi di questa condizione sono riportati nell'immagine seguente.

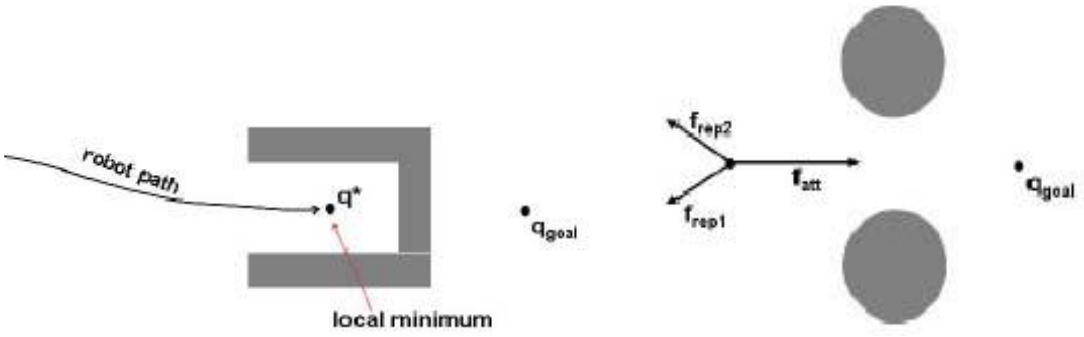


Figura 55: Esempi di trappole per il robot

Uno dei motivi per cui ciò può avvenire è dato dal fatto che le forze applicate dipendono esclusivamente dalla posizione del robot e non dalla sua velocità. Verranno infatti

generate forze repulsive verso il robot indipendentemente dal fatto che quest'ultimo sia fermo o in movimento e che la direzione del movimento effettivamente si diriga verso un ostacolo. Per risolvere questo problema oltre che per garantire anche una miglior rappresentazione cinematica del sistema controllato nei pianificatori PFM moderni si utilizza un'equazione simile a questa:

$$F_{att}(q_{t_i}) = K_{att} n_{q_{target}}$$

$$F_{rep}(q_{t_i}) = \begin{cases} K_{rep} \sum_j \left(\frac{a\dot{q}_{t_i}}{[2ad(q_{t_i}, p_j) - \dot{q}_{t_i}^2]} \right) n_{p_j} \cdot n_{\dot{q}_{t_i}}, & \dot{q}_{t_i} > 0 \\ 0, & \dot{q}_{t_i} \leq 0 \end{cases}$$

Dove:

- $\dot{q}_{t_i}$: È l'attuale velocità del robot.
- $n_{\dot{q}_{t_i}}$: È il versore che punta in direzione della velocità
- a: È l'accelerazione massima del veicolo

Questa nuova espressione è ricavata considerando l'inverso della differenza tra il tempo stimato perché avvenga una collisione e il tempo necessario per invertire il senso di marcia con accelerazione *a*. Da notare come questa volta la formula intervenga con forze repulsive solo nella direzione del moto del robot senza bisogno di vincoli di tipo strettamente geometrico.

Questa relazione è preferita in generale e rende questo metodo molto popolare visto che mantiene una relativa semplicità unita ad un chiaro formalismo matematico.

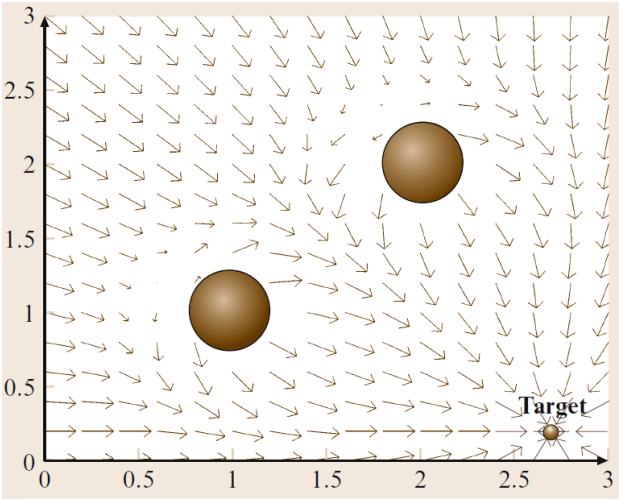


Figura 56: Esempio di campo ottenuto con il metodo classico

Una volta ottenuto il campo la risoluzione del percorso diventa immediata dal momento che si tratta della semplice risoluzione delle equazioni del moto per un corpo immerso in un campo.

3.4 Vector Field Histogram

Ad oggi sicuramente il metodo di obstacle avoidance più popolare. A differenza degli algoritmi precedenti la decisione della direzione da seguire è basata principalmente su un principio probabilistico. Nella zona intorno al robot, infatti, diventa più corretto parlare di zone in cui probabilmente si trova un ostacolo. Questo cambiamento apparente banale in realtà rende l'algoritmo estremamente più robusto rendendolo di fatto impervio al rumore che si può avere da letture di tipo realistico. Per il resto non funziona in maniera molto diversa agli algoritmi già analizzati.

Il suo funzionamento può essere racchiuso in due macro-step[10]:

- **Step 1:** Vengono generate delle traiettorie praticabili dal robot
- **Step 2:** Attraverso l'applicazione del criterio opportuno si sceglie la direzione migliore

Inizialmente lo spazio attorno al robot è suddiviso in settori. Da questa suddivisione viene poi creato un istogramma polare *H* situato attorno al robot in cui ogni componente rappresenta la densità polare degli ostacoli in quel determinato settore. La funzione utilizzata per ricavare la distribuzione degli ostacoli, $h^k(q_{ti})$, nel settore *k* è la seguente:

$$h^k(q_{t_i}) = \int_{\Omega_k} P(p)^n \cdot \left(1 - \frac{d(q_{t_i}, p)}{d_{max}}\right)^m \cdot dp$$

$$\Omega_k = \{p \in W \backslash p \in k \wedge d(q_{t_i}, p) < d_0\}$$

Dove:

- $\boldsymbol{\Omega_k}$**:** È il dominio di integrazione.
- ***P(p)*****:** Probabilità che un punto sia occupato da un ostacolo.

In parole semplici questa formula ci dice che la densità di ostacoli in un dato settore *k* è data dal fatto che un ostacolo ci sia o meno, ovviamente, e dalla distanza a cui ci si trova da esso.

Tipicamente l'istogramma risultante è caratterizzato dalla presenza di picchi (elevata densità di ostacoli) e valli (bassa densità di ostacoli). L'insieme delle direzioni candidate per poter essere seguite è formato dall'insieme di componenti adiacenti che presentano una densità di ostacoli minore di una certa soglia e che tendono a far avvicinare il robot al punto di arrivo.

A questo punto è necessario effettuare la scelta della direzione da intraprendere. Ciò

avviene attraverso l'impiego di tre differenti euristiche in base alla dimensione della valle che si vuole sondare e a quanto questa effettivamente avvicini al traguardo finale. Le tre casistiche sono illustrate di seguito:

1. **Il settore del traguardo è nella valle selezionata:** Il settore da intraprendere selezionato è quello che contiene il traguardo k_{goal}:
$$k_{sol} = k_{goal}$$
2. **Settore del traguardo non è nella valle selezionata e la valle è ampia:** Con valle ampia si intende che al suo interno è presente un numero di settori maggiore di un arbitrario numero m. Il valore di m non è definito a priori e dipende molto dal caso particolare in base alla sensibilità del sensore (numero di settori maggiore o minore a parità di angolo) e dalla mobilità del robot stesso (quanto la cinematica reale permette di intraprendere date direzioni). Si avrà una soluzione:
$$k_{sol} = k_i \pm \frac{m}{2}$$
Dove k_i è il settore più vicino a k_{goal}.
3. **Settore del traguardo non è nella valle selezionata e la valle è stretta:** La definizione di ampia e stretta è identica al caso precedente. In questo caso la soluzione è:
$$k_{sol} = \frac{(k_i + k_j)}{2}$$
Dove k_i e k_j sono i settori estremi della valle.

Ottenuto il settore d'interesse k_{sol} la direzione da intraprendere θ_{sol} sarà la bisettrice del settore e la velocità v_{sol} potrà essere ricavata direttamente da un comando dipendente da direzione scelta e configurazione attuale del robot.

Come già detto l'approccio probabilistico rende l'algoritmo VFH estremamente robusto e performante, adatto ad essere impiegato in applicazioni più realistiche e contesti altamente dinamici. Intrinsecamente, infatti, risulta chiara una gran resistenza a quelli che possono essere i rumori dati dalla sensoristica a bordo. Nell'immagine seguente è brevemente illustrato un esempio di funzionamento dell'algoritmo.

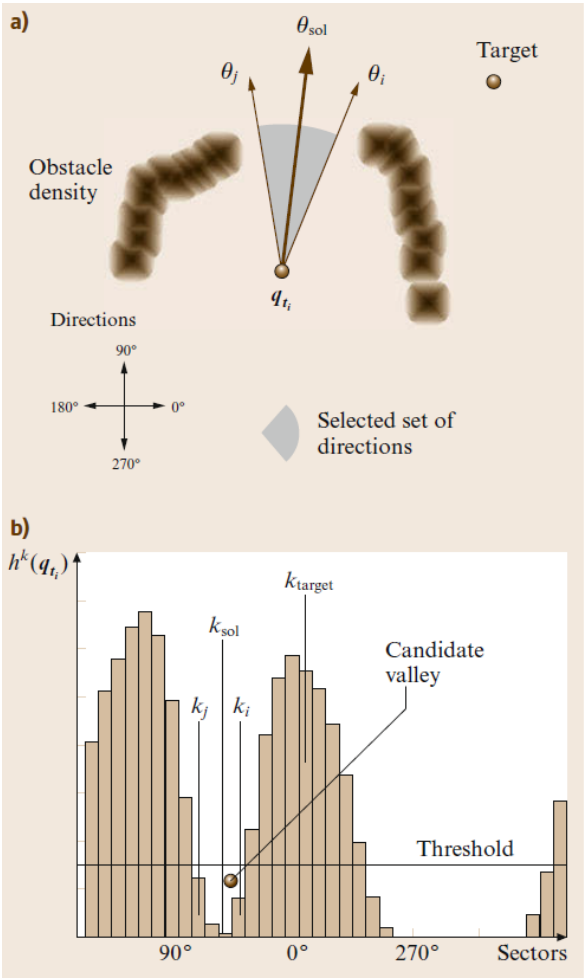


Figura 57: Esempio algoritmo VFH

Capitolo 4

L'ultimo capitolo di questa tesi tratta brevemente quello che è stato un tentativo per la creazione di un sistema AGV tramite Reinforcement Learning.
La logica di controllo funzionante tramite Reinforcement Learning è stata implementata in MATLB grazie alle librerie già presenti di default, mentre il modello rappresentante la dinamica del robot da controllare è stata sviluppata con il software Adams View.
Nelle seguenti pagine verranno discussi molto brevemente i principi teorici sui quali si basa il Reinforcement Learning, come è stato sviluppato il modello con cui effettuare le simulazioni, come sono state effettuate le simulazioni stesse e i risultati ottenuti in merito.

4.1 Reinforcement Learning

Il Reinforcement Learning è una particolare branca del Machine Learning che si occupa del controllo di sistemi particolarmente complessi.
Ciò che differenzia drasticamente questo metodo da tutti i vari algoritmi visti in precedenza è il modo in cui funziona il Reinforcement Learning. Se prima infatti il problema veniva analizzato è risolto da un operatore umano che in seguito si limitava ad implementare la soluzione finale nella macchina qui è la macchina che impara da zero attraverso l'esperienza. Data questa premessa risulta quindi ovvio che lo strumento di simulazione non ha soltanto validità di verifica come negli algoritmi precedenti, ma è anche e soprattutto parte integrante di quello che è il processo di *training* dell'intelligenza artificiale.
Il prodotto finale dovrebbe essere un'intelligenza artificiale in grado di prendere decisioni in maniera autonoma in modo da massimizzare la propria ricompensa in ogni situazione. Il tutto senza interventi umani esterni o programmazione diretta delle azioni da intraprendere. Detto in parole semplici il robot dovrebbe sviluppare un vero e proprio "senso dell'orientamento".

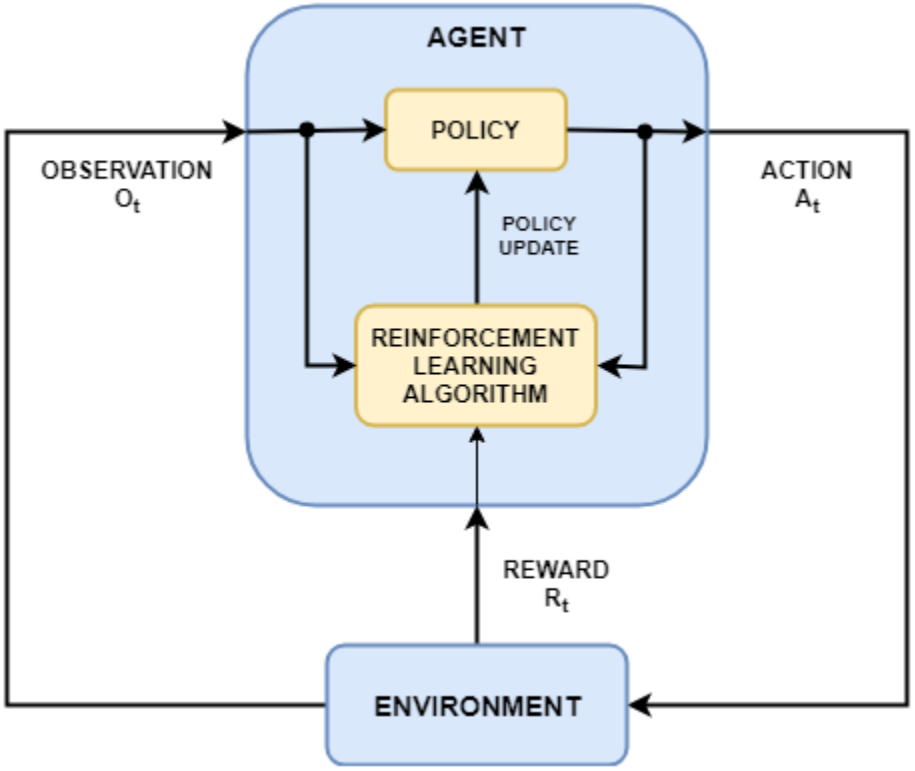


Figura 58: Schema di funzionamento Reinforcement Learning

Lo schema sovrastante[12] mostra brevemente quello che è il principio del Reinforcement Learning. Come si può vedere gli elementi principali che lo compongono sono sostanzialmente due:

- **Agente (Agent):** L'intelligenza artificiale vera e propria. In base alle osservazioni che riceve stabilisce delle azioni e se necessario altera la policy per massimizzare la ricompensa.
- **Ambiente (Environment):** Sostanzialmente tutto ciò che non è l'agente. In questo caso il robot vero e proprio immerso nello spazio in una qualsivoglia configurazione.

Come si può vedere dallo schema l'agente che guida il robot è a sua volta formato da due blocchi distinti al suo interno:

- **Policy:** È il criterio aggiornato con cui l'agente decide che azioni intraprendere in base alle osservazioni che riceve. Può essere visto come una funzione. Molto spesso, specialmente in applicazioni di controllo continuo, tale funzione è ricavata da reti neurali dal momento che la complessità dei problemi in cui viene applicato il Reinforcement Leaning è molto elevata. In sintesi, da *m* osservazioni in input ricava *n* azioni per il controllo in output.

- **Reinforcement Learning Algorithm:** L'algoritmo vero e proprio. Si occupa di aggiornare la policy in base alla ricompensa ottenuta in accordo con le osservazioni e le azioni intraprese ad ogni step. L'obbiettivo finale è molto semplicemente quello di trovare la policy ottimale per il problema di controllo in questione.

Finora sono stati nominati dando per assunto il loro significato, ma prima di procedere oltre è opportuno chiarire meglio il loro ruolo:

- **Osservazioni (Observations):** Sono tutte le grandezze rilevate dall'ambiente ad ogni step. Sono molto semplicemente i segnali che arrivano dalla varia sensoristica di controllo. Possono essere una posizione data da un GPS, una temperatura da un termometro, una forza da un accelerometro e così via. In generale non c'è una regola fissa su quali osservazioni siano necessarie e quali no. Dipende molto dall'applicazione e cosa si vuole ottenere da essa.
- **Azioni (Actions):** Sono gli impulsi di controllo dati dall'agente. Come nel caso delle osservazioni possono essere qualunque cosa da una coppia motrice ad un angolo di sterzo.
- **Ricompensa (Reward):** Forse il segnale più importante e delicato. È sostanzialmente una funzione che "valuta" il comportamento dell'agente in base alle azioni e alle osservazioni ad ogni step. Questa funzione è definita dall'utente e sebbene non esista regole assolute per costruirla il concetto alla base è di dare ricompense negative per azioni che si vogliono scoraggiare (es. una macchina che esce di strada o tampona un ostacolo) e positive per azioni che vogliono essere incoraggiate (es. la macchina si parcheggia correttamente oppure segue la strada senza uscire). L'importanza della ricompensa non è assolutamente da sottovalutare in quanto, come già detto, è il parametro che si vuole massimizzare per avere un controllo ottimale.

Figura 59: Workflow Reinforcement Learning

Il workflow nella figura precedente riassume quelli che possono essere considerati i sette punti fondamentali per applicare correttamente il Reinforcement Learning:

1. **Formulazione del problema:** Definizione del compito da insegnare all'agente, inclusi gli obbiettivi primari e secondari da ottenere e come quest'ultimo interagisce con l'ambiente esterno.
2. **Creazione dell'ambiente:** Definizione dell'ambiente in cui l'agente opera, inclusa l'interfaccia tra agente e ambiente (osservazioni) ed un modello dinamico di quest'ultimo
3. **Definizione della ricompensa:** Forse la parte più delicata. Bisogna definire il segnale di ricompensa che è quello che nella pratica guida l'agente verso gli scopi che si vogliono ottenere.
4. **Creazione dell'agente:** L'agente va creato assemblando la policy e l'algoritmo con cui essa impara e si aggiorna.
5. **Allenamento dell'agente:** Usando l'ambiente, la funzione di ricompensa e l'algoritmo di apprendimento dell'agente si procede con quello che è l'allenamento per ottenere la policy finale. Per applicazioni di Reinforcemente Learning dovendo sovente allenare delle reti neurali si tratta di un processo molto lungo e computazionalmente oneroso.
6. **Validazione dell'agente:** Vengono simulati l'agente con la policy allenata e l'ambiente insieme in modo da avere una valutazione della bontà della policy ottenuta.
7. **Estrazione della policy ottimale:** Molto semplicemente una volta ricavata la policy ottimale questa andrà estratta per essere applicata all'uso desiderato.

Come già detto il processo per allenare l'agente è ricavare la policy è un processo lungo e di tipo iterativo. È molto probabile che per ottenere un risultato accettabile sia necessario andare avanti e indietro nel workflow più volte. Questo, insieme all'onere computazionale intrinseco di ogni tentativo, è parte integrante del perché questo tipo di metodologie non è ancora di ampio uso nella pratica quotidiana. I parametri da dover aggiornare possono essere riassunti nelle seguenti categorie:

- Opzioni di allenamento
- Configurazione dell'algoritmo di apprendimento
- Costituzione della policy
- Definizione della funzione di ricompensa

- Segnali di osservazioni e azioni
- Dinamica dell'ambiente

Bisogna inoltre tenere in conto che ognuna di queste categorie di parametri è caratterizzata da ulteriori iperparametri e che in generale non esistono regole empiriche per aggiornarli di volta in volta.

Nelle pagine successive verrà analizzato molto brevemente il procedimento svolto per cercare di applicare il Reinforcement Learning ad un'applicazione AGV e alcune delle difficoltà che hanno portato ad un sostanziale insuccesso dell'applicazione verranno discusse più nel dettaglio in seguito.

4.2 Modello in ADAMS View del veicolo

Come prima cosa, una volta definito il problema, è stato necessario costruire un modello dell'ambiente di lavoro da collegare all'intelligenza artificiale.
Sebbene la logica di controllo si trovi interamente su Matlab e Simulink per ragioni di semplicità ci si è affidati alla simulazione del comportamento dinamico del veicolo al software Adams View, specificatamente dedicato al multibody. Vista l'opzione di poter esportare il modello in ambiente Simulink in seguito la scelta è stata quindi considerata ragionevole in un primo momento.
Il modello è quanto di più semplice si possa immaginare. Lo scopo principale, infatti, non era tanto un modello iperrealistico quanto un modello che desse risposte dinamiche ragionevoli agli impulsi di controllo.

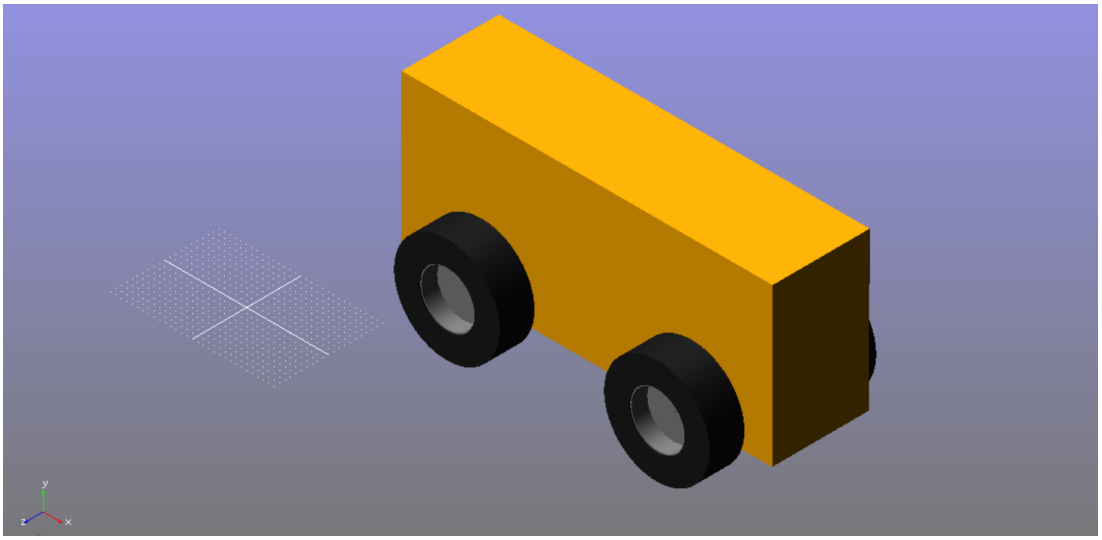


Figura 60: Modello Adams

Più nel dettaglio, la cinematica è stata ipotizzata nella seguente maniera. Il robot è a trazione posteriore con sterzata adibita alle ruote anteriori tramite sistema Ackermann[13].
In maniera molto semplice il sistema di sterzata Ackermann ha la seguente funzione: durante la sterzata del veicolo fa sì che ruote interne ed esterne al raggio di sterzata abbiano angoli di sterzata differenti in modo da evitare lo slittamento delle ruote in curva. Il funzionamento risulta chiaro dalla seguente figura dove è possibile osservare l'effetto del meccanismo durante la sterzata.

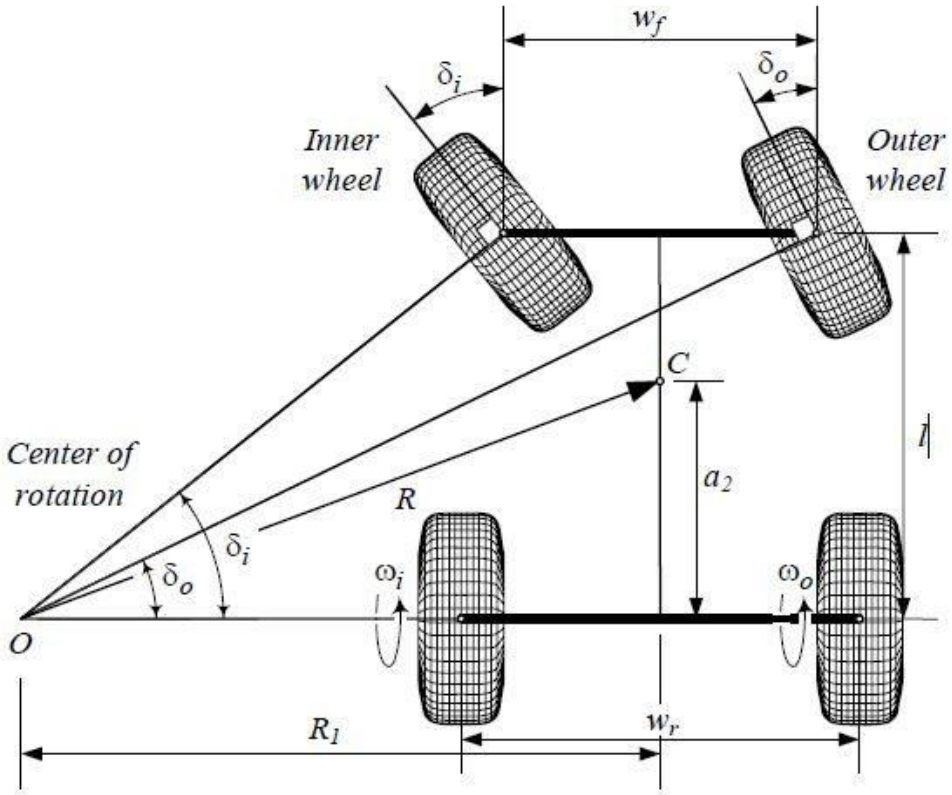


Figura 61: Sterzata Ackermann

Come è possibile osservare il sistema fa sì che anche durante l'azione di sterzata le ruote mantengano tutte lo stesso centro di istantanea rotazione garantendo l'assenza di slittamento e permettendo di poter riassumere matematicamente la traiettoria del veicolo nel suo centro di massa. Tale modello è detto "modello a bicicletta"[14]. Più formalmente quindi è ora possibile definire la traiettoria solo grazie all'angolo δ che rappresenta la traiettoria intermedia del veicolo e che quindi ci permette un controllo più diretto sul raggio di sterzata. Considerando l'angolo δ come parametro noto ed imposto dall'alto gli angoli esterni delle ruote si possono esprimere come:

$$\delta_i = \tan^{-1}\left(\frac{2l\sin\delta}{2l\cos\delta - w_r\sin\delta}\right)$$

$$\delta_o = \tan^{-1}\left(\frac{2l\sin\delta}{2l\cos\delta + w_r\sin\delta}\right)$$

A questo punto il raggio di curva può essere espresso come:

$$R_1 = \frac{l}{\tan\delta}$$

Per l'applicazione in questione si è scelto un angolo $\delta = 50°$ che conferisce al veicolo un raggio di sterzata $R = 1.62\ m$.

Risolta la cinematica l'ennesimo aspetto da curare è stato quello dinamico. Per i valori d'inerzia si è cercato di rendere la somiglianza dei valori con veicoli compatti da lavoro agricolo. Tale aspetto seppur in maniera approssimata è stato trattato con cura in quanto una risposta di forze e momenti quanto più realistica è fondamentale non solo per un mero aspetto dinamico, ma anche per il successivo allenamento delle reti neurali atte al controllo. Nella tabella sottostante sono riassunti i principali valori inerziali del modello.

Proprietà inerziali	
Massa [kg]	2873
Tensore d'inerzia baricentrico	
Ixx [kg*m^2]	1207
Iyy [kg*m^2]	3274
Izz [kg*m^2]	3552

Tabella 1: Proprietà inerziali

Definite le proprietà d'inerzia è stato poi fondamentale stabilire la legge di contatto ruota-terreno. La scelta del tipo di forza è stata di una forza d'impatto con attrito coulombiano e nella tabella seguente sono elencati i valori selezionati per i vari coefficienti.

Forza normale		**Forza d'attito**	
Esponente della forza	2.2	**Coefficiente d'attrito statico**	0.6
Rigidezza	1.0E+09	**Coefficiente d'attrito dinamico**	0.4
Smorzamento	5.0E+04	**Velocità di transizione di distacco**	0.1
Profondità di penetrazione	1.0E-0.5	**Velocità di transizione d'attrito**	1

Tabella 2: Coefficienti del contatto ruota-terreno

Come ultimo passaggio effettuato sul software Adams View sono infine stati scelti quelli che sarebbero stati i parametri di controllo del modello derivanti dalle azioni dell'agente e le osservazioni necessarie per la decisione di tali azioni.

Per quanto concerne le azioni di controllo si è optato per due coppie differenti. La prima coppia è quella motrice da impartire alle ruote posteriori per generare il movimento e la seconda una coppia fittizia che andasse a simulare l'azione di un volante che tenta di sterzare.

Per quanto riguarda la coppia motrice sono stati imposti diversi vincoli in quanto è stato necessario che potesse simulare l'azione di un motore reale. Fino ad una velocità angolare di 40 *rpm* delle ruote si ha un regime di coppia nominale di 1000 *Nm* sull'asse mentre per velocità successive si passa ad un regime di potenza costante di 4.189 *kW*. La caratteristica è brevemente sintetizzata nell'immagine sottostante.

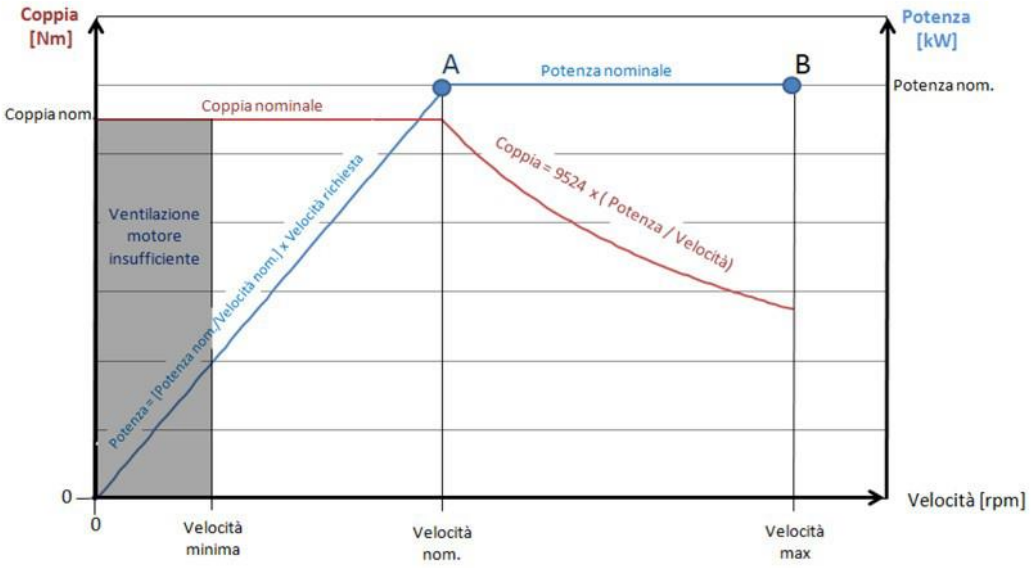


Figura 62: Caratteristica meccanica del motore

Una volta stabilite le grandezze di controllo sono successivamente state scelte le osservazioni da utilizzare. Per effettuare un controllo di base sono state quindi scelte le variabili rappresentanti la posizione e direzione del centro di massa del robot $\{x,y,\theta\}$ e quelle rappresentanti lo stato cinematico, ovvero la velocità e la velocità angolare delle ruote $\{v,\omega\}$.

4.3 Implementazione del modello Adams in Simulink

Una volta ottenuto un modello soddisfacente in ambiente Adams è stato necessario trasportarlo su Simulink in modo da poter iniziare la fase di simulazione con le reti neurali.

È a questo punto del lavoro che è stata riscontrata la prima grande difficoltà di realizzazione dell'intero progetto. Se infatti già di per sé non è un'operazione banale riuscire ad importare il modello ADAMS in ambiente Simulink; prevalentemente per problematiche relative ad intervalli di integrazione e comunicazione tra i blocchi; la difficolta aumenta nuovamente quando entra in gioco la libreria Matlab relativa al Reinforcement Learning. Come già detto, infatti, per essere allenato l'agente deve seguire un processo iterativo di tipo trial-and-error, ovvero deve simulare *n* volte il controllo del robot in una situazione tipo. Ognuna di queste simulazioni si definisce un *episodio*. Normalmente ogni volta che un episodio finisce dovrebbe avvenire un reset dell'ambiente; tuttavia, per ragioni che non è stato possibile comprendere ciò non avviene per il blocco del modello Adams. Ciò comporta due problemi particolari. Il primo, che si è dimostrato risolvibile, le variabili di stato, che vengono direttamente dal blocco Adams, non resettandosi di fatto fanno ripartire ogni volta il robot nelle condizioni in cui aveva terminato l'episodio precedente segnalando quindi un successo o un fallimento prima ancora che sia possibile effettuare un vero e proprio controllo. Tale problema è stato risolto estrapolando da Adams le derivate delle grandezze di interesse che venendo in seguito integrate in Simulink permettevano il reset delle condizioni iniziali.

Il secondo problema, che si è dimostrato irrisolvibile, è che conservando comunque internamente le informazioni relative alle variabili di stato e potendo gli episodi arrivare anche ad essere migliaia il blocco Adams dopo un po' diventava di fatto incontrollabile. Ciò è anche dovuto al fatto che nell'ambiente di lavoro di Adams View non è presente un "terreno infinito" e quindi dopo un po' era semplicemente come se il robot cadesse dal terreno diventando di fatto incontrollabile.

L'ultimo problema si è rivelato insormontabile cercando di mantenere il blocco Adams collegato all'agente da allenare così è stato necessario creare un surrogato di quest'ultimo. Molto semplicemente dal modello Adams originale ne è stato interpolato uno nuovo in ambiente Matlab e Simulink. L'interpolazione è avvenuta attraverso delle funzioni polinomiali che approssimassero le azioni di accelerazione e sterzata rispettivamente. I

due comportamenti sono poi successivamente assemblati nel seguente sottosistema Simulink.

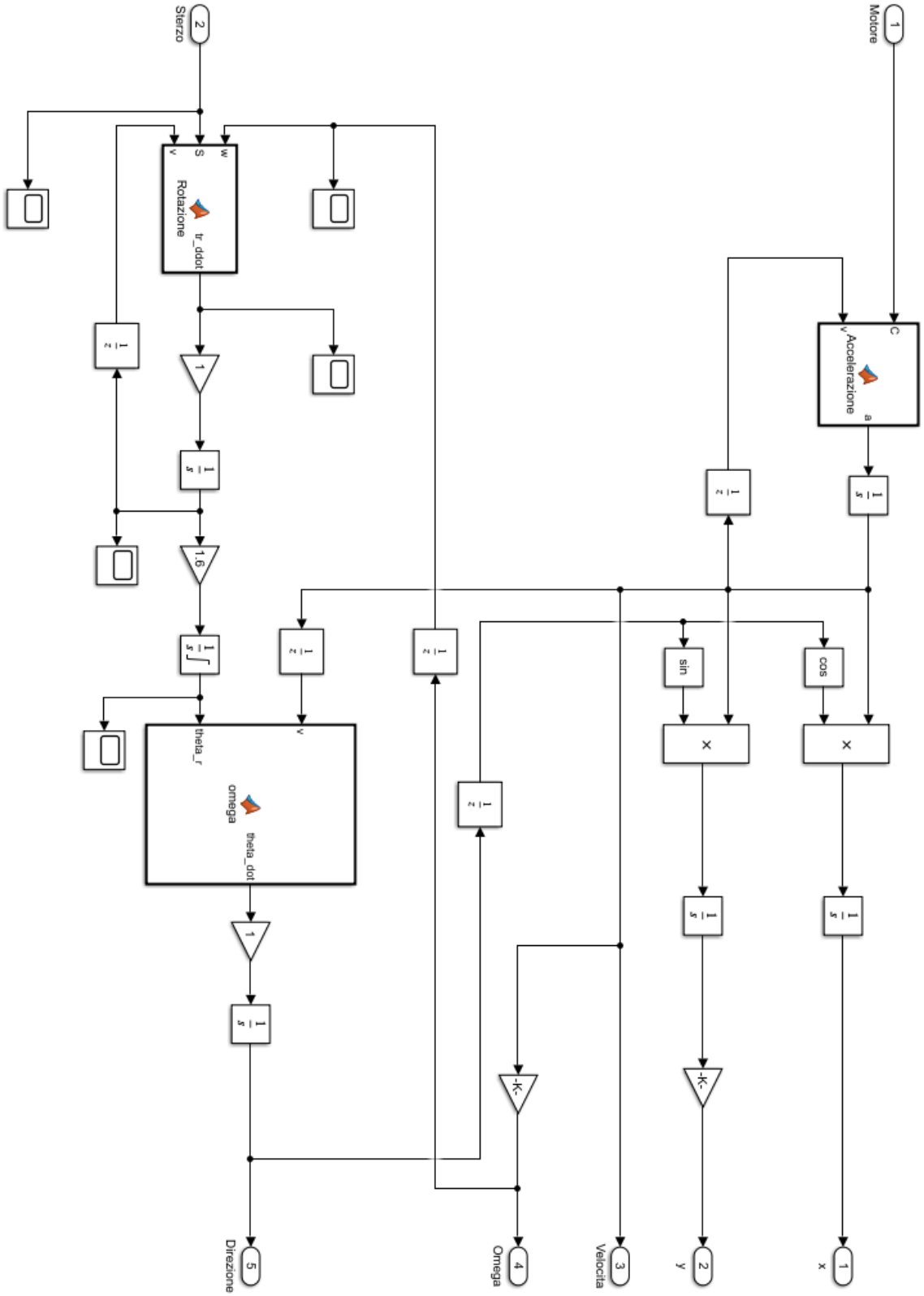


Figura 63: Approssimazione del modello Adams in Simulink

Il modello così ottenuto per la natura di tipo semplificato con cui è stato ricavato non è sicuramente perfetto. Tuttavia, per periodi brevi i comportamenti dei due modelli non differiscono di molto e il nuovo modello si resetta ad ogni nuovo episodio dell'allenamento. Sotto sono riportati i grafici delle posizioni *x* e *y* occupate dal robot a parità di impulsi e condizioni iniziali.

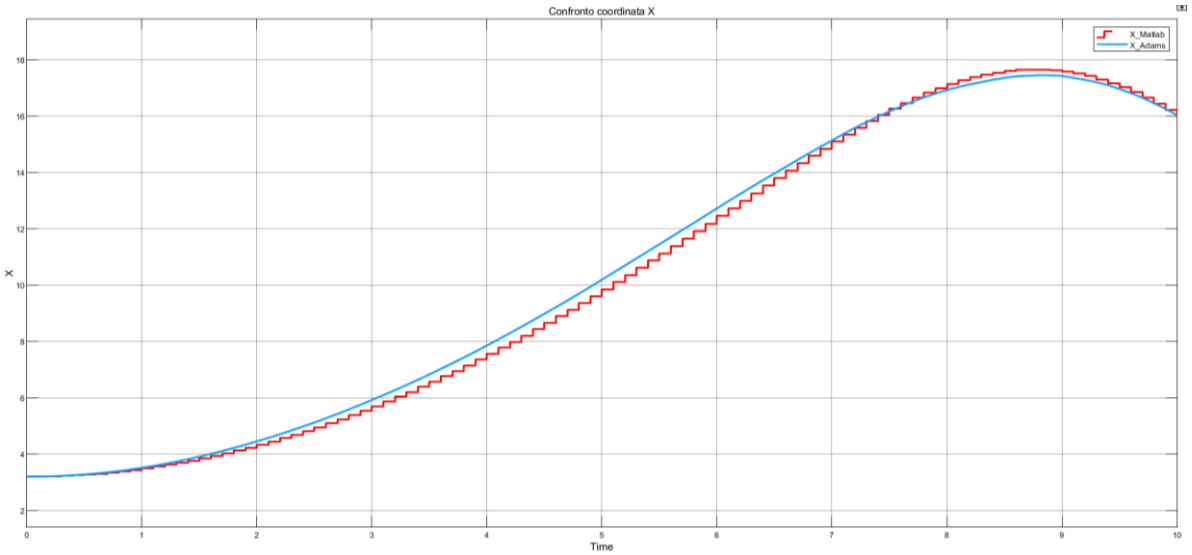


Figura 64: Confronto coordinata X

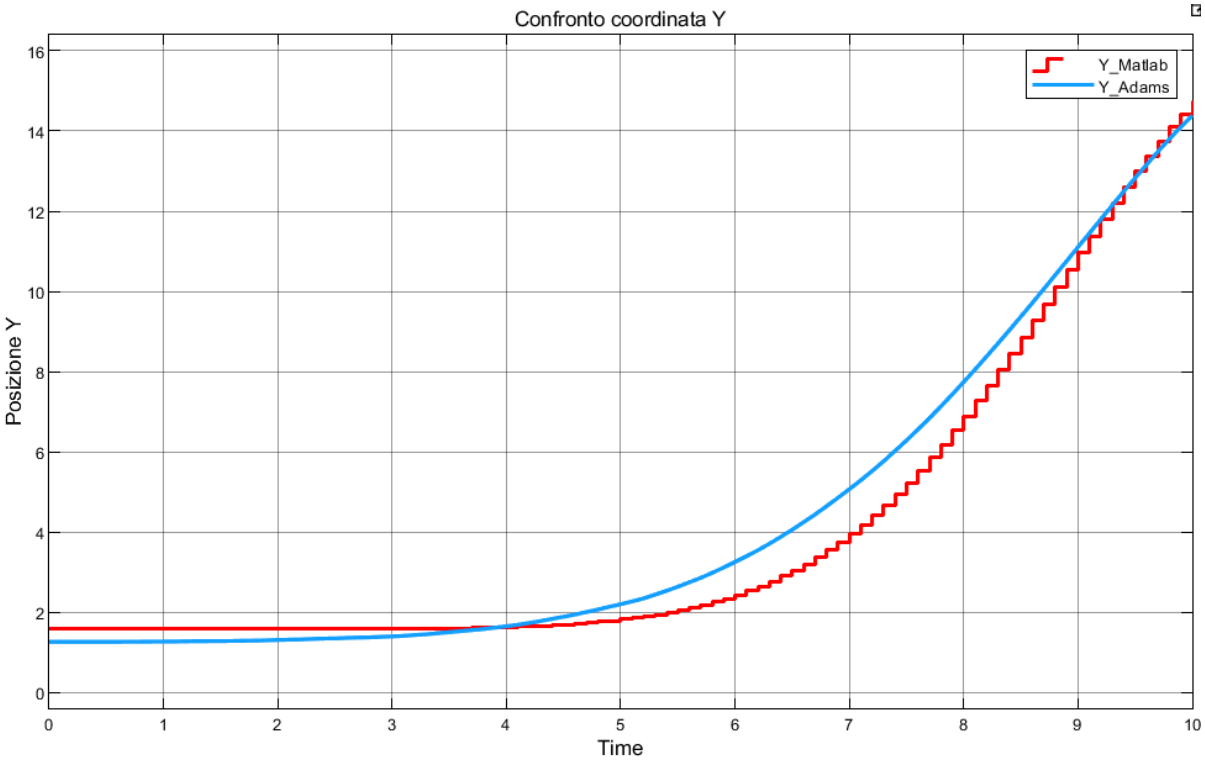


Figura 65: Confronto coordinata Y

4.4 Costruzione logica di controllo e modello finale

Ottenuto un modello funzionante in ambiente Matlab e Simulink è finalmente stato possibile applicare una logica di controllo tramite Reinforcement Learning al robot.
L'intero modello può essere riassunto in quattro blocchi che permettono di riassumere il funzionamento generale dell'intero sistema.
Il primo blocco, ovviamente, è il blocco che rappresenta il modello dinamico del robot già discusso nel sotto capitolo precedente.

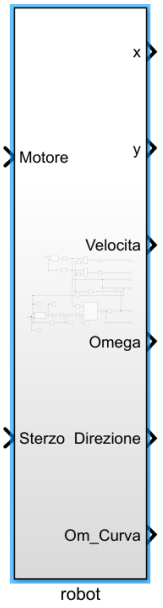


Figura 66: Blocco robot

Il secondo blocco è il controllore vero e proprio del sistema, ovvero l'agente del Reinforcement Learning. Come già accennato è possibile utilizzare varie tipologie di agente per poter ottenere l'obbiettivo desiderato. Senza dilungarsi nell'esegesi di tutte le possibili varianti verrà ora brevemente illustrata la tipologia di agente adottata. L'agente scelto è del tipo comunemente identificato come *Deep Deterministic Policy Gradient (DDPG)*. Questa tipologia di agente è ideale per applicazioni caratterizzate da osservazioni ed azioni di tipo continuo e quindi ideale per il controllo di sistemi dinamici come appunto il sistema AGV in questione. La policy di questa tipologia di agente è essenzialmente formata da due diverse componenti:

- **Critico (Critics):** Riceve in input sia le azioni intraprese che le osservazioni generate da queste ultime. Sulla base di ogni insieme di input restituisce la ricompensa totale attesa per il resto dell'episodio. Di fatto è l'euristica che si

utilizzava negli algoritmi precedenti. Nel Reinforcement Learning anzi che essere ipotizzata a monte viene stabilita durante l'allenamento.

- **Attore (Actor):** Riceve in input le osservazioni e restituisce come output le azioni che ritiene possano massimizzare la ricompensa totale per il resto dell'episodio.

In generale non è obbligatorio utilizzare sia critico che attore in un agente, ma per questo tipo di applicazioni risulta estremamente più vantaggioso. Un agente formato dal solo critico, infatti, mal gestisce ambienti con azioni continue e rischia di diventare computazionalmente troppo oneroso. D'altro canto, un agente formato da solo attore sebbene possa gestire facilmente azioni di tipo continuo risulta poco robusto ad eventuali rumori e rischia di convergere a minimi locali. Un agente formato da entrambi riesce ad arginare le debolezze di entrambi gli approcci anche se non va dimenticato che essendo entrambi rappresentati da reti neurali l'allenamento può richiedere parecchio tempo.

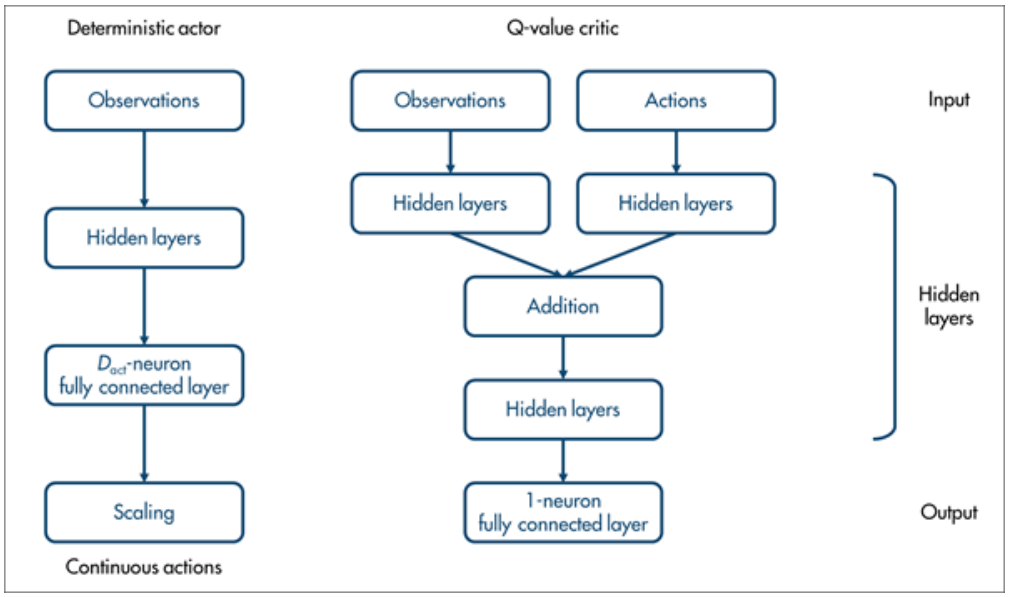


Figura 67: Struttura di attore e critico

Per l'applicazione attuale le reti neurali sono state costruite con i seguenti parametri:

Attore	
Layer di input (Osservazioni)	[5,1]
Numero di hidden layers	3
Neuroni per hidden layer	70
Layer di output (Azioni)	[2,1]

Tabella 3: Parametri attore

Critico			
Osservazioni		**Azioni**	
Input	[5,1]	**Input**	[2,1]
N. hidden layers	3	**N. hidden layers**	2
Neuroni per hl	70	**Neuroni per hl**	70
Addition layer			
Layer di output (Ricompensa prevista)		[1,1]	

Tabella 4: Parametri critico

Nel modello Simulink il modello che racchiude al suo interno l'agente DDPG è il seguente:

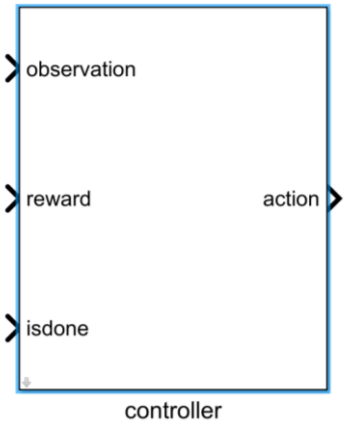


Figura 68: Blocco agente

Il terzo blocco altro non è se non la ricompensa assegnata per le azioni intraprese dal robot.

La ricompensa è ovviamente legata a ciò che si vuole ottenere dall'agente. In questo caso il compito è uno dei più semplici possibili se non addirittura il più semplice. Il robot deve essere in grado di spostarsi in linea retta dal punto *A* al punto *B*. La funzione ideata per lo scopo è la seguente:

$$r = 1 - e^{\sqrt{d_x^2 + d_y^2}} + 5 * madeit - 2 * collided - 0.01 * \omega_{curva} - \frac{1}{1 + v^2}$$

I vari addendi hanno tutti un loro significato. Il primo punisce il fatto di essere distanti dall'obbiettivo, il secondo (unico positivo) premia l'arrivo al traguardo, il terzo punisce le collisioni, il quarto punisce il fatto di girare in tondo e il quinto ed ultimo punisce il

fatto di stare fermi. Questo è solo un esempio delle funzioni ricompensa tentate per le varie simulazioni, in particolare quello associato al tentativo migliore che è stato ottenuto. Il blocco che assegna la ricompensa in Sinulink è il seguente.

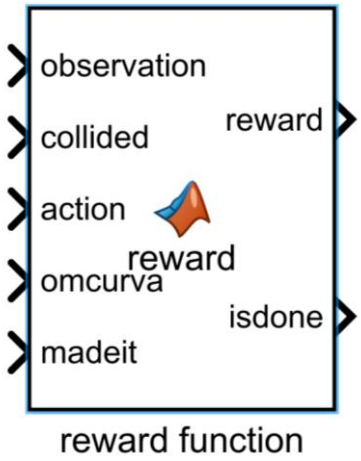


Figura 69: Blocco ricompensa

L'ultimo blocco molto semplicemente è il blocco per la rilevazione delle collisioni ed il calcolo dei parametri per la rappresentazione grafica del robot. Il blocco è rappresentato nell'immagine seguente.

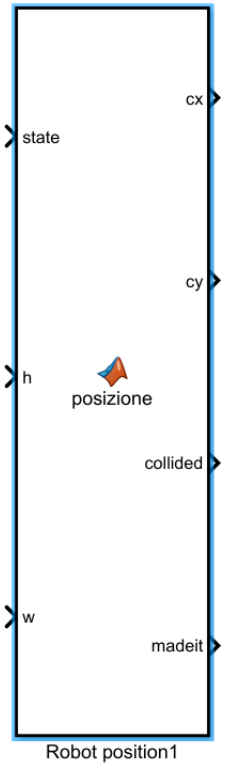


Figura 70: Blocco di rilevamento collisioni

Una volta assemblati tutti i vari blocchi tra loro il modello finale rappresentante risulta essere quello nell'immagine della pagina successiva.

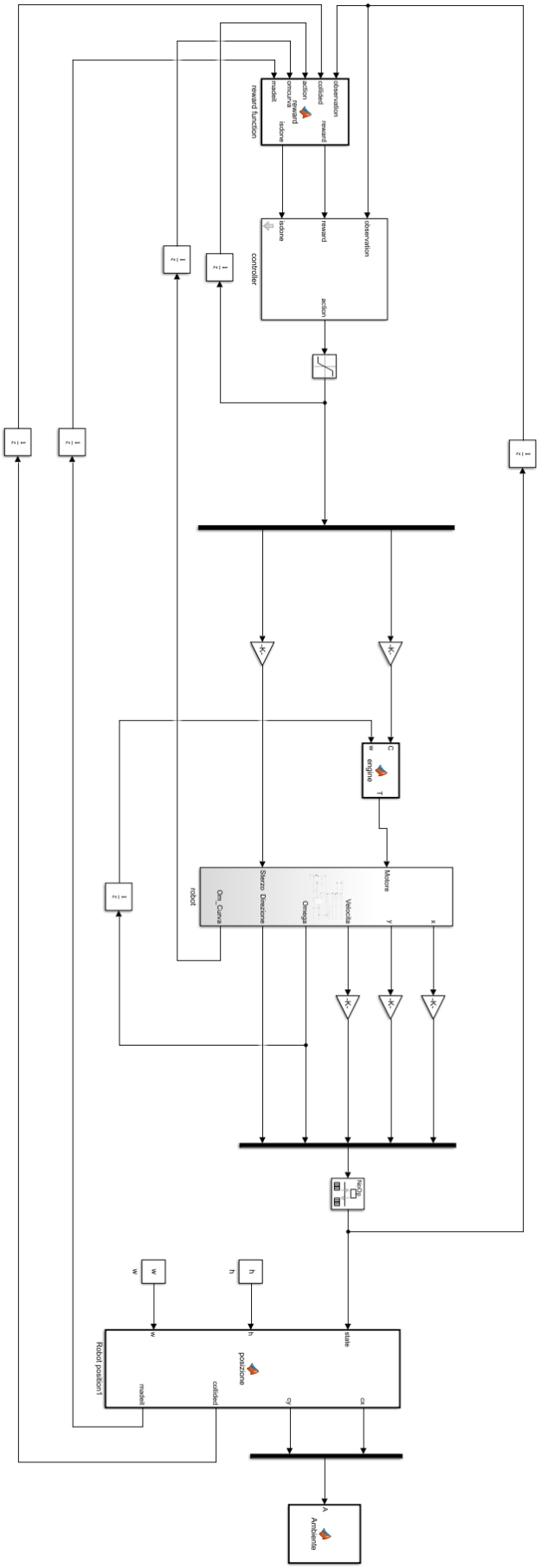


Figura 71: Modello completo

4.5 Risultati

Come già accennato i risultati non si sono rivelati molto positivi. Il tentativo migliore è stato allenato per 20000 episodi e non è tuttavia riuscito a convergere alla soluzione perfetta. Come si può vedere dalla figura successiva il robot riesce a percorrere i circa 20 metri verso l'ostacolo, ma la traiettoria mostra chiaramente come ci sia ancora la tendenza a discostarsi dall'ovvio percorso ottimale per continuare ad esplorare.

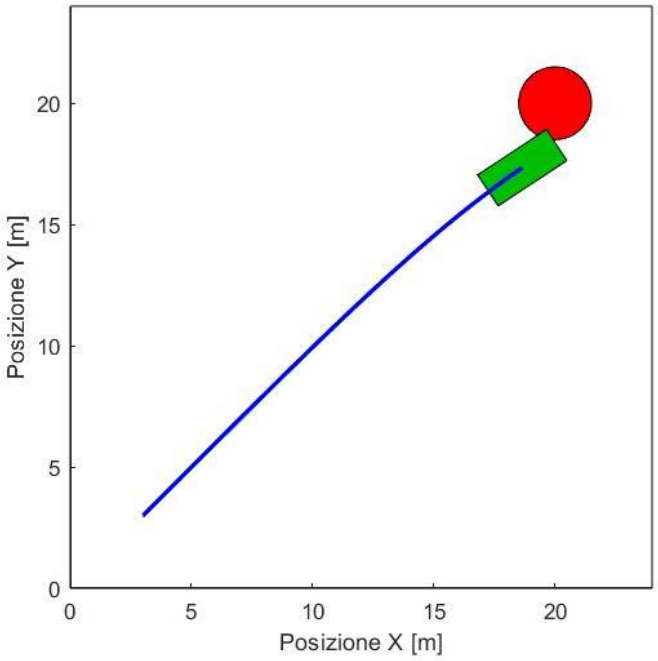


Figura 72: Traiettoria finale

Da questa applicazione risulta immediato capire perché il Reinforcement Learning ancora non sia una tecnologia sufficientemente competitiva con quelle rappresentate nel resto della tesi. Nonostante riesca a portare a termine i compiti assegnati è un tipo di tecnologia che presenta problemi di arbitrarietà e tempistiche troppo grandi per non essere tenuti in conto. Il vero tallone d'Achille sta infatti nella gestione degli iperparametri dell'allenamento e delle reti neurali stesse. Tali parametri sono molti e non esiste un algoritmo o un crivello che permetta di stabilirli a priori. Oltretutto il tipo di ambiente utilizzato deve garantire una certa varietà di casistiche per permettere di ottenere una policy sufficientemente flessibile per l'applicazione sul campo. Tutto questo unito ai costi computazionali immensi intrinseci del metodo hanno decretato la bocciatura di questo tipo di soluzione rispetto a quelle proposte in precedenza.

L’unico frangente in cui questo tipo di soluzione potrebbe essere considerato superiore ai precedenti sarebbe il caso di un ambiente troppo complesso per poter essere tradotto in un algoritmo da operatori umani e la disponibilità di una potenza di calcolo estremamente elevata. Per quest’ultimo punto danno speranza i computer quantistici che di recente sono un argomento di ricerca decisamente scottante proprio come il Reinforcement Learning, o più in generale il Machine Learning nel suo complesso.

Conclusioni

Il lavoro di tesi proposto si è prefisso l'obbiettivo di illustrare quello che è lo stato attuale delle tecnologie di path planning e obstacle avoidance tentando anche di sperimentare possibili coinvolgimenti futuri del Machine Learning.

Quelli proposti sono solo alcuni degli algoritmi più utilizzati nella pratica moderna ed hanno il semplice scopo di dare una panoramica di quello che ad oggi è sicuramente un settore in grande fermento. Ogni giorno vengono pubblicati nuovi articoli con nuove proposte di algoritmo o raffinamento di algoritmi già esistenti come illustrato per esempio negli algoritmi sample-based (da RRT fino a Kinodynamic RRT*). Per maggiore dettaglio degli algoritmi trattati e per la ricerca di algoritmi alternativi a quelli proposti si invita a fare riferimento alla bibliografia dove sono elencati gli articoli da cui è stato ricavato il presente lavoro.

Come invece dimostrato, per ora, la tecnologia del Reinforcement Learning non risulta competitiva con le altre tipologie di algoritmi proposti. Gli eccessivi costi computazionali e la mancanza di una vera e propria possibilità di esercitare un controllo empirico sui risultati finali rendono questo tipo di soluzione inadatta ad un'applicazione ingegneristicamente rilevante.

Bibliografia

[1] P. Torino “Studio di algoritmi e co-simulazione per la guida autonoma di veicoli da lavoro”, 2019

[2] P. Milano and I. Industriale “Confronto di algoritmi di pianificazione di traiettoria sampling-based per robot mobili”, 2018

[3] Steven M. LaValle, “Planning Algorithms”, 2006

[4] S. Vigna, “L’algoritmo di Dijkstra”, 2006

[5] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, S. Thrun “Principles of Robot Motion”

[6] M. Piastra “Intelligenza Artificiale: Ricerca euristica L’algoritmo A*”, 2005

[7] “Intelligenza artificiale: algoritmo A*”. Online. Available: https://engineering.facile.it/blog/ita/intelligenza-artificiale-algoritmo-a-star/. [Accessed: 29-Nov-2021].

[8] S. Karaman, E. Frazzoli, “Sampling-based Algorithms for Optimal Motion Planning”, 2011

[9] D. J. Webb, J. van der Berg, “Kinodynamic RRT*: Optimal Motion Planning for Systems with Linear Differential Constraints”, 2012

[10] B. Siciliano, O. Kathib, “Springer Handbook of Robotics”, Springer-Verlag Berlin Heidelberg 2016

[11] V. Sezer, M. Gokasan, “A novel obstacle avoidance algorithm: “Follow the Gap Method””, 2012

[12] “Reinforcement Learning Toolbox User’s Guide”, Mathworks, 2021

[13] R. Eisele, “Ackermann Steering”. Online. Available: https://www.xarg.org/book/kinematics/ackerman-steering/. [Accessed: 29-Nov-2021].

[14] D. Wang, F. Qi, “Trajectory Planning for a Four-Wheel-Steering Vehicle”, 2001

[15] R. L. Norton, “Adams Tutorial Kit for Mechanical Engineering Courses”, 2013

[16] J. A. Oroko, “Obstacle Avoidance and Path Planning Schemes for Autonomous Navigation of a Mobile Robot: A Review”, 2012

[17] V. A. Bhavesh, “Comparison of Various Obstacle Avoidance Algorithms”, 2015

[18] P. Torino, “Obstacle Avoidance Algorithms for Autonomous Navigation system in Unstructured Indoor areas”, 2018

[19] J. Borenstein, Y. Koren, “The Vector Field Histogram – Fast Obstacle Avoidance for Mobile Robot”, 1991

[20] M. Zohaib, M. Pasha, R. A. Riaz, N. Javaid, M. Ilahi, R. D. Khan, “Control Strategies for Mobile Robot with Obstacle Avoidance”, 2013

[21] A. Swingler, Dike University, “A Cell Decomposition Approach to Robotic Trajectory Planning via Disjunctive Programming”, 2012

[22] Michael A. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015

[23] Stuart J. Russel, P. Norvig, “Artificial Intelligence A Modern Approach Second Edition”, 2003

[24] “Reinforcement Learning Onramp” [Online], Mathworks, Available: https://matlabacademy.mathworks.com/details/reinforcement-learning-onramp/reinforcementlearning. [Accessed: 29-11-2021].