Appunti di Robotica

Claudio Metelli

A.A. 2024-2025

Contents

1 Introduzione

I robot nascono dalla filmografia, e si sviluppano per poter adempire a svariate funzioni, che possiamo rivedere principalmente nelle 3D:

Dirty: task in ambienti sporchi oppure contaminati, come ad esempio pulizia in ambienti tossici o industriali.

Dangerous: task in ambienti ad alto pericolo, ad esempio scenari di guerra oppure per disinnescare bombe

Dull: task ripetitivi oppure faticosi, per cui è meglio avere una alta produttività.

Inoltre i robot sono utilizzati anche in ambienti di assistenza, intrattenimento ed altro. Una prima tassonomia per la robotica riguarda l'impiego dei robot:

- Unmanned Ground Vehicles (UGV), robot terrestri
- Unmanned Aerial Vehicles (UAV), robot di aria
- Unmanned Marine Vehicles (UMV), robot d'acqua

Esistono vari tipi di robot come visto, secondo diverse tassonomie, ma nell'ambito del corso, sono i **robot intelligenti** ad essere di nostro interesse: si intende un agente intelligente fisicamente situato in un luogo; un agente intelligente è un sistema che percepisce l'ambiente circostante e prende azioni che massimizzano le sue probabilità di successo. Si noti dunque che il robot agisce sul mondo e lo cambia.

Vista questa tipologia di robot, è necessario ricordare la differenza tra automazione ed autonomia.
Nel caso di **automazione** si parla di un ambito che riguarda robot industriali che eseguono task operativi predefiniti e ben definiti, che utilizzano l'assunzione di mondo chiuso (per la quale le informazioni a me non note sono considerate false, cioè sono false le affermazioni che non sono esplicitamente vere), per cui le informazioni e le situazioni sono note a priori e possono essere modellate in precedenza, eliminando il sensing.
Nel caso di **autonomia** si parla di agenti che si possono adattare in una assunzione di mondo aperto (per la quale le informazioni a me non note sono considerate vere, cioè sono vere le affermazioni che non sono esplicitamente false), dove ambienti e task non sono noti a priori, e dunque bisogna gestire e generare nuovi piani, sulla base del sensing del robot.

Figure 1: Esempio di automazione, un robot industriale ad esempio per chiudere bottiglie con i tappi, ed esempio di autonomia, un rover per ambienti come marte.

2 Architetture per l'Autonomia

La tassonomia delle architetture robotiche prevede tre diversi tipi di architettura, a seconda di come il sistema funziona, da tre diversi punti di vista.

Operational: descrive cosa fa il sistema ad alto livello, ma non come.

System: descrive come lavora un sistema in termini dei maggiori sottosistemi.

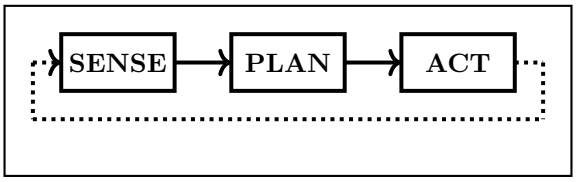
Technical: descrive come funziona un sistema in base ai dettagli implementativi.

Le *architetture operazionali* (operational) sono delle tipologie di architetture a tre livelli: **deliberative**, **reactive (o behavioral)**, **interaction**. Tali livelli hanno diversi scopi.

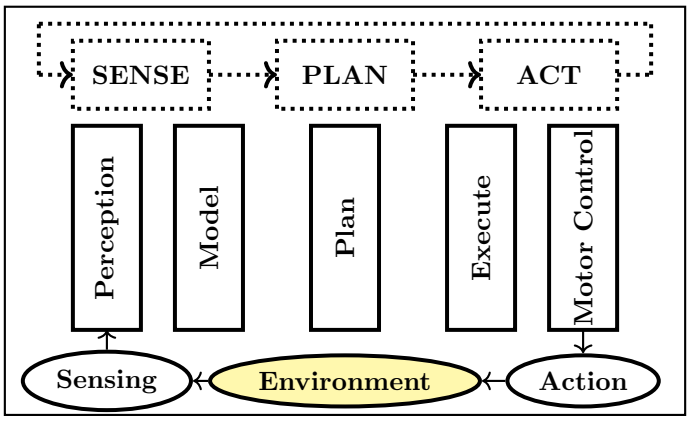
- Il *piano deliberativo* (deliberative) si occupa della programmazione a lungo termine, dunque del reasoning e del planning, attraverso i dati fornitogli da vari sensori che vengono adeguatamente trasformati in simboli; si associa a linguaggi funzionali come Lisp o PDDL.
- Il *piano reattivo* (behavioral) si occupa invece delle azioni immediate, è formalizzato in maniera molto rigida ed ha un mapping preciso; si associa a linguaggi come C, C++ oppure Java.
- Il *piano interattivo* (interactive) si occupa invece del learning dal comportamento di altri agenti; si associa a linguaggi funzionali, procedurali oppure ontologici come OWL.

Nelle *architetture di sistema* vi sono tre paradigmi principali, descritti di seguito.

- **Gerarchico:** è un continuo loop di sense, plan, act, usato nei primissimi sistemi robotici, in una gerarchia che separa nettamente queste tre primitive. Questo paradigma risulta poco modulare e vi è difficoltà nell'inserire nuovi moduli, infatti è un paradigma orizzontale, che però non riesce ad offrire verticalità sulle varie primitive (esempio in figura 2).
  STRIPS è un esempio di paradigma gerarchico, utilizato nel primo robot, Shakey.



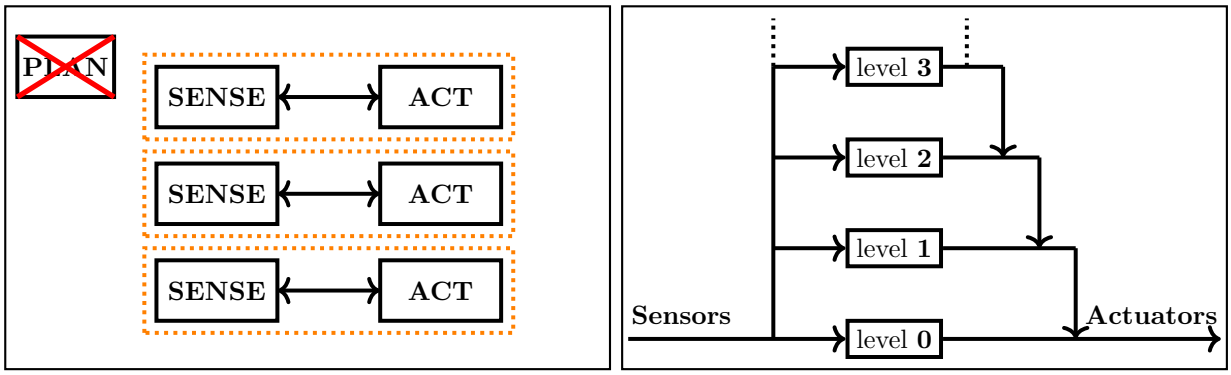
(a) Schema del paradigma gerarchico.



(b) Schema del paradigma gerarchico con moduli verticali.

Figure 2: Paradigma gerarchico, orizzontalità e verticalità.

- **Reattivo:** in questo caso, dalle tre primitive precedenti viene eliminato il planning, e rimane un modello con n moduli sense-act (tale modulo viene detto *behavior* o comportamento), ognuno dei quali ha un ruolo diverso e ben specificato. Risulta molto più modulare del paradigma gerarchico, in quanto per aggiungere funzionalità è sufficiente aggiungere un modulo. Qualora però condividano informazioni, risulta difficile lo scambio di tali informazioni, in quanto ogni comportamento è indipendente.
  L'architettura di Rodney Brooks, *subsumption*, vuole ovviare al problema effettuando raggruppamenti in layer di competenza che non hanno uno stato. In figura 3 abbiamo una rappresentazione del sia modello reattivo in generale (figura 3a), sia della versione di Brooks (figura 3b). Un esempio generico potrebbe essere un robot tagliaerba, avente due moduli: uno per evitare collisioni, ed uno per raggiungere un determinato punto nel giardino. L'architettura partirebbe dal livello 0 (evitare collisioni), poi aggiunge il livello 1 (possibilità di movimento) e successivamente il livello 2 (aggiunta di un goal nello spazio). Attraverso il paradigma reattivo, generalmente, si ottiene verticalità, in quanto possono essere eseguite diverse azioni in parallelo.



(a) Schema del Paradigma Reattivo.

(b) Schema della Architettura Subsumption.

Figure 3: Schema del Paradigma Reattivo e schema della Architettura Subsumption

- **Ibrido:** reintroduce la possibilità di effettuare planning insieme alla parallelizzazione dei vari behaviour per seguire il piano; essenzialmente il piano viene eseguito dal behavior finchè il task non è completo oppure deve essere cambiato. Attualmente è un modello flessibile ed anche il più utilizzato; è possibile aggiungere nuovi behaviors anche se non è semplice, vista la coordinazione necessaria tra i vari livelli. Lo schema è in figura 4.

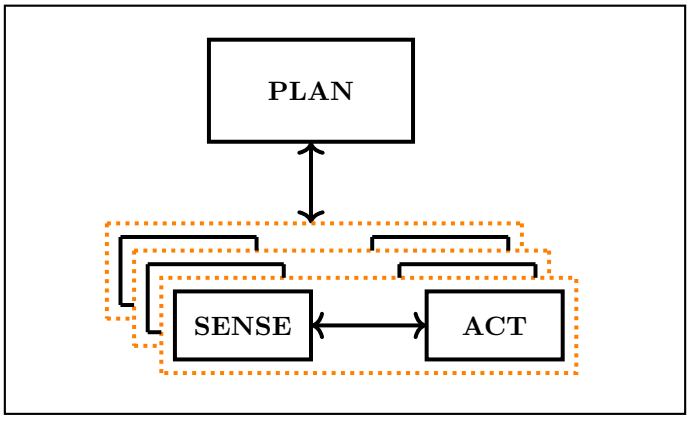


Figure 4: Schema del Paradigma Ibrido.

3 Locomozione dei Robot Mobili

La locomozione è l'abilità il movimento del robot, ed è una parte necessaria nel progetto del robot, in averne un modello permette di sapere come e dove ci si sta muovendo. I modelli in generale gestiscono un centro di curvatura istantaneo (ICC), cioè il punto attorno a cui ruota il robot, che si può calcolare come intersezione degli assi attorno a cui avviene il movimento; tale punto necessita di essere unico, affinchè la locomozione avvenga.

3.1 Tipi di Locomozione

Esistono differenti metodi di locomozione.

Differential Drive. Un robot a differential drive è dotato di due ruote motrici indipendenti (figura 5), una a sinistra e una a destra, montate su un asse fisso, ed il movimento del robot è determinato dalla velocità delle due ruote.

- Se entrambe girano alla stessa velocità, si ha un movimento rettilineo.
- Se una gira più veloce dell'altra, si ha un movimento curvilineo.
- Se girano in direzioni opposte, si ha una rotazione sul posto.

A partire dalle velocità delle ruote, v_r per la ruota di destra e v_s per la sinistra, e dalla distanza tra esse l, si può calcolare la velocità angolare di rotazione ω rispetto al centro ICC, ed il raggio di curvatura R, ovvero la distanza tra il centro del robot ed ICC, attraverso le formule:

$$R = \frac{l(v_r + v_l)}{2(v_r - v_l)}, \qquad \omega = \frac{v_r - v_l}{l}.$$

Notiamo dunque che, qualora $v_l = v_r$, il risultato sarebbe $R \to \infty$, ovvero ICC è inesistente in quanto non si sta ruotando, ma si sta eseguendo un movimento rettilineo. Al contrario, se avessimo $v_l = -v_r$, otterremmo $R = 0$, cioè un movimento di rotazione su se stesso. Successivamente, tramite ulteriori formule che utilizzano integrali, è possibile calcolare le coordinate (x, y, θ) per capire dove il robot si trovi nello spazio.
ICC rimane sempre sull'asse passante per le due ruote, inoltre spesso viene aggiunta una *custer wheel* libera di girare per aggiungere stabilità.

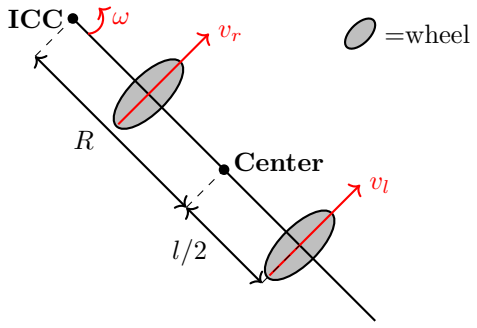


Figure 5: Schema del movimento differential drive.

Ackermann Drive. In questa locomozione si hanno due ruote sullo stesso asse ed una ruota sterzante anteriore. Sono utilizzate le stesse equazioni di Differential Drive, aggiungendo la variabile dell'angolo di sterzata sulla nuova ruota anteriore. Questo caso può essere esteso utilizzando due ruote, non parallele, aventi la perpendicolare che si intereseca in un solo punto, cioè ICC.

Synchronous Drive. Vi sono n ruote, solitamente 3, che girano su se stesse, tutte con lo stesso angolo, ed il robot si muove in qualsiasi direzione, traslando. Non può però ruotare sull'asse z.

XR4000 Drive. Vi sono quattro ruote indipendendti, che possono ruotare in maniera indipendente, dando la possibilità di curvare, traslare, ruotare. Naturalmente tutte le ruote puntano nel centro di curvatura istantaneo.

Mecanum Wheels. Modello con quattro ruote composte da rulli a 45° che consentono, con opportune velocità delle stesse, di effettuare non solo movimenti avanti/indietro ma anche destra/sinistra. Come dice il prof, "modello più esotico per fare derapate".

3.1.1 Vincoli non Olonomici

I vincoli non olonomici sono vincoli che limitano i movimenti nello spazio, ad esempio nei robot con differential drive o synchronous drive, contrariamente al XR4000 o al robot mecanum wheeled, che non hanno vincoli. Si differenziano dai vincoli olonomici, cioè vincoli che riducono lo spazio di configurazione (ad esempio le rotaie per i treni).

3.2 Problema della Localizzazione

Uno dei problemi principali nell'ambito della robotica è sapere il punto in cui un robot si trova durante il movimento; a questo scopo vengono utilizzati dei sensori.
Tali sensori possono essere di due tipi: attivi e passivi.

Attivi: emottono un segnale e poi ragionano sull'eco che ricevono di ritorno; un esempio sono i sensori time-of-flight ad ultrasuoni o laser. Per questa tipologia di sensori sorge però il problema per cui, dato un oggetto che deve riflettere il segnale, se esso è assente, se viene restituito un segnale non centrato rispetto al robot oppure si hanno interferenze, il robot non potrà ricevere segnali. Per ovviare a tale problematica,è necessario conoscere la dimensione massima di riferimento.
Infatti, essendo l'obiettivo del sensore quello di stimare la distanza d da un oggetto, conoscendo la velocità del segnale v ed il tempo t di ritorno del segnale ($d = \frac{vt}{2}$), se noi conosciamo la massima distanza, trascorso il tempo corrispondente, se non ritorna alcun segnale, ipotizziamo che non vi siano oggetti da rilevare.
Inoltre, se si considera che l'ampiezza massima di un sensore è di 15 gradi, va considerato che per scannerizzare l'area circostante servano altri 23 ($24 * 15 = 360$) sensori e dunque il tempo di tale scannerizzazione sarà molto maggiore, in quanto parallelizzando le misurazioni, è possibile ottenere un risultato che però è maggiormente affetto da interferenza (*crosstalk*).

Passivi: ragionano su un segnale esterno, ad esempio la luce ricevuta o da ragionano su un oggetto a contatto col sistema. Esempi sono i sensori tattili e le telecamere.

In generale, il probelma di localizzazione presenta due possibili risoluzioni, il dead reckoning (o odometria) e l'approccio map-based.

3.2.1 Dead Reckoning

Questa è una metodologia basata sul cammino percorso da un punto iniziale, continuamente aggiornato attraverso la direzione del robot, la velocità del robot ed il tempo trascorso; il problema principale è che l'errore in questa casistica è cumulativo.
Nell'ambito dell'odometria, possiamo immaginare dei raggi virtuali (dati per esempio dai sensori tipo laser) su una ruota, avente centro nel robot, che permettono di identificare posizione e velocità: questo modello può portare a grandi errori, ad esempio per un diverso diametro di ruota, slittamento su un tappetto o scossoni vari; tale errore sarà poi accumulato di volta in volta.

3.2.2 Map-Based Positioning

Nell'approccio map-based vi sono due tipi di approcci: può venire fornita all'agente una mappa (molto accurata ma adatta ad un solo ambiente), oppure può venire computata la mappa dall'agente (in questo

modo ci si adatta ad ogni ambiente ma è più difficile ed i risultati sono meno accurati).
Il problema principale di questa metodologia sta nel capire il posizionamento senza sapere dove si è: si può risolvere attraverso **SLAM** (Simultaneous Localization and Mapping, vedi capitolo 9), attuabile grazie a dei **landmark**, ovvero dei punti di interesse o oggetti nell'ambiente.

Per position tracking si intende trovare la posizione sapendo un punto di partenza, mentre per global localization si intende trovare la posizione senza conoscere nulla rispetto a dove si è partiti.

4 Ragionamento Probabilistico e Filtri di Bayes

Nell'ambito della robotica e dell'ambiente nel quale si muove il robot, è normale assumere incertezza e non determinismo, motivo per cui si utilizza la teoria della probabilità.

L'obiettivo è quello di stimare lo stato del sistema (**perception**) e di capire l'azione migliore da effettuare (**action**).

4.1 Basi di Teoria della Probabilità

Per fare questo, enunciamo innanzitutto gli assiomi della teoria della probabilità:

- $0 \leq P(A) \leq 1$ dato un evento A e la funzione di probabilità P.
- $P(\text{True}) = 1,\ P(\text{False}) = 0$.
- $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$.
- Data X una variabile aleatoria, considerando che essa può assumere diversi valori $x_1, x_2, \ldots, x_n$, si denota $P(X = x_i)$ o semplicemente $P(x_i)$ la probabilità che X assuma valore x_i.
- Nel caso di X variabile continua si ha una funzione di densità della probabilità: $P(X \in (a, b)) = \int_a^b p(x)dx$.

La probabilità inoltre può essere legata a più variabili casuali, ad esempio X e Y, e si scriverà:

$$P(X = x \wedge Y = y) = P(x, y).$$

Inoltre, se X e Y sono indipendenti:

$$P(x, y) = P(x)P(y).$$

La probabilità che accada l'evento x, supponendo che sia successo l'evento y si denota come:

$$P(x|y) = \frac{P(x, y)}{P(y)}.$$

Per cui:

$$P(x, y) = P(x|y)P(y),$$

da cui deriva che, se X ed Y sono indipendenti:

$$P(x|y) = P(x).$$

Inoltre si ha la legge di probabilità totale, che enunciamo nel caso discreto (non è scritto nelle slide ma si assume che la variabile X può assumere in insieme di valori contabile infinito, mutualmente esclusivo e di cui almeno un evento è certo che avverrà, dunque $X = \{x_1, x_2, \ldots\}$; non sono dettagli di poco conto, ma in quanto ingegnere $\pi = 3 = e$, quindi anche sti cazzi):

- $\sum_{x \in X} P(x) = 1$
- $P(y) = \sum_{x \in X} P(x, y)$ e dunque $P(y) = \sum_{x \in X} P(y|x)P(x)$ (**marginalizzazione**).

4.2 Ragionamento Bayesiano

Introduciamo qui il **teorema di Bayes**:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} = \frac{\text{likelihood} \cdot \text{prior}}{\text{evidence}} \qquad \text{con} \qquad P(x|y) = \text{posterior}$$

La formula introdotta si ricava facilmente dalle altre introdotte in precedenza, ma da qui si introduce l'idea di voler stimare una probabilità a partire da ciò che è in nostro possesso.

Noi intendiamo calcolare posterior, ovvero la probabilità di un'ipotesi (o di uno stato) una volta integrate le informazioni derivanti dalle osservazioni.
Likelihood indica quanto sia probabile osservare y (ad esempio una misura raccolta da un sensore) se l'ipotesi su x è vera, essa risulta utile in quanto si può ricavare attraverso il conteggio di frequenza (cioè posso contare quante volte ho misurato x essendo che si è verificato un evento y)
Prior invece in un contesto pratico, può essere basata su conoscenze pregresse, statistiche storiche o ipotesi iniziali.
Evidence infine è il termine di normalizzazione, ossia la probabilità complessiva di osservare y tenendo conto di tutte le possibili ipotesi. Tale evidenza può essere specificata attraverso il termine η, che viene spesso calcolato sommando o integrando $P(y|x)P(x)$ su tutte le ipotesi x; serve a garantire che la probabilità "posterior" rimanga una vera distribuzione di probabilità (ossia che sommi a 1) ed anche essa ricavata a partire da conteggi di frequenza e prior:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} = \eta P(y|x)P(y)$$
$$\text{con} \quad \eta = P(y)^{-1} = \frac{1}{\sum_x P(y|x)P(x)}$$

Inoltre, la regola di Bayes è applicabile anche ad n variabili, ad esempio x, y, z:

$$P(x|y,z) = \frac{P(y|x,z)P(x|z)}{P(y|z)}$$

4.3 State Estimation

Supponiamo ora un caso pratico di stima dello stato di un robot: il robot riceve il valore z da un sensore e deve scegliere se aprire una porta, qual'è la probabilità $P(open|z)$?
Innanzitutto vediamo che a livello teorico, data una misurazione z ed una causa, abbiamo due tipi di ragionamenti:

- $P(z|causa)$ è **causale**, in genere più sempilce da stimare in quanto può effettuare un conteggio di frequenza (nel caso del robot, quante volte ho misurato z, sapendo che la porta era aperta).
- $P(causa|z)$ è **diagnostico**, cioè data una misurazione, si vuole capire qual'è la probabilità che un dato evento sia vero, nel caso del robot, il fatto che la porta sia aperta.

Dalla teoria precedente, è evidente che $P(open|z) = \frac{P(z|open)P(open)}{P(z)}$.
In generale, un robot però effettua n misurazioni, dunque sorge un nuovo problema: vogliamo calcolare $P(x|z_1, \ldots, z_n)$.

4.4 Recursive Bayesian Updating

Possiamo derivare che:

$$P(x|z_1, \ldots, z_n) = \frac{P(z_n|x, z_1, \ldots, z_{n-1})P(x|z_1, \ldots, z_{n-1})}{P(z_n|z_1, \ldots, z_{n-1})}.$$

Per proseguire nel calcolo, utilizziamo **l'assunzione di Markov**: z_n è indipendente da $z_1, \ldots, z_{n-1}$ se noi conosciamo x. Naturalmente è una assunzione che si compie e che potrebbe non essere accettata, ma risulta necessaria per semplificare problemi ad ampio spettro nell'ambito della robotica. La ragione intuitiva dietro questa ipotesi è data dal fatto che, conoscendo lo stato attuale x e le misurazioni $z_1, \ldots, z_{n-1}$, è possibile ricavare l'errore di tali misurazioni, non influenzando z_n, vista appunto la conoscenza dello stato.

Ne consegue che $P(z_n|x, z_1, \ldots, z_{n-1}) = P(z_n|x)$ e dunque:

$$\begin{aligned} P(x|z_1, \ldots, z_n) &= \frac{P(z_n|x)P(x|z_1, \ldots, z_{n-1})}{P(z_n|z_1, \ldots, z_{n-1})} \\ &= \eta_n P(z_n|x)P(x|z_1, \ldots, z_{n-1}) \\ &= \eta_{n-1}\eta_n P(z_{n-1}|x)P(z_n|x)P(x|z_1, \ldots, z_{n-2}) \\ &= \eta_{1\ldots n} \prod_{i=1\ldots n} P(z_i|x)P(x) \end{aligned}$$

Questa formula, attraverso il continuo raccoglimento di evidenze, può essere aggiornata.

4.5 Azioni nel Mondo

Un ulteriore fattore da considerare è la scena dinamica: le azioni del robot, le azioni di altri agenti ed il passare del tempo cambiano il mondo.
Ogni azione effettuata porta infatti incertezza, dunque ad ogni azione vi è un aumento del grado di tale incertezza, al contrario di ciò che avviene attraverso le misurazioni.
Per rappresentare un'azione e la probabilità associata, si ha che, dato uno stato attuale x', l'azione u e lo stato successivo x, la probabilità di raggiungere x dallo stato x' con l'azione u è definita come $P(x|u, x')$.
Ulteriormente, utilizzando la marginalizzazione, si ha che:

$$P(x|u) = \sum_{x'} P(x|u, x')P(x')$$

4.6 Filtri di Bayes

Si delinea dunque il framework per definire i filtri di Bayes, per cui a partire dai dati:

- **una serie di dati di osservazione** z e di **dati di azioni** u al tempo t: $d_t = \{u_1, z_1, \ldots, u_t, z_t\}$;
- **un modello dei sensori**: $P(z|x)$ sullo stato del sistema x;
- **un modello delle azioni**: $P(x|u, x')$ sullo stato del sistema x a partire dallo stato x';
- **prior** la probabilità dello stato del sistema $P(x)$.

Si vuole stimare lo stato x di un sistema dinamico e la probabilità posterior dello stato, definita come **belief** (essenzialmente la probabilità di essere in uno stato x_t data la mia conoscenza pregressa del mondo):

$$Bel(x_t) = P(x_t|u_1, z_1, \ldots, u_t, z_t)$$

Per calcolare questa probabilità, effettuiamo due assunzioni, dette **assunzioni di Markov**.

- La misurazione effettuata al tempo t dipende solo dallo stato del sistema al tempo t, cioè x_t:

$$P(z_t|x_{0\ldots t}, z_{1\ldots t}, u_{1\ldots t}) = P(z_t|x_t)$$

- Lo stato del sistema al tempo t, x_t, dipende unicamente dallo stato precendente del sistema x_{t-1} e dall'azione al tempo t:

$$P(x_t|x_{1\ldots t-1}, z_{1\ldots t}, u_{1\ldots t}) = P(x_t|x_{t-1}, u_t)$$

Tali assunzioni possono intuitivamente essere rappresentate dallo schema in figura 6.
Altre assunzioni inoltre sono sottintese: un mondo statico, indipendenza dal rumore e perfezione del modello. Viste le ipotesi elencate, possiamo ricavare $Bel(x_t)$.

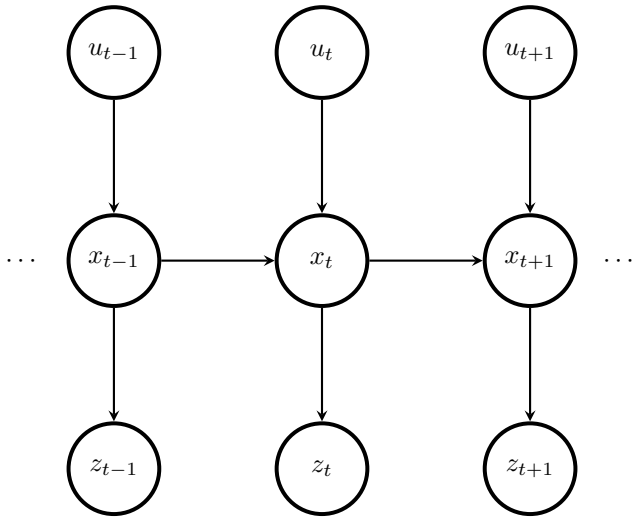


Figure 6: Schema delle assunzioni di Markov

$$
\begin{aligned}
\boldsymbol{Bel(x_t)} &= P(x_t|u_1, z_1, \ldots, u_t, z_t) \\
\text{(Bayes)} &= \eta P(z_t|x_t, u_1, z_1, \ldots, z_{t-1}, u_t) P(x_t|u_1, z_1, \ldots, z_{t-1}, u_t) \\
\text{(Markov)} &= \eta P(z_t|x_t) P(x_t|u_1, z_1, \ldots, z_{t-1}, u_t) \\
\text{(Total Probability)} &= \eta P(z_t|x_t) \int P(x_t|u_1, z_1, \ldots, z_{t-1}, u_t, x_{t-1}) P(x_{t-1}|u_1, z_1, \ldots, z_{t-1}, u_t) dx_{t-1} \\
\text{(Markov)} &= \eta P(z_t|x_t) \int P(x_t|u_t, x_{t-1}) P(x_{t-1}|u_1, z_1, \ldots, z_{t-1}, u_t) dx_{t-1} \\
\text{(Markov)} &= \eta P(z_t|x_t) \int P(x_t|u_t, x_{t-1}) P(x_{t-1}|u_1, z_1, \ldots, u_{t-1}, z_{t-1}) dx_{t-1} \\
&= \eta \underbrace{P(z_t|x_t)}_{\text{Correzione}} \int \underbrace{P(x_t|u_t, x_{t-1})}_{\text{Predizione}} \underbrace{Bel(x_{t-1})}_{\text{Ricorsione}} dx_{t-1}
\end{aligned}
\tag{1}
$$

In forma discreta si avrà invece:

$$
Bel(x_t) = \eta P(z_t|x_t) \sum_{x_{t-1}} P(x_t|u_t, x_{t-1}) Bel(x_{t-1}) \tag{2}
$$

Abbiamo applicato una serie di passaggi data dai teoremi e dalle assunzioni di partenza; in particolare abbiamo una equazione ricorsiva nella quale la parte di correzione è data dal **modello dei sensori** e la parte di predizione è data dal **modello delle azioni**.

5 Modelli di Sensori e Azioni

In questa sezione ci occupiamo di provare ad implementare un filtro Bayesiano, specificando i modelli di azione ed i modelli dei sensori. Questi modelli sono basati sul non determinismo del robot, e dunque si occupano di dare una probabilità di incertezza che accompagna gli stati e le azioni del modello.

5.1 Modelli di Azione

I modelli di azioni vogliono restituire il valore $P(x_t, |u_t, x_{t-1})$ nel filtro di Bayes, cioè la probabilità "posterior" che dato uno stato x_{t-1} ed una azione u, noi raggiungiamo effettivamente lo stato x_t; da notare che vi è una discretizzazione temporale sottintesa.
I modelli utilizzano sei dimensioni maggiori, ovvero le coordinate cartesiane nello spazio tridimensionale e le tre angolature di Eulero; ma qui per semplicità vi è una riduzione a tre dimensioni, una superficie bidimensionale ed una angolatura su tale piano: (x, y, θ).
Le principali tipologie di modelli di azione sono due:

- **Odometry based**: basati su sensori odometrici (*wheel encoders*), attraverso i quali si effettua una stima dello spostamento avvenuto.
- **Velocity based** (dead reckoning): basati sugli input interni, come la velocità, la coppia e la velocità angolare attuale del robot, di cui però non si ha un risultato riguardante lo spostamento effettivo (da qui il non determinismo citato all'inizio della sezione).

Per entrambi i casi, è necessario precisare una importante assunzione sottostante a tutte le procedure: di fatto noi conosciamo lo stato in cui siamo quando stimiamo il modello (x_t). Infatti, per quanto controintuitivo, va ricordato che noi stiamo cercando di stimare la probabilità $P(x_t, |u_t, x_{t-1})$, ovvero la probabilità, dati i due stati x_t, x_{t-1} e l'azione u_t, di arrivare effettivamente allo stato x_t, così da poter poi sfruttare tale valore probabilistico nella pratica.

Inoltre, viene considerato un intervallo temporale generalmente piccolo, al fine di cercare di ridurre l'errore totale.

5.1.1 Modelli Odometry-based

Nei modelli odometry-based si ha che un movimento è articolato in tre fasi: una rotazione iniziale, una traslazione ed una rotazione finale; tutti questi movimenti sono movimenti di piccola entità, così da minimizzare l'errore, nel lasso di tempo breve $(t-1, t]$.
Nello specifico, un robot si muove da una posa, ovvero il punto iniziale $x_{t-1} = (x, y, \theta)$ ad un'altra posa, il punto finale $x_t = (x', y', \theta')$ noti, come spiegato dall'assunzione iniziale. Il robot, al contrario, utilizza informazioni odometriche, le quali riportano informazioni sull'avanzamento da $\bar{x}_{t-1} = (\bar{x}, \bar{y}, \bar{\theta})$ a $\bar{x}_t = (\bar{x}', \bar{y}', \bar{\theta}')$. L'informazione odometrica u_t è data dalla coppia $(\bar{x}_{t-1}, \bar{x}_t)$.

Da questo punto di partenza è possibile ricavare le informazioni odometriche sulle tre fasi del movimento (uno schema è visibile in figura 7), che risultano dunque le misure odometriche, ovvero quelle rilevate dal robot.

- $\delta_{trans} = \sqrt{(\bar{x}' - \bar{x})^2 + (\bar{y}' - \bar{y})^2}$
- $\delta_{rot1} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
- $\delta_{rot2} = \bar{\theta}' - \bar{\theta} - \delta_{rot1}$

Naturalmente, il movimento misurato attraverso i sensori, risulta differente dal movimento effettivo dallo stato x_{t-1} a x_t; tale discostamento si può descrivere come rumore, descrivibile sottoforma di gaussiana (poi approssimata a distribuzione triangolare per motivi di semplicità computazionale).

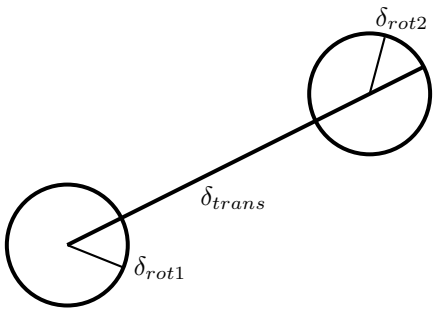


Figure 7: Modello odometrico.

La somma del valore δ e del rumore, detto ϵ, porta a $\hat{\delta}$, che definiamo il valore "vero". Questo valore si ottiene attraverso le formule:

- $\hat{\delta}_{rot1} = \delta_{rot1} + \epsilon_{\alpha_1|\delta_{rot1}|+\alpha_2|\delta_{trans}|}$
- $\hat{\delta}_{trans} = \delta_{trans} + \epsilon_{\alpha_3|\delta_{trans}|+\alpha_4|\delta_{rot1}+\delta_{rot2}|}$
- $\hat{\delta}_{rot2} = \delta_{rot2} + \epsilon_{\alpha_1|\delta_{rot2}|+\alpha_2|\delta_{trans}|}$

I valori $\alpha_{1\ldots4}$ sono pesi che necessitano di essere trovati, ma dei quali non ci occuperemo per adesso. Naturalmente, essendo i valori $\hat{\delta}$ corretti per ipotesi, si ha che conoscendi i valori reali di x_{t-1} ed x_t si possono ottenere gli stessi valori utilizzando le formule odometriche precedenti.
L'algoritmo per ottenere la probabilità a posteriori di uno stato x_t è dunque il seguente (algoritmo 1):

Algorithm 1 motion_model_odometry(x_t, u_t, x_{t-1})

1: $\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
2: $\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^s}$
3: $\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

4: $\hat{\delta}_{\text{rot1}} = \text{atan2}(y' - y, x' - x) - \theta$
5: $\hat{\delta}_{\text{trans}} = \sqrt{(x - x')^2 + (y - y')^2}$
6: $\hat{\delta}_{\text{rot2}} = \theta' - \theta - \hat{\delta}_{\text{rot1}}$

7: $p_1 = \text{prob}(\delta_{\text{rot1}} - \hat{\delta}_{\text{rot1}}, \alpha_1\hat{\delta}_{\text{rot1}} + \alpha_2\hat{\delta}_{\text{trans}})$
8: $p_2 = \text{prob}(\delta_{\text{trans}} - \hat{\delta}_{\text{trans}}, \alpha_3\hat{\delta}_{\text{trans}} + \alpha_4(\hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}))$
9: $p_3 = \text{prob}(\delta_{\text{rot2}} - \hat{\delta}_{\text{rot2}}, \alpha_1\hat{\delta}_{\text{rot2}} + \alpha_2\hat{\delta}_{\text{trans}})$

10: **return** $p_1 \cdot p_2 \cdot p_3$

Nelle prime tre righe otteniamo i valori odometrici u, mentre nelle tre successive utilizziamo le stesse formule con i dati veritieri, ottenendo dunque i valori di δ reali, $\hat{\delta}$. Le probabilità dei tre valori infine sono date dalla probabilità nella distribuzione normale del discostamento tra i due valori calcolati in precedenza; dando per ipotesi che p_1, p_2, p_3 siano indipendenti, ritorniamo il loro prodotto.
Questo metodo ci permette di trovare la probabilità di un dato stato x_t a partire dallo stato noto x_{t-1} e dai valori odometrici u_t, che ricordiamo essere la coppia $(\bar{x}_{t-1}, \bar{x}_t)$.

Qualora dovessimo però voler calcolare quale sia lo stato successivo x_t, a partire dato lo stato precedente x_{t-1} e dalle informazioni odometriche u_t, l'algoritmo precedente risulta poco efficace in quanto il

calcolo dovrebbe essere effettuato per ogni x_t possibile. Il passo successivo è dunque trovare un algoritmo ideale a questo scopo.

Per il calcolo della probabilità è necessario effettuare sampling: per distribuzioni note come la normale o la triangolare è possibile sfruttare il teorema del limite centrale per approssimarle. Altrimenti, possiamo effettuare **rejection sampling**: si prende un sample x a partire da una distribuzione uniforme su $[-b, b]$, dopodiché si utilizza un sample y in una distribuzione uniforme $[0, max(f)]$, se $y > f(x)$ si scarta il sample.

Potendo sfruttare le distribuzioni, ed in particolare quella gaussiana (**sample** nell'algoritmo 2), l'idea a questo punto è quella di partire dai valori ricavabili dai sensori odometrici δ, e attraverso le suddette distribuzioni arrivare al valore statisticamente veritiero $\hat{\delta}$.
Attraverso tali valori, si arriva infine all'output desiderato, cioè alla posa di arrivo, $x_t = (x', y', \theta')^T$ (algoritmo 2).

Algorithm 2 sample_motion_model_odometry(u_t, x_{t-1})

$\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
$\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$
$\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

$\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta_{\text{rot1}} + \alpha_2 \delta_{\text{trans}})$
$\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3 \delta_{\text{trans}} + \alpha_4 (\delta_{\text{rot1}} + \delta_{\text{rot2}}))$
$\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta_{\text{rot2}} + \alpha_2 \delta_{\text{trans}})$

$x' = x + \hat{\delta}_{\text{trans}} \cos(\theta + \hat{\delta}_{\text{rot1}})$
$y' = y + \hat{\delta}_{\text{trans}} \sin(\theta + \hat{\delta}_{\text{rot1}})$
$\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

return $x_t = (x', y', \theta')^T$

5.1.2 Modelli Velocity-based

Nei modelli velocity-based il movimento è descritto da due parametri principali: una velocità traslazionale al tempo t, v_t ed una velocità rotazionale al tempo t, ω_t, per cui una azione u_t è data dalla coppia (v_t, ω_t). L'idea è di sviluppare due algoritmi analoghi a quelli precedenti, uno per trovare la probabilità "posterior" di uno stato $P(x_t, |u_t, x_{t-1})$, dati x_t, u_t ed x_{t-1}, ed uno per trovare lo stato x_t più probabile a partire da u_t ed x_{t-1}. Innanzitutto osserviamo alcuni dati attraverso il modello di movimento in figura 8.

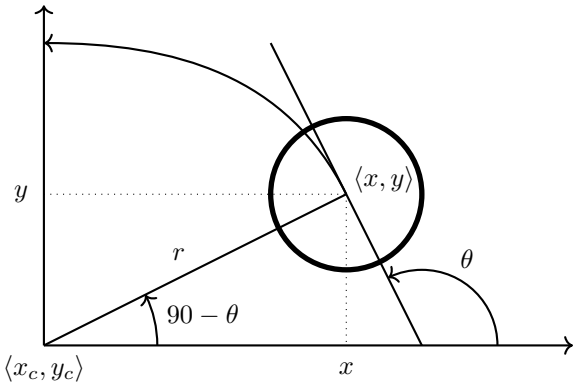


Figure 8: Modello velocity-based.

Abbiamo il centro di curvatura istantaneo, ICC, nei punti $\langle x_c, y_c \rangle$, con raggio r ed angolatura θ; il robot ha centro nel punto $\langle x, y \rangle$, durante l'azione u_t. Parallelamente ai modelli odometrici, anche qui abbiamo dei dati "veritieri", dati dalla velocità e dalla rotazione effettiva (conosciuti al robot in quanto è lui stesso ad imprimeli) e da un rumore ϵ.

- $\hat{v} = v + \epsilon_{\alpha_1|v|+\alpha_2|\omega|}$
- $\hat{\omega} = \omega + \epsilon_{\alpha_3|v|+\alpha_4|\omega|}$

A cui si aggiunge successivamente un terzo termine $\hat{\gamma}$, atto a considerare la rotazione finale (senza di essa si avrebbe una probabilità pressochè nulla di avere una rotazione corretta) e collegato al rumore:

- $\hat{\gamma} = \epsilon_{\alpha_5|v|+\alpha_6|\omega|}$

Utilizzando questi valori, si vuole poi calcolare la posa del robot:

- $x' = x - \frac{\hat{v}}{\hat{\omega}} \sin\theta + \frac{\hat{v}}{\hat{\omega}} \sin(\theta + \hat{\omega}\Delta t)$
- $y' = y + \frac{\hat{v}}{\hat{\omega}} \cos\theta - \frac{\hat{v}}{\hat{\omega}} \cos(\theta + \hat{\omega}\Delta t)$
- $\theta' = \theta + \hat{\omega}\Delta t + \hat{\gamma}\Delta t$

Chiamando ora le coordinate del centro di rotazione $\begin{pmatrix} x^* & y^* \end{pmatrix}^T$, ne definiamo le equazioni utilizzando due incognite $\lambda, \mu \in \mathbb{R}$.

$$\begin{pmatrix} x^* \\ y^* \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} -\lambda \sin\theta \\ \lambda \cos\theta \end{pmatrix} = \begin{pmatrix} \frac{x+x'}{2} + \mu(y - y') \\ \frac{y+y'}{2} + \mu(x' - x) \end{pmatrix}$$

Da cui, attraverso alcuni passaggi matematici, si può arrivare ad esprimere il tutto in funzione di μ:

$$\mu = \frac{1}{2} \frac{(x - x')\cos\theta + (y - y')\sin\theta}{(y - y')\cos\theta - (x - x')\sin\theta}$$

Ne consegue che, conoscendo i dati in input all'algoritmo (x_t, u_t ed x_{t-1}), posso ricavare μ, da cui ottengo poi le coordinate del centro di rotazione ed il raggio; da qui è facile ottenere i valori "veri" $\hat{v}, \hat{\omega}, \hat{\gamma}$. Il processo è descritto dall'algoritmo 3.

Algorithm 3 motion_model_velocity (x_t, u_t, x_{t-1})

1: $\mu = \frac{1}{2} \frac{(x-x')\cos\theta + (y-y')\sin\theta}{(y-y')\cos\theta - (x-x')\sin\theta}$
2: $x^* = \frac{x+x'}{2} + \mu(y - y')$
3: $y^* = \frac{y+y'}{2} + \mu(x' - x)$
4: $r^* = \sqrt{(x - x^*)^2 + (y - y^*)^2}$
5: $\Delta\theta = \text{atan2}(y' - y^*, x' - x^*) - \text{atan2}(y - y^*, x - x^*)$
6: $\hat{v} = \frac{\Delta\theta}{\Delta t} r^*$
7: $\hat{\omega} = \frac{\Delta\theta}{\Delta t}$
8: $\hat{\gamma} = \frac{\theta' - \theta}{\Delta t} - \hat{\omega}$
9: **return** $\text{prob}(v - \hat{v}, \alpha_1|v| + \alpha_2|\omega|) \cdot \text{prob}(\omega - \hat{\omega}, \alpha_3|v| + \alpha_4|\omega|) \cdot \text{prob}(\hat{\gamma}, \alpha_5|v| + \alpha_6|\omega|)$

Analogamente a quanto fatto per i modelli odometrici, è possibile utilizzare distribuzione di probabilità (**sample**) per predire lo stato x_t a partire da x_{t-1} ed u_t (algoritmo 4):

Algorithm 4 sample_motion_model_velocity (u_t, x_{t-1})

1: $\hat{v} = v + \textbf{sample}(\alpha_1|v| + \alpha_2|\omega|)$
2: $\hat{\omega} = \omega + \textbf{sample}(\alpha_3|v| + \alpha_4|\omega|)$
3: $\hat{\gamma} = \textbf{sample}(\alpha_5|v| + \alpha_6|\omega|)$
4: $x' = x - \frac{\hat{v}}{\hat{\omega}} \sin\theta + \frac{\hat{v}}{\hat{\omega}} \sin(\theta + \hat{\omega}\Delta t)$
5: $y' = y + \frac{\hat{v}}{\hat{\omega}} \cos\theta - \frac{\hat{v}}{\hat{\omega}} \cos(\theta + \hat{\omega}\Delta t)$
6: $\theta' = \theta + \hat{\omega}\Delta t + \hat{\gamma}\Delta t$
7: **return** $x_t = (x', y', \theta')^T$

Un ulteriore tipo di modello sono i map consistent motion model, in cui per rendere più accurate le stime, indipendentemente dal modello utilizzato, posso aggiungere la conoscenza della mappa per verificare non ci siano collisioni (per cui si passa da $p(x|u, x')$ a $p(x|u, x', m)$).

In sintesi, in entrambi i casi (odometrico, basato sulla velocità), possiamo calcolare la probabilità "posterior", ovvero la probabilità che un determinato stato sia stato raggiunto, oppure fare sampling per capire quale sarà lo stato raggiunto a partire dallo stato precedente e dall'azione compiuta. I parametri delle distribuzioni di probabilità devono essere trovati empiricamente (solitamente attraverso una fase di learning precedente).

5.2 Modelli di Sensori

Lo scopo dei modelli di sensori è restituire il valore $P(z_t, |x_t)$ nel filtro di Bayes, cioè la probabilità della correttezza della misura, ovvero la correzione, a partire da uno stato noto x_t. Esistendo vari sensori, (ad esempio di prossimità, GPS, camere ecc.), la classificazione dei modelli di sensori avviene basandoci su due distinzioni:

- **beam-based models**: ogni scan è composta da diverse misurazioni nelle varie direzioni, assunte indipendenti ignorando il problema del *cross-talk*;
- **scan-based models**: parte dall'idea di non seguire la lunghezza del raggio ma controllare solo il suo punto finale.

In generale, come per i modelli di azione, adoperiamo delle assunzioni: conosciamo sia la misurazione z_t, che è naturalmente nota al robot, essendo stata effettuata da esso stesso, sia lo stato x_t e dunque la sua posizione, così da poter appunto stimare la probabilità obiettivo del sensor model; inoltre è nota una mappa dell'ambiente, m.

5.2.1 Modelli Beam-based

Data una scan z, essa si compone di K misurazioni per cui $z = \{z_1, z_2, \ldots, z_K\}$, le quali sono considerate indipendenti, motivo per cui si ha che:

$$P(z|x, m) = \prod_{k=1}^{K} P(z_k|x, m)$$

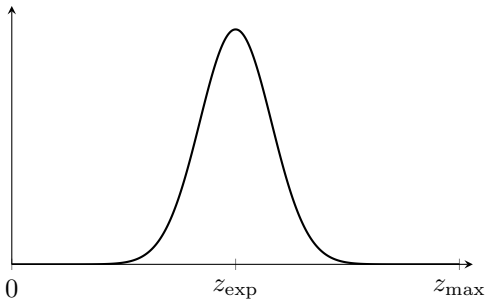
In realtà, vi sono quattro principali fonti di errore in queste misurazioni, che noi dobbiamo prevedere affinchè si possa effettuare la correzione, e dunque la probabilità (ognuna di esse ha una formula di probabilità oltre che un grafico, ma per brevità non viene riportata).

Rumore nelle misurazioni (P_{hit}) le misurazioni posssono avere minime variazioni dovute al rumore. Ad esempio dato uno stato x, la posizione del robot è distante z_{exp} sulla direzione di uno scan rispetto da un muro, ne consegue che la probabilità di trovarsi effettivamente a distanza z_{exp} data una misurazione generica z è data da una distribuzione normale attorno a z_{exp} (figura 9a).

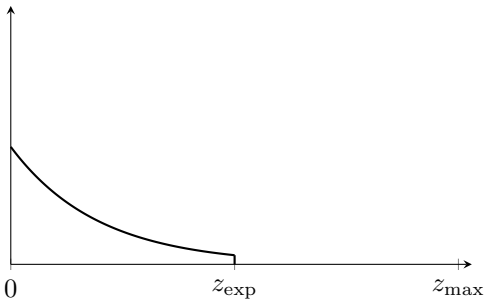
Raggi riflessi da ostacoli (P_{unexp}) può accadere che ad esempio in una stanza, se una persona cammina e dunque interrompe il raggio, la misuazione rispetto al muro nella mappa non sia corretta. Dall'idea di una distribuzione uniforme degli ostacoli ne deriva che fino alla distanza reale z_{exp}, si avrà una probabilità sempre minore di incontrare un ostacolo, e successivamente sarà ovviamente nulla (figura 9b).

Misurazioni casuali (P_{rand}) in questo caso si vuole modellare la probabilità di ottenere una misurazione casuale, che risulta una probabilità uniforme (figura 9c).

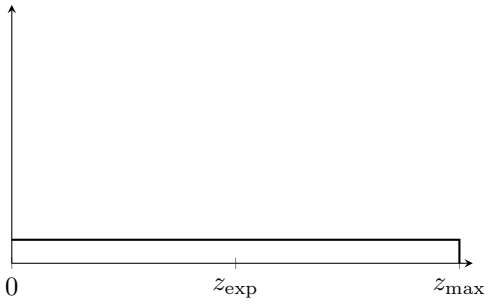
Max range (P_{max}) si vuole modellare la probabilità di un errore dato da un falso max range, che dunque sarà una alta probabilità attorno al max range z_{max} (figura 9d).



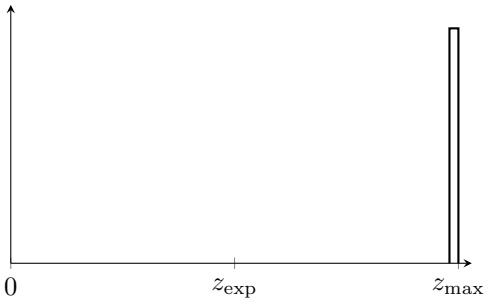
(a) Distribuzione del rumore nelle misurazioni.



(b) Distribuzione di probabilità degli ostacoli inaspettati nelle misurazioni.



(c) Distribuzione della probabilità delle misurazioni casuali.



(d) Distribuzione della probabilità di falso max range.

Figure 9: Distribuzione di probabilità delle problematiche nelle misurazioni beam-based.

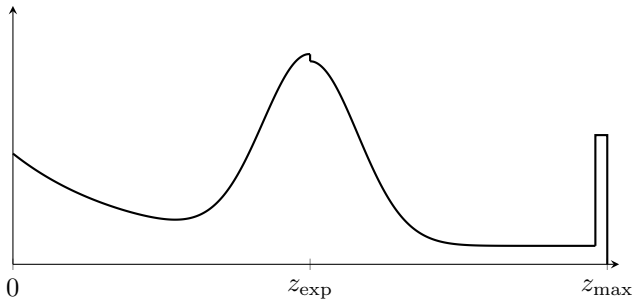


Figure 10: Distribuzione di probabilità delle problematiche di errore nei modelli di sensori beam-based.

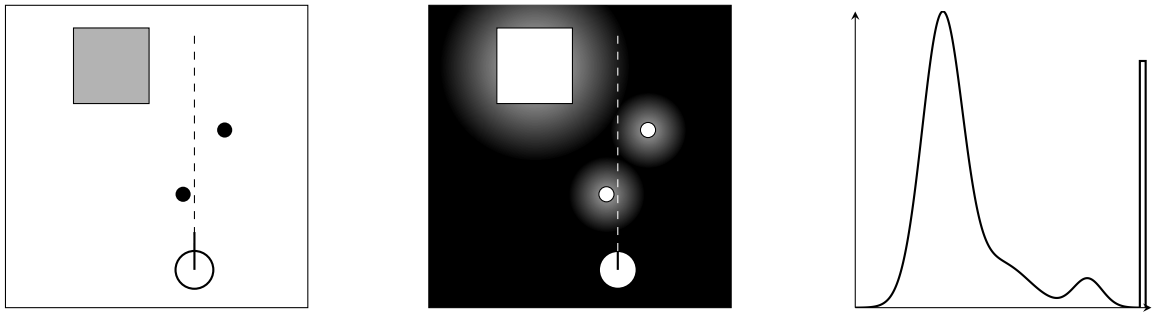
Il risultato della considerazione di questi quattro fattori è una distribuzione di probabilità simile a quella in figura 10, che si ricava da una combinazione lineare delle quattro probabilità con 4 pesi $(\alpha_{hit}, \alpha_{unexp}, \alpha_{rand}, \alpha_{max})$. Questi quattro valori si ricavano in generale da ricerche nello spazio dei parametri attraverso tecniche di machine learning (ad esempio hill-climbing, gradient-descent ecc.).
Il problema di questa modellazione è che è overconfident dato che considera misure indipendenti, non è "smooth" per piccoli ostacoli e bordi (si hanno discese improvvise nella distribuzione sui bordi, in quando si passa immediatamente da valori magari piccoli a grandi, portando a distribuzioni con cambi repentini e difficilmente gestibili da alcuni algoritmi) ed è poco efficiente (si finisce ad avere a che fare con problemi di grandezza esponenziale attraverso i parametri e le angolazioni). Un'idea per risolvere questo problema è seguire il raggio e verificare il punto finale.

5.2.2 Modelli Scan-based

In questi modelli si ha che si considera solamente il punto finale e non il raggio, le misurazioni sono sempre considerate indipendenti l'una con l'altra, e la probabilità è data da:

- una distribuzione normale avente come punto medio la distanza dall'ostacolo più vicino (z_{exp} nei modelli beam-based),
- una distribuzione uniforme per le misurazioni casuali,
- una piccola distribuzione uniforme per le misurazioni per le misurazioni della portata massima.

Essenzialmente abbiamo rimosso il fattore degli ostacoli casuali dal caso precedente. Inoltre utilizza una mappa per associare la distanza dall'ostacolo più vicino (esempio in figura 11). Questa metodologia è utile in quanto non fa ray-casting, è più "smooth" del modello beam-based e dunque più ideale ad approcci gradient descent, maggiormente efficienti.



(a) Map m. (b) Likelihood field. (c) $P(z \mid x, m)$.

Figure 11: Distribuzione di probabilità (likelihood field) data una mappa m attraverso il modello scan-based.

Landmarks Vi è inoltre la possibilità di utilizzare dei *landmark* nella mappa (beacon attivi come radio o GPS, oppure becon passivi come sensori visuali o retroriflessivi), con dei sensori che restituiscono la distanza e la direzione da tali landmark. Conoscendo la distanza di almeno tre landmark è possibile effettuare triangolazione, da cui poi si effettua il calcolo probabilistico della posizione (sempre considerando rumore ecc.)

6 Filtri Discreti e Particellari

Definiti il sensor model e il motion model, si vuole applicare il filtro di Bayes a livello pratico. Per farlo vedremo due filtri non parametrici, il filtro di Bayes discreto ed il filtro particellare.

6.1 Filtro di Bayes Discreto

Il filtro di Bayes discreto effettua una discretizzazione del tempo, come già noto, ed anche dello spazio. Questo passaggio matematico si traduce nel passaggio già visto dall'equazione 1 all'equazione 2.
L'equazione 2 dunque si traduce nell'algoritmo 5.

Algorithm 5 Discrete_Bayes_filter $(Bel(x), d)$

1: $\eta = 0$
2: **if** d is a **perceptual** data item z **then**
3: **for** all x **do**
4: $Bel'(x) = P(z \mid x)Bel(x)$
5: $\eta = \eta + Bel'(x)$
6: **end for**
7: **for** all x **do**
8: $Bel'(x) = \eta^{-1}Bel'(x)$
9: **end for**
10: **else if** d is an **action** data item u **then**
11: **for** all x **do**
12: $Bel'(x) = \sum_{x'} P(x \mid u, x')Bel(x')$
13: **end for**
14: **end if**
15: **return** $Bel'(x)$

Questo approccio è quello più intuitivo, corrispondente ad un *brute force*, che però risulta poco efficiente, visto soprattutto la difficoltà quadratica dell'operazione a riga 12, considerando anche le tre o sei dimensioni di operabilità, per cui i possibili stati possono arrivare ad essere miliardi.
Da questo presupposto si cerca di migliorare l'algoritmo; ad esempio considerando che gran parte degli stati ha probabilità nulla o quasi nulla, si possono sfruttare le proprietà delle matrici sparse, effettuando calcoli computazionali solo sugli stati maggiormente probabili. Lo svantaggio di questo approccio è che nel caso in cui il robot venga spostato in una posizione di cui non si sta tenendo traccia è necessario ricostruire da zero il suo belief.
Alternativamente è possibile ridurre la complessità utilizzando una struttura ad albero che permetta di rappresentare lo spazio degli stati con granularità variabile, approfondendo meglio gli stati più probabili e aggregando gli stati più improbabili in uno unico, similmente a quanto avviene con algoritmi di computer vision.

6.2 Filtro Particellare

L'idea principale alla base del filtro particellare è quella rappresentare i belief state come un insieme di particelle, utilizzando dei *random sample* pescati attraverso la probabilità "posterior"; ogni particella corrisponde ad una ipotesi ed ha un peso proporzionale alla probabilità. Si parte da una distribuzione iniziale, e successivamente vengono aggiornati pesi e distribuzione utilizzando il motion model e il sensor model. Un esempio intuitivo è visibile in figura 12: i sample (punti rossi) si aggiornano continuamente e si accentrano sulla posizione reale attraverso le stime del motion model e delle misurazioni.

Matematicamente, l'insieme di partenza è dato dall'insieme pesato delle ipotesi al tempo $t-1$, ovvero:

$$\mathcal{X}_{t-1} = \{x_{t-1}^{[i]} = \langle s_{t-1}^i, w_{t-1}^i \rangle | i = 1, \ldots, M\}.$$

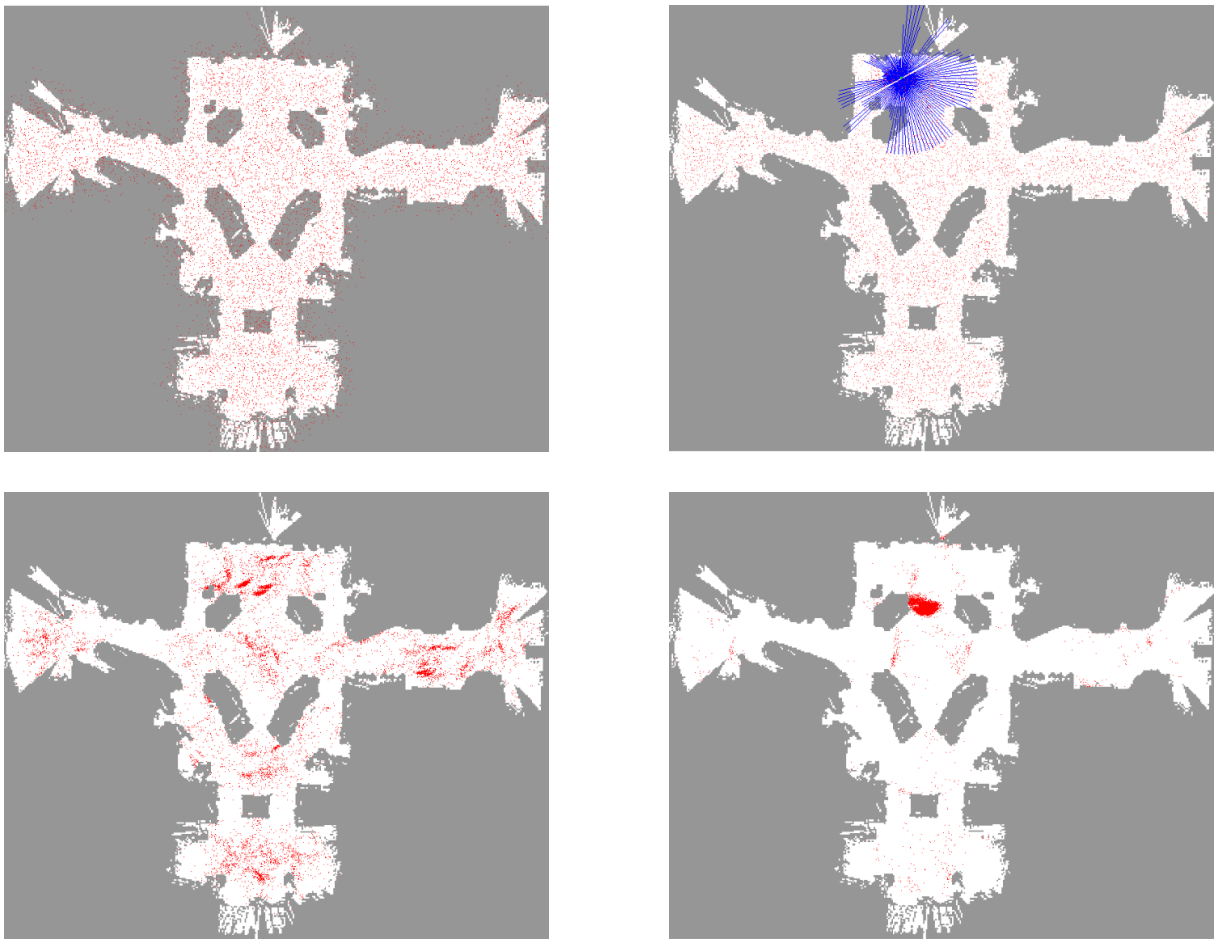


Figure 12: Esempio di applicazione del filtro particellare all'interno di un museo.

Il peso associato idealmente riflette il belief state, infatti ogni ipotesi è direttamente proporzionale al belief state:

$$x_t^{[i]} \sim Bel(x_t) = p(x_t|z_{1:t}, u_{1:t}).$$

Nell'algoritmo vengono utilizzate diverse tecniche, spiegate nelle sezioni successive così da avere una più chiara visione dell'algoritmo finale.

6.2.1 Importance Sampling

Importance sampling è una tecnica che ha come idea principale quello di pescare samples da una distribuzione f, detta *target*, a partire da una distribuzione iniziale nota g, detta *proposal*. Vengono utilizzati dei pesi $w = \frac{f}{g}$ per dare una importanza ai sample pescati da g, tenendo conto di f (un esempio è visibile in figura 13). Al contrario, disponendo di una distribuzione proposal con dei pesi associati, sarà possibile generare la distribuzione target a partire dai sample proposal, dando ad essi maggiore peso: $f = w \cdot g$.

6.2.2 Resampling

Vi è la necessità di effettuare un resampling quando si ha a disposizione una distribuzione pesata, cioè è necessario pescare dei sample da tale distribuzione in maniera efficiente. In generale, avendo l'insieme di partenza $\mathcal{X}$ descritto in precedenza, si vuole che la probabilità di pescare $x^{[i]}$ sia proporzionale a w^i.

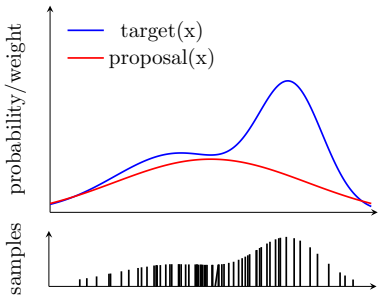


Figure 13: Esempio di importance sampling.

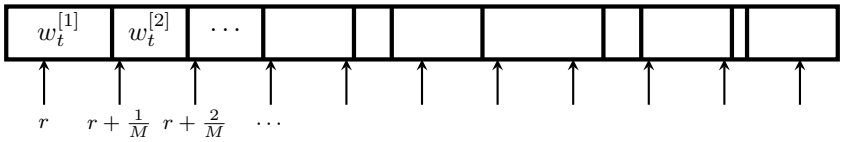


Figure 14: Esempio di systematic resampling.

La tecnica più immediata è quella di distribuire ed ordinare tutti i pesi e sorteggiare M numeri casuali nell'intervallo $[0, \sum_{i=1}^{M} w^i]$, verificando poi a quale ipotesi i corrisponda ogni numero sorteggiato; questo metodo ha complessità $O(M \log M)$.

Una alternativa è il **systematic resambling**, nel quale viene scelto un numero r casuale e successivamente si aggiunge M^{-1} per ogni sample (figura 14), ovvero: $u = r + (m-1) \cdot (1/M)$ con $m = 1, \ldots, M$.

6.2.3 Algoritmo del Filtro Particellare

Nell'algoritmo del filtro particellare (algoritmo 6) l'idea è quella di partire dall'insieme pesato delle particelle precedenti, e pescare degli indici attraverso resampling, seguendo la distribuzione data dai pesi delle particelle, ai quali corrisponde uno stato.

Algorithm 6 Particle_filter $(\mathcal{X}_{t-1}, u_t, z_t)$

1: $\mathcal{X}_t = \emptyset, \eta = 0$
2: **for** $i = 1$ to M **do**
3: sample index $j(i)$ — using **resampling** on weighted set $\mathcal{X}_{t-1}$
4: sample $x_t^{[i]}$ — from $p(x_t^{[i]} \mid u_t, x_{t-1}^{[j(i)]})$ using **motion model**
5: $w_t^{[i]} = p(z_t \mid x_t^{[i]})$ — using **sensor model**
6: $\eta = \eta + w_t^{[i]}$ — update normalization factor η
7: $\mathcal{X}_t = \mathcal{X}_t \cup \langle x_t^{[i]}, w_t^{[i]} \rangle$ — update particle set
8: **end for**
9: **for** $i = 1$ to M **do**
10: $w_t^i = w_t^i / \eta$ — normalize weights
11: **end for**
12: **return** $\mathcal{X}_t$

Si noti che se l'insieme delle particelle è tecnicamente un *multiset*, cioè uno stesso stato può essere pescato due volte nel sampling e comparire dunque due volte nell'insieme di particelle, aumentando dunque la sua probabilità di essere pescato. Per tale stato viene calcolato lo stato successivo più probabile utilizzando il motion model (ad esempio tramite algoritmo 2 o 4) e viene ad esso assegnato un peso sulla base del sensor model, cioè si cerca di dare una misura alla verosimilità dello stato successivo, basandosi

sulla misurazione effettuata al tempo t, ovvero z_t. Infine ogni nuova ipotesi pesata viene aggiunta al particle set e vengono normalizzati i pesi, ritornando l'insieme $\mathcal{X}_t$ finale.

I vantaggi di questo approccio sono una maggiore efficienza di algoritmo rispetto al filtro di Bayes discreto, sia a livello di tempo che di spazio, ottenendo un algoritmo capace di localizzare lo stato del robot, che però fatica a risolvere problematiche come il *kidnapped robot*, cioè il problema per cui un robot viene spostato senza tenere traccia di tale spostamento (potremmo aver scartato tutte le particelle relative alla nuova zona). Tali problematiche possono essere affrontate inserendo sample casuali, che possono anche essere proporzionali alla probabilità del kidnapping. Inoltre, anche se non specificato, l'algoritmo lavora in uno spazio discreto, che non è però discretizzato a priori, ad esempio tramite una griglia, ma è discretizzato nel senso che ad ogni iterazione si effettuano computazioni su un ristretto insieme di stati (l'insieme delle particelle), che però ha valori in un dominio continuo, e dunque tali valori cambiano continuamente di iterazione in iterazione.

7 Filtro di Kalman

Il filtro di Kalman è una delle possibili implementazioni del filtro di Bayes, in particolare si è scoperto essere un caso specifico di filtro di Bayes. Questo è un filtro *parametrico*, che ha 3 assunzioni iniziali.

A. Lo spazio di lavoro è continuo.

B. L'incertezza è definibile attraverso una distribuzione gaussiana.

C. Il sistema evolve tramite una funzione lineare (approfondito successivamente).

Iniziamo dunque approfondendo teoricamente le gaussiane.

7.1 Gaussiane

Una distribuzione normale (o gaussiana) è definita unidimensionalmente dal valore atteso μ e dalla deviazione standard σ:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \sim N(\mu, \sigma^2),$$

che nello spazio n-dimensionale, con $\boldsymbol{\mu}$ vettore dei valori attesi e $\boldsymbol{\Sigma}$ matrice di covarianza simmetrica positiva, diventa:

$$p(\mathbf{x}) \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma}) :$$
$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})'\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right).$$

La gaussiana ha due principali proprietà.

- Se una distribuzione di questo tipo viene modificata attraverso una combinazione lineare, tale distribuzione rimane normale.

$$\begin{cases} X \sim N(\mu, \sigma^2) \\ Y = aX + b \end{cases} \quad \Rightarrow \quad Y \sim N(a\mu + b, a^2\sigma^2).$$

- Se due distribuzioni normali vengono moltiplicate, la distribuzione rimane normale, con dei cambiamenti ai valori μ e σ.

$$\begin{cases} X_1 \sim N(\mu_1, \sigma_1^2) \\ X_2 \sim N(\mu_2, \sigma_2^2) \end{cases} \quad \Rightarrow \quad p(X_1) \cdot p(X_2) \sim N\left(\frac{\sigma_2^2}{\sigma_1^2+\sigma_2^2}\mu_1 + \frac{\sigma_1^2}{\sigma_1^2+\sigma_2^2}\mu_2,\ \frac{1}{\sigma_1^{-2}+\sigma_2^{-2}}\right)$$

7.2 Modellazione delle Azioni e delle Misurazioni

Nel filtro di Kalman la stima di uno stato x al tempo t, ovvero x_t si definisce come una combinazione lineare (teoricamente una equazione differenziale) disturbata da un rumore; questa è dunque la formalizzazione matematica del punto C della sezione 7:

$$x_t = A_t x_{t-1} + B_t u_t + \epsilon_t,$$

e la misurazione è definita come:

$$z_t = C_t x_t + \delta_t,$$

i cui componenti sono:

- $\boldsymbol{A_t}$: matrice $(n \times n)$ che descrive come lo stato evolve da t a $t-1$ senza controlli né rumore.
- $\boldsymbol{B_t}$: matrice $(n \times l)$ che descrive come il controllo u_t modifica lo stato da t a $t-1$.
- $\boldsymbol{C_t}$: matrice $(k \times n)$ che descrive come mappare lo stato x_t in un'osservazione z_t.

- $\boldsymbol{\epsilon_t}$ e $\boldsymbol{\delta_t}$: variabili aleatorie che rappresentano il rumore di processo e di misura, che si assumono indipendenti e distribuite normalmente con covarianza R_t e Q_t, rispettivamente (questi valori spiegano il punto B della sezione 7).

Dobbiamo ora definire il motion model ed il sensor model, sulla base di come sono stati definiti uno stato ed una misurazione.

7.2.1 Motion Model

Nella definizione di stato è implicita la linearità con cui lo stato si evolve (cioè esso è una combinazione lineare che considera lo stato precedente e l'azione compiuta), che però viene disturbata da rumore gaussiano. Ne consegue che dunque la probabilità che il motion model esprime sia gaussiana:

$$P(x_t \mid u_t, x_{t-1}) = N(x_t; A_t x_{t-1} + B_t u_t, R_t),$$

utilizzando la notazione $N(a; b, c)$, per cui si ha che a sia di fatto l'input della normale definita da b e c. In particolare, per risolvere il problema del filtro di Bayes siamo interessati a $\overline{bel}(x_t)$, ovvero:

$$\overline{bel}(x_t) = \int \underbrace{P(x_t|u_t, x_{t-1})}_{\sim N(x_t; A_t x_{t-1} + B_t u_t, R_t)} \quad \underbrace{bel(x_{t-1})}_{\sim N(x_{t-1}; \mu_{t-1}, \Sigma_{t-1})} \quad dx_{t-1}.$$

Viste le distribuzioni normali che sappiamo rappresentare i termini dell'equazione per assunzione iniziale, si ha come risultato, dopo una interminabile dimostrazione matematica che a noi, da bravi ingegneri, non interessa, si possono ottenere i nuovi parametri della distribuzione normale di $\overline{bel}(x_t)$:

$$\begin{cases} \overline{\mu}_t = A_t \mu_{t-1} + B_t u_t \\ \overline{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t \end{cases}$$

7.2.2 Sensor Model

Un ragionamento simile può essere compiuto per il sensor model; sappiamo infatti che:

$$p(z_t \mid x_t) \sim N(z_t; C_t x_t, Q_t).$$

Dunque integrando con il motion model questa distribuzione, si ottiene il risultato finale $bel(x_t)$:

$$bel(x_t) = \eta \quad \underbrace{P(z_t \mid x_t)}_{\sim N(z_t; C_t x_t, Q_t)} \quad \underbrace{\overline{bel}(x_{t-1})}_{\sim N(x_t; \overline{\mu}_t, \overline{\Sigma}_t)} .$$

Da qui, come in precedenza, si possono ottenere, saltando vari passaggi matematici, i nuovi parametri della distribuzione normale:

$$\begin{cases} \mu_t = \overline{\mu}_t + K_t(z_t - C_t \overline{\mu}_t) \\ \Sigma_t = (I - K_t C_t)\overline{\Sigma}_t \end{cases} \qquad \text{with} \qquad K_t = \overline{\Sigma}_t C_t^T (C_t \overline{\Sigma}_t C_t^T + Q_t)^{-1}$$

7.3 Algoritmo del Filtro di Kalman

Viste le modellazioni delle azioni e delle misurazioni, possiamo enunciare l'algoritmo del filtro di Kalman. Questo algoritmo a due fasi principali, la *predizione* e la *correzione*:

In generale lo step predittivo aumenta l'incertezza in quanto l'azione rende lo stato più incerto, successivamente, con l'osservazione tramite sensori si effettua la fase di correzione in cui si restringono gli stati possibili riducendo l'incertezza. Questo filtro ha complessità polinomiale nelle dimensioni dello stato e delle misurazioni, inoltre è *ottimo* per sistemi gaussiani lineari.

Algorithm 7 Kalman_filter $(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$

1: **Prediction:**
2: $\overline{\mu}_t = A_t \mu_{t-1} + B_t u_t$
3: $\overline{\Sigma}_t = A_t \Sigma_{t-1} A_t^T + R_t$
4: **Correction:**
5: $K_t = \overline{\Sigma}_t C_t^T (C_t \overline{\Sigma}_t C_t^T + Q_t)^{-1}$
6: $\mu_t = \overline{\mu}_t + K_t (z_t - C_t \overline{\mu}_t)$
7: $\Sigma_t = (I - K_t C_t) \overline{\Sigma}_t$
8: **return** μ_t, Σ_t

7.4 Filtro di Kalman Esteso (EKF)

Di fatto però, come visto nelle analisi dei vari motion model, difficilmente essi sono descrivibili linearmente, così come i sensor model. Ne consegue che per poter applicare il filtro di Kalman dobbiamo linearizzare attraverso un'approssimazione tali modelli.
L'idea chiave è quella di effettuare una linearizzazione del sistema attorno al valore atteso dello stato, in modo tale che, essendo esso lo stato più probabile, l'approssimazione sia più accurata in quel punto e, via via che ci si allontana, l'approssimazione è meno precisa fino a che non si diverge (esempio in figura 15).

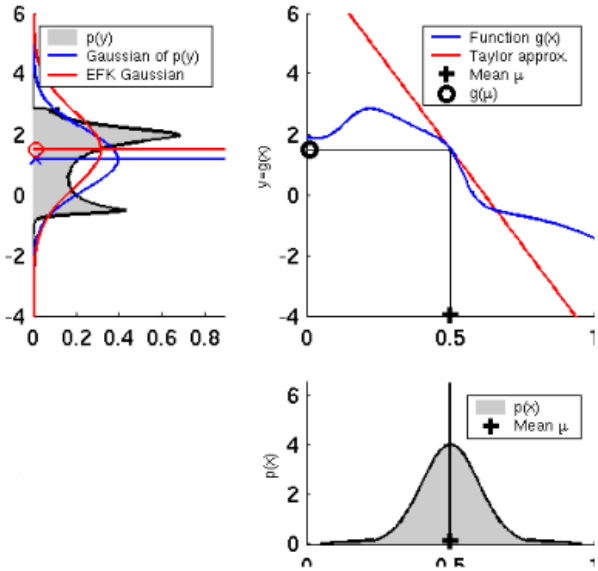


Figure 15: Linearizzazione attraverso l'approssimazione di Taylor della funzione $g(x)$.

Nell'algoritmo del filtro di Kalman esteso si sostituiscono le matrici A e C con le Jacobiane G e H (matrici che, per farla breve, rappresentano l'orientamento del piano tangente di una funzione in un determinato punto) calcolate nel punto μ_{t-1}.

Questo algoritmo è largamente utilizzato in quanto è efficiente, esattamente come il filtro di Kalman liscio, anche se al contrario di quest'ultimo non risulta ottimo; un punto critico dell'algoritmo è la possibile divergenza citata in precedenza, anche se la'lgoritmo in generale risulta spesso efficace.

8 Mapping

Il problema del mapping è uno di problemi principali nella robotica mobile, in quanto imparare le mappe è necessario per task di pianificazione, localizzazione, ecc. Il problema del mapping può essere esplicitato come:

$$m^* = \arg\max_m P(m \mid d)$$

Con d insieme contenente le varie misurazioni ed i vari input al tempo t:

$$d = \{u_1, z_1, u_2, z_2, \ldots, u_n, z_n\}$$

Nel problema del mapping, daremo come assunzione generica quella di conoscere lo stato presente e passato del robot $x_{1:t}$.

8.1 Grid Maps

Noi ci occuperemo ora di **grid maps**, cioè delle mappe in forma di griglia con una struttura rigida, nelle quali utilizziamo alcune assunzioni per semplificare il contesto ed i calcoli.

A. Ogni cella della griglia ha uno stato binario, cioè può solamente essere libera oppure occupata.
Tale assunzione è rappresentata dalla probabilità: data per esempio la cella i, ovvero m_i, si ha che $p(m_i) \in \{0, 0.5, 1\}$, con $p(m_i) = 1$ se la cella è occupata, $p(m_i) = 0$ se la cella è libera, oppure $p(m_i) = 0.5$ se non vi è alcuna informazione a riguardo.
L'ambiente in questo caso si assume che sia statico, dunque non vi sono ostacoli mobili come ad esempio persone che camminano.

B. Le celle sono indipendenti tra loro. ovvero lo stato di una singola cella è ininfluente sullo stato di tutte le altre celle, siano esse adiacenti oppure lontane.
Questa assunzione è data principalmente da motivi computazionali, in quanto, ad esempio con una mappa da 100 celle, se non vi fosse indipendenza, avremo uno spazio di input, vista una probabilità binaria, di 2^{100}, mentre con una assunzione di indipendenza si arriva a dover considerare 100 singole celle.
Ciò si traduce nel fatto che, vista la mappa intera m, la probabilità $p(m)$ sia $p(m) = \prod_i p(m_i)$.

Tornando al probelma postoci inizialmente, proviamo a trovare la probabilità della mappa m. Il primo passaggio deriva dall'assunzione di indipendenza precedente **B**:

$$p(m \mid z_{1:t}, x_{1:t}) = \prod_i p(m_i \mid z_{1:t}, x_{1:t})$$

con m_i variabile binaria, vista l'assunzione precedente **A**. Dunque, spostiamo il problema sulla singola cella m_i.

$$\begin{aligned}
\boldsymbol{p(m_i \mid z_{1:t}, x_{1:t})} \quad &= \\
\text{(Bayes)} \quad &= \frac{p(z_t \mid m_i, z_{i:t-1}, x_{1:t}) p(m_i \mid z_{i:t-1}, x_{1:t})}{p(z_t \mid z_{i:t-1}, x_{1:t})} \\
\text{(Markov)} \quad &= \frac{p(z_t \mid m_i, x_t) p(m_i \mid z_{i:t-1}, x_{1:t-1})}{p(z_t \mid z_{i:t-1}, x_{1:t})} \\
\text{(Bayes)} \quad &= \frac{p(m_i \mid z_t, x_t) p(z_t \mid x_t)}{p(m_i \mid x_t)} \frac{p(m_i \mid z_{i:t-1}, x_{1:t-1})}{p(z_t \mid z_{i:t-1}, x_{1:t})} \\
\text{(Markov)} \quad &= \frac{p(m_i \mid z_t, x_t) p(z_t \mid x_t)}{p(m_i)} \frac{p(m_i \mid z_{i:t-1}, x_{1:t-1})}{p(z_t \mid z_{i:t-1}, x_{1:t})}.
\end{aligned} \tag{3}$$

Spiegamo ora maggiormente nel dettaglio i passaggi: innanzitutto è stata applicata la regola di Bayes alla formula iniziale; successivamente l'assunzione di Markov. Quest'ultima riprende l'assunzione di Markov proposta con il ragionamento probabilistico (figura 6), nella quale è stata aggiunta la mappa m, visibile

come un nodo che si collega ad ogni misurazione $z_{1:t}$ (figura 16). Dunque z_t è indipendente da $z_{1:t-1}$ ed $x_{1:t-1}$ ed m_i è indipendente da x_t.
Successivamente si applica la regola di Bayes al termine $p(z_t \mid m_i, x_t)$. L'ultima applicazione si rifà alla precedente, in quanto m_i è dipendente dalla coppia (z_t, x_t), ma non dal singolo x_t.

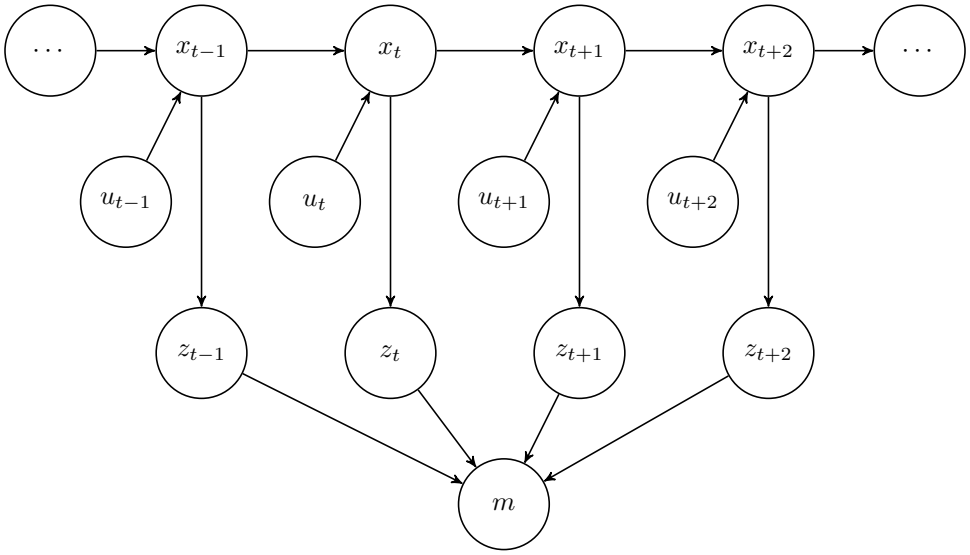


Figure 16: Schema delle dipendenze di Markov con mappa m (non ne sono sicuro ma credo di si).

Notazione Odds Questa notazione permettedi descrivere una probabilità $p(x)$, visto il rapporto tra essa e $p(\neg x)$: $c = p(x)/p(\neg x)$, dunque visto $p(\neg x) = 1 - p(x)$ e quindi $p(x) = c/(1+c)$. Utilizzando la formula precedente con $\neg m_i$ si ha:

$$p(\neg m_i \mid z_{i:t}, x_{1:t}) = \frac{p(\neg m_i \mid z_t, x_t)p(z_t \mid x_t)}{p(\neg m_i)} \frac{p(\neg m_i \mid z_{i:t-1}, x_{1:t-1})}{p(z_t \mid z_{i:t-1}, x_{1:t})}.$$

L'idea a questo punto è quella di mettere in rapporto le due probabilità, così da semplificare alcuni delle probabilità più "spinose" ed utilizzare la notazione odds.

$$\begin{aligned} \frac{p(m_i \mid z_{i:t}, x_{1:t})}{p(\neg m_i \mid z_{i:t}, x_{1:t})} &= \frac{p(m_i \mid z_t, x_t)p(m_i \mid z_{1:t-1}, x_{1:t-1})p(\neg m_i)}{p(\neg m_i \mid z_t, x_t)p(\neg m_i \mid z_{1:t-1}, x_{1:t-1})p(m_i)} \\ &= \underbrace{\frac{p(m_i \mid z_t, x_t)}{1 - p(m_i \mid z_t, x_t)}}_{\text{utilizzando } z_t} \underbrace{\frac{p(m_i \mid z_{1:t-1}, x_{1:t-1})}{1 - p(m_i \mid z_{1:t-1}, x_{1:t-1})}}_{\text{termine ricorsivo}} \underbrace{\frac{1 - p(m_i)}{p(m_i}}_{\text{prior}}. \end{aligned} \tag{4}$$

Spesso scritto come:

$$Bel(m_t^i) = \left[1 + \frac{1 - p(m_t^i \mid z_t, x_t)}{p(m_t^i \mid z_t, x_t)} \cdot \frac{p(m_t^i)}{1 - p(m_t^i)} \cdot \frac{1 - Bel(m_{t-1}^i)}{Bel(m_{t-1}^i)}\right]^{-1}.$$

Inoltre, utilizzando la **notazione log odds**, ovvero: $l(x) = \log \frac{p(x)}{1-p(x)}$, le moltiplicazioni di probabilità possono essere scritte come somme, per cui:

$$l(m_i \mid z_{1:t}, x_{1:t}) = \underbrace{l(m_i \mid z_t, x_t)}_{\text{modello dei sensori inverso}} + \underbrace{l(m_i \mid z_{1:t-1}, x_{1:t-1})}_{\text{termine ricorsivo}} - \underbrace{l(m_i)}_{\text{prior}}.$$

O, in forma breve:

$$l_{t,i} = \text{inv\_sensor\_model}(m_i, x_t, z_t) + l_{t-1,i} - l_0.$$

Ovviamente, il termine $l(m_i \mid z_{1:t}, x_{1:t})$ riporta direttamente alla probabilità $p(m_i \mid z_{1:t}, x_{1:t})$, visto che in generale si ha che $p(x) = 1 - (1 + \exp l(x))^{-1}$.

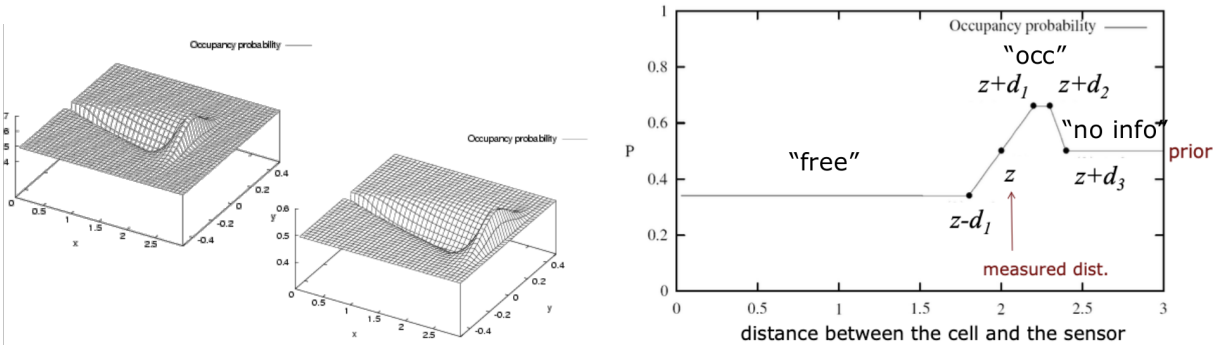
Dunque, è stato trovato un modo per calcolare la probabilità desiderata, cioè $p(m \mid z_{1:t}, x_{1:t})$; ne consegue che ora possiamo applicarla all'interno di un algoritmo, il **occupancy grid map** o mapping con pose note (algoritmo 8).

Algorithm 8 occupancy_grid_mapping $(\{l_{t-1,i}\}, x_t, z_t)$

1: **for** all cells m_i **do**
2: **if** m_i in perceptual field of z_t **then**
3: $l_{t,i} = l_{t-1,i} + \text{inv\_sensor\_model}(m_i, x_t, z_t) - l_0$
4: **else**
5: $l_{t,i} = l_{t-1,i}$
6: **end if**
7: **end for**
8: **return** $\{l_{t,i}\}$

8.2 Modello dei Sensori Inverso

Uno dei termini usati dall'algoritmo 8 è $p(m_i \mid z_t, x_t)$, per la quale necessitiamo di un modello. Basandoci su sensori sonar, si ha una distribuzione di probabilità a forma di cono che forma un picco intorno alla distanza del primo oggetto posto di fronte. Oltre quella distanza e fuori dal cono d'influenza la probabilità è costante a $\frac{1}{2}$ in quanto non si ha alcuna informazione (figura 17).



(a) Modello dei sensori inverso, 3D (posizione 2D, probabilità).

(b) Modello dei sensori inverso, 2D (distanza, probabilità).

Figure 17: Modelli dei sensori inversi.

9 SLAM: Simultaneous Localization and Mapping

Il problema che SLAM vuole risolvere è quello di trovare una mappa ed una posa contemporaneamente: spesso infatti il robot non ha una mappa predesignata, e dunque non è nemmeno in grado di localizzarsi all'interno di una mappa. Abbiamo visto risoluzioni ai problemi singoli tramite il mapping e tramite la localization, ma per risolverli entrambi contemporaneamente si ha un problema di dipendenza ciclica. In pratica, è nato prima l'uovo o la gallina?

SLAM risponde a questa filosofica domanda trovando applicazioni in numerosi ambiti della robotica, dai robot indoor per l'aspirazione, fino a robot spaziali o sottomarini.
Per la risposta, specifichiamo che esistono due principali tipi di mappe:le griglie o scan (sia 2d che 3d), e le mappe basate su landmark, già accennate, sulle quali opereremo principalmente.

Gli stessi problemi SLAM possono dividersi in due tipi.

- **Full SLAM**: stima l'intero cammino e la mappa, ovvero $p(x_{1:t}, m \mid z_{1:t}, u_{1:t})$.
- **Online SLAM**: stima lo stato attuale e la mappa, ovvero $p(x_t, m \mid z_{1:t}, u_{1:t})$.

9.1 Feature-Based SLAM

In questo tipo di SLAM il problema viene formulato nella maniera seguente:

- Input:
  1. i comandi del robot $u_{1:k} = \{u_1, u_2, \ldots, u_k\}$;
  2. le osservazioni relative $z_{1:k} = \{z_1, z_2, \ldots, z_k\}$.
- Output:
  1. la mappa delle features $m = \{m_1, m_2, \ldots, m_n\}$;
  2. il percorso del robot $x_{1:k} = \{x_1, x_2, \ldots, x_k\}$.

L'idea deriva per la risoluzione deriva dal filtro di Kalman. Al filtro classico, infatti, vengono aggiunte le posizioni dei landmarks (features), così da riuscire a stimare anche la loro posizione. Nel filtro di Kalman classico avevamo la posa $\boldsymbol{x}_k$ e la matrice di covarianza Σ_k:

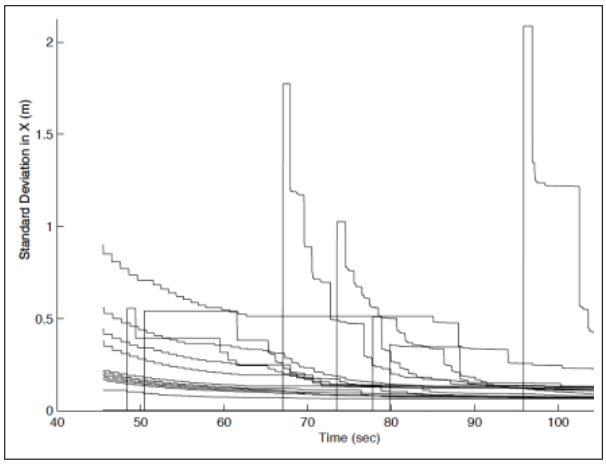
$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} \quad \Sigma_k = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\theta} \\ \sigma_{yx} & \sigma_y^2 & \sigma_{y\theta} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_\theta^2 \end{bmatrix}$$

Tali matrici, abbiamo detto essere estese dai landmarks, per cui:

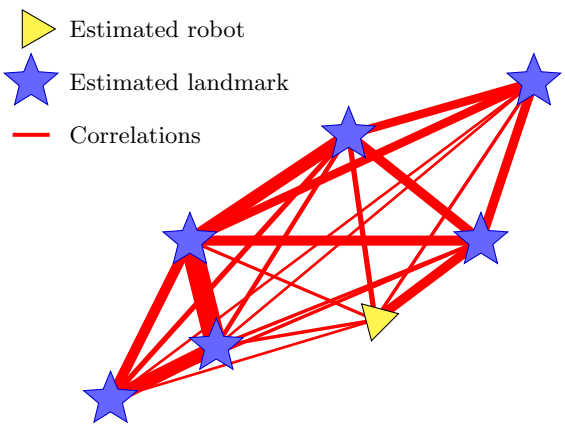
$$\mathbf{x}_k = \begin{bmatrix} x_R \\ m_1 \\ m_2 \\ \vdots \\ m_n \end{bmatrix} \quad \Sigma_k = \begin{bmatrix} \Sigma_R & \Sigma_{RM_1} & \Sigma_{RM_2} & \cdots & \Sigma_{RM_n} \\ \Sigma_{M_1R} & \Sigma_{M_1} & \Sigma_{M_1M_2} & \cdots & \Sigma_{M_1M_n} \\ \Sigma_{M_2R} & \Sigma_{M_2M_1} & \Sigma_{M_2} & \cdots & \Sigma_{M_2M_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \Sigma_{M_nR} & \Sigma_{M_nM_1} & \Sigma_{M_nM_2} & \cdots & \Sigma_{M_n} \end{bmatrix}$$

In particolare la matrice di covarianza si può evidenziare:

$$\mu = \begin{bmatrix} x \\ y \\ \theta \\ m_{1,x} \\ m_{1,y} \\ \vdots \\ m_{n,x} \\ m_{n,y} \end{bmatrix}, \Sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xm_{1,x}} & \sigma_{xm_{1,y}} & \cdots & \sigma_{xm_{n,x}} & \sigma_{xm_{n,y}} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} & \sigma_{ym_{1,x}} & \sigma_{ym_{1,y}} & \cdots & \sigma_{ym_{n,x}} & \sigma_{ym_{n,y}} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_{\theta\theta} & \sigma_{\theta m_{1,x}} & \sigma_{\theta m_{1,y}} & \cdots & \sigma_{\theta m_{n,x}} & \sigma_{\theta m_{n,y}} \\ \sigma_{m_{1,x}x} & \sigma_{m_{1,x}y} & \sigma_{m_{1,x}\theta} & \sigma_{m_{1,x}m_{1,x}} & \sigma_{m_{1,x}m_{1,y}} & \cdots & \sigma_{m_{1,x}m_{n,x}} & \sigma_{m_{1,x}m_{n,y}} \\ \sigma_{m_{1,y}x} & \sigma_{m_{1,y}y} & \sigma_{m_{1,y}\theta} & \sigma_{m_{1,y}m_{1,x}} & \sigma_{m_{1,y}m_{1,y}} & \cdots & \sigma_{m_{1,y}m_{n,x}} & \sigma_{m_{1,y}m_{n,y}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \sigma_{m_{n,x}x} & \sigma_{m_{n,x}y} & \sigma_{m_{n,x}\theta} & \sigma_{m_{n,x}m_{1,x}} & \sigma_{m_{n,x}m_{1,y}} & \cdots & \sigma_{m_{n,x}m_{n,x}} & \sigma_{m_{n,x}m_{n,y}} \\ \sigma_{m_{n,y}x} & \sigma_{m_{n,y}y} & \sigma_{m_{n,y}\theta} & \sigma_{m_{n,y}m_{1,x}} & \sigma_{m_{n,y}m_{1,y}} & \cdots & \sigma_{m_{n,y}m_{n,x}} & \sigma_{m_{n,y}m_{n,y}} \end{bmatrix} \quad (5)$$



(a) Esempio di diminuzione dell'incertezza tramite loop closure.



(b) Correlazione completa tra landmarks e tra landmarks e robot.

Figure 18: Esempio di loop closure e correlazione tra landmarks.

L'algoritmo si basa ancora una volta su quello del filtro di Kalman: la prima fase è quella di *predizione*, in cui a partire dall'input corrente u_t si stima il nuovo stato x_t (in base al modello di odometry che si adotta); ciò aumenta l'incertezza e dunque aumenta anche la covarianza Σ (essendo che però questa fase coinvolge solo il robot e non i landmark, a cambiare sono le prime tre colonne e le prime tre righe della matrice Σ nell'equazione 5).

Successivamente, vi è, sempre seguendo l'algoritmo del filtro di Kalman, una fase di *correzione*, nella quale si correggono le covarianze in base ai landmark rilevati dai sensori. Guardando dal libro, questa fase e quelle successive risultano dense di calcoli, talvolta molto complessi, per i quali non entreremo nel dettaglio.

Nella fase di correzione, tuttavia, è necessario affiancare una fase di *data association*, il cui scopo è quello di associare i landmark alle misurazioni, e si distinguono i nuovi landmark da quelli già annotati; nel caso di nuovi landmark si espanderanno il vettore μ e la matrice Σ.
In questa fase può inoltre verificarsi il cosiddetto *loop closure*, ovvero può avvenire il momento in cui il robot incontra un landmark nuovamente, e dunque l'incertezza circostante si riduce notevolmente, convergendo verso i valori reali. Infatti, appena un landmark viene scoperto, la sua incertezza (misurata dal determinante della sottomatrice associata) è massima, e successivamente va riducendosi monotonicamente (esempio in figura 18a).

In questa fase la cross correlazione tra il robot ed i landmarks e tra i vari landmarks è fondamentale. Immaginando infatti di eliminare queste cross correlazioni (equazione 6) non si potrebbe avere la fase di *data association*, senza la possibilità di scindere vecchi landmark dai nuovi e dunque aggiungendo landmarks già visti.

Con l'aumentare della certezza (cioè asintoticamente), vi è una correlazione completa tra tutti i landmarks e il robot e tutti i landmarks (esempio in figura 18b).
Inoltre, asintoticamente la covarianza associata ai landmark dipende solamente dalla covarianza associata alla stima della posizione iniziale.

$$\Sigma_k = \begin{bmatrix} \Sigma_R & 0 & \cdots & 0 \\ 0 & \Sigma_{M_1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \Sigma_{M_n} \end{bmatrix} \quad \Sigma_{RM_i} = \mathbf{0}_{3\times 2} \quad \Sigma_{M_i M_{i+1}} = \mathbf{0}_{2\times 2} \tag{6}$$

Lo step finale è quello di update degli stati, in cui vengono calcolati e restituiti i nuovi valori delle matrici μ e Σ.

NB La formulazione matematica dell'algoritmo non è chiarissima, ma nel libro di riferimento, *Probabilistic Robots*, si può vedere completamente alla tabella 10.1 di pagina 249; nel caso ti venga voglia di addentrarti nei perchè e i per come di questo algoritmo, buona martellata sui coglioni.

Note sull'algoritmo La complessità di questo algoritmo è $O(n^2)$ nel numero di landmark, ma il costo totale per costruire una mappa con n landmarks è $O(n^3)$, con consumo di $O(n^2)$. In generale funziona bene per mappe medio-piccole ed esistono approssimazioni per ridurre la complessità computazionale, quanto non utilizzabile per mappe grandi. Inoltre, se vi è molta non linearità può esserci divergenza (di fatto si basa su EFK), ma vi è prova di convergenza per casi di linearità gaussiana.

9.2 FastSLAM

Abbiamo visto che all'aumentare della mappa, il metodo feature-based di SLAM riduce drasticamente le proprie performance. Vogliamo quindi utilizzare l'idea alla base del particle filter (6.2) ed applicarla allo SLAM. Contrariamente all'idea del feature-based SLAM, inoltre, ci poniamo la domanda: "è necessario calcolare tutte le correlazioni tra tutti i landmark?". Esse infatti portano una quadratica espansione dello spazio per ogni nuovo landmark; possiamo dunque evitare una correlazione completa ma considerare solo alcune dipendenze, in modo da risolvere più efficacemente il problema. Per altro, non sarebbe computazionalmente fattibile applicare un filtro particellare a SLAM in maniera brute force proprio a causa dell'elevato numero di correlazioni e dimensioni.

9.2.1 Rao-Blackwellization

Un teorema matematico chiave alla risoluzione del problema è il teorema di Rao-Blackwell. Dati due eventi a e b, si può effettuare la fattorizzazione:

$$p(a, b) = p(a) \cdot p(b \mid a)$$

Se $p(b \mid a)$ è computabile in forma chiusa, allora si può rappresentare $p(a)$ tramite sampling e calcolare $p(b \mid a)$ per ogni sample, ottenendo dunque $p(a, b)$.
In questa formulazione poniamo la posizione del robot come evento a e la mappa come evento b; le particelle del particle set ($p(a)$) possono essere associate ad un insieme di landmark.

9.2.2 Formulazione del Factored Posterior

Per effettuare la formulazione (di tipo *full SLAM*) si parte dall'assuzione per cui le variabili dei landmark siano indipendenti se il cammino del robot è noto (figura 19).

$$\begin{aligned} \boldsymbol{p(x_{1:t}, l_{1:m} \mid z_{1:t}, u_{1:t-1})} &= \\ \text{(Bayes)} \quad &= p(x_{1:t} \mid z_{1:t}, u_{1:t-1}) \cdot p(l_{1:m} \mid x_{1:t}, u_{1:t-1}, z_{1:t}) \\ \text{(Indipendenza)} \quad &= p(x_{1:t} \mid z_{1:t}, u_{1:t-1}) \cdot p(l_{1:m} \mid x_{1:t}, z_{1:t}) \\ \text{(Rao-Blackwell)} \quad &= \underbrace{p(x_{1:t} \mid z_{1:t}, u_{1:t-1})}_{\substack{\text{path posterior} \\ \text{(localization problem)}}} \prod_{i=1}^{M} \underbrace{p(l_i \mid x_{1:t}, z_{1:t})}_{\substack{\text{indipendent} \\ \text{landmarks}}} \end{aligned} \tag{7}$$

Nella formula, il path posterior è il problema della localization, tradotto nel filtro di Bayes (formulazione iniziale, sottosezione 4.6) e risolto varie volte nei capitoli precedenti), mentre i landmark fanno capo al problema del mapping (sezione 8).

Ogni landmark l è rappresentato da un filtro di Kalman esteso di dimensione 2×2, ed ogni particella è associata a M filtri di Kalman estesi. Dunque, al contrario dello SLAM feature-based, in cui dati n landmarks si aveva una filtro di Kalman con una matrice delle covarianze $2n \times 2n$, si è passati ad n matrici 2×2 per ogni particella. Ognuno di questi filtri di Kalman viene aggiornato ad ogni istante temporale,

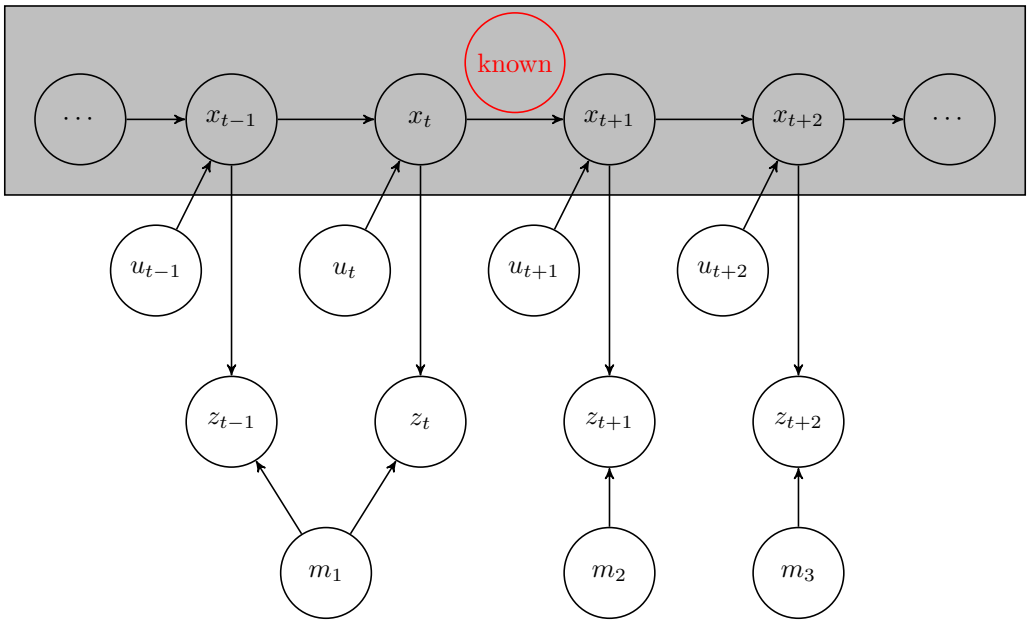


Figure 19: Assunzione di inpendenza sulla mappa dato un cammino del robot noto.

ed ogni particella viene pesata ancora sulla base del sensor model, cioè osservando la consistenza della nuova misurazione rispetto ai filtri di Kalman dello stato precedente della particella.

Naturalmente, nell'algoritmo vi è una fase di data association per associare i vari landmarks: ogni particella può decidere di prendere il match migliore o scegliere casualmente in maniera pesata alla probabilità oppure se troppo bassa generare un landmark nuovo. Si noti che l'aggiunta di nuovi landmarks può essere fatto sulla singola particella invece che sul sistema intero, come nello SLAM basato su feature (per cui avremmo aggiunto una riga ed una colonna intere).

Questo algoritmo avrebbe complessità $\mathcal{O}(NM)$, date N particelle ed M lanmarks, ma utilizzando una struttura dati ad albero per la memorizzazione, si può arrivare ad una complessità $\mathcal{O}(N \log M)$.

10 Path e Motion Planning

Essendo che il robot ha necessità di sapere quale percorso compiere, nota la propria posizione attuale e la mappa il motion planning si occupa di definire tale percorso, dandosi come obiettivo di arrivare alla destinazione il più velocemente possibile evitando collisioni.

Il mondo infatti non è statico e il robot deve evitare anche ostacoli dinamici, per cui si applicano due diversi approcci:

- **proattivo**: calcola il cammino ottimo tenendo in considerazione le potenziali incertezze delle azioni (Markov Decision Process);
- **reattivo**: lavora su un orizzonte temporale più breve e reagisce velocemente in caso di imprevisti.

Esistono dunque due livelli separati, uno più ad alto livello che lavora su un orizzonte temporale più lontano effettuando una pianificazione a lungo termine e fornendo sub-goal al livello sottostante, il quale invece lavora invece su un orizzonte temporale più breve allo scopo di effettuare *collision avoidance*.

10.1 Dynamic Window Approach

Un primo approccio per ovviare alla *collision avoidance* è quello, a partire dal controllo dei comandi della velocità v e della velocità rotazionale ω, capire quali sono i valori (v, ω) raggiungibili. Ciò avviene rispettando dei vincoli di ammissibilità e raggiungibilità:

ammissibilità le velocità che permettono al robot di fermarsi prima di raggiungere l'ostacolo;
$V_a = \{(v, \omega) \mid v \leq \sqrt{2\,\mathrm{dist}(v, \omega) a_{trans}} \wedge \omega \leq \sqrt{2\,\mathrm{dist}(v, \omega) a_{rot}}\}$

raggiungibilità le velocità che sono raggiungibili con l'accelerazione;
$V_d = \{(v, \omega) \mid v \in [v - a_{trans}t, v + a_{trans}t] \wedge \omega \in [\omega - a_{rot}t, \omega + a_{rot}t]\}$

Questi vincoli determinano una riduzione dello spazio di ricerca delle velocità V_s fino ad arrivare allo spazio delle velocità V_r, che prende il nome di *dynamic window* ($V_r = V_s \cap V_a \cap V_d$). All'interno di questa finestra le velocità sono selezionate tramite un'euristica che cerca di "andare velocemente nella giusta direzione" nota come navigation function:

$$G(v, \omega) = \sigma(\alpha \cdot \mathrm{heading}(v, \omega) + \beta \cdot \mathrm{dist}(v, \omega) + \gamma \cdot \mathrm{vel}(v, \omega))$$

modificata poi in modo che sia catturato il fatto che il robot stia seguendo il path pianificato. Questo approccio ha il vantaggio di essere molto veloce, computazionalmente poco costoso ma ha il difetto di oltrepassare il goal in quanto non rallenta in tempo in determinate circostanze (problema dei passaggi stretti della figura 20).

10.2 Motion Planning

Il problema del motion planning può essere esposto come l'insieme di:

- una posa iniziale del robot,
- una posa di destinazione desiderata,
- una descrizione geometrica del robot,
- una rappresentazione geometrica dell'ambiente ambiente.

In questo problema l'obiettivo è trovare un percorso che sposti il robot gradualmente dall'inizio alla meta senza toccare alcun ostacolo. Il robot non è rappresentato dalla sua sola posizione bidimensionale ma da anche rotazioni, tale rappresentazione è detta Configuration Space o **C-space** e il robot è un punto in questo spazio. Dunque il problema implica che bisognerà muovere il robot all'interno di questo spazio.

Tale spazio è continuo, ma può essere discretizzato seguendo due approcci differenti:

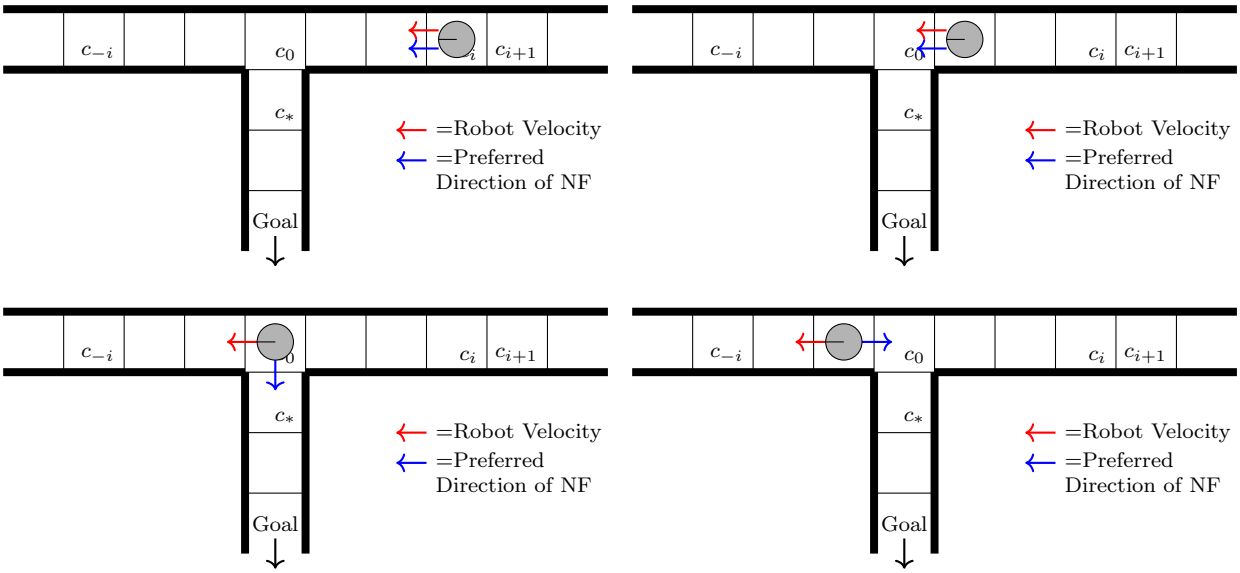


Figure 20: Problema dei passaggi stretti con il dynamic window approach.

- **Combinatorial Planning**: cattura la porzione di C-space libera da ostacoli in un grafo su cui applica ricerca per trovare una soluzione;
- **Sampling-Based Planning**: usa il rilevamento delle collisioni per sondare e cercare incrementamente una soluzione nel C-space.

Bisogna però capire sia possibile effettuare una ricerca in questi spazi.

10.3 Ricerca

Il problema della ricerca consiste nel trovare una sequenza di azioni, ovvero un percorso, che porti a degli stati desiderabili (stati obiettivo). La ricerca può essere non informata (con algoritmi come breadth-first, depth-first, uniform-cost, ...) oppure informata (ovvero con informazioni riguardo il dominio tramite delle euristiche, come A* e varianti). Un problema di ricerca può essere definito come una tupla $\langle S, I, A, AP, T, C\rangle$ dove:

- S: insieme delle etichette, ciascuna rappresentata da una stringa,
- $I \in S$: uno stato iniziale appartenente allo spazio degli stati,
- A: insieme delle etichette delle azioni, ciascuna rappresentata da una stringa,
- $AP : S \to 2^A$: azioni applicabili (in seguito chiamato semplicemente *azioni*),
- $T : S \times A \to S$: stati successori (in seguito chiamato semplicemente *risultato*),
- $C : S \times A \to \mathbb{N}$: costo.

Inoltre, si introducono alcuni concetti generici:

soluzione una sequenza $\pi : \langle a_0, a_1, \ldots, a_n\rangle$ di etichette da A,

stati indotti una sequenza di stati $\langle I = s_0, s_1, s_2, \ldots, s_{n+1}\rangle$ tale che $s_{i+1} = T(s_i, a_i)$ per ogni i,

soluzione valida una soluzione in cui ogni azione è applicabile nello stato che la precede; ovvero: $a_0 \in AP(I), \ldots, a_i \in AP(s_{i-1}), \ldots$,

soluzione ottimale una soluzione π che è valida e tale da minimizzare $\sum_{a \in \pi} C(s_i, a)$.

Come visto nel corso di Intelligenza Artificiale, è possibile costruire un albero di ricerca per esplorare le soluzioni, avente una frontiera (ovvero un insieme dei possibili nodi da espandere per proseguire nell'esplorazione dell'albero), e tale albero è esplorabile in vari modi.
Le possibile metodologie di visita dell'albero sono valutabili secondo vari criteri, di seguito riportati.
▶ **Completezza**: se esiste una soluzione, l'algoritmo la trova? E se non c'è, l'algoritmo è in grado di dirlo?
▶ **Complessità temporale**: quanto tempo impiega?
▶ **Complessità spaziale**: quanta memoria consuma?
▶ **Garanzia di ottimalità**: è sempre possibile trovare soluzioni ottimali? La risposta è binaria (si oppure no).
Tali proprietà sono trovate in funzione del branching factor b, profondità della soluzione d e della massima profondità di un cammino m.

Le modalità (algoritmi) di esplorazione dell'albero sono di seguito descritti.

Breadth-first è completo se b e d sono finiti; ottimo se hanno azioni con costo uniforme, temporale $O(b^d)$, spaziale $O(b^d)$.

Depth-first è completo se non ci sono loop e lo spazio degli stati è finito; non ha garanzia di ottimalità, temporale $O(b^m)$, spaziale $O(b \cdot m)$.

Costo uniforme (algoritmo di Dijkstra) espande prima il nodo con costo minore, è completo se b e d sono finiti ed è ottimo, temporale $O(b^{1+C^*/\epsilon})$, spaziale $O(b^{1+C^*/\epsilon})$.

Greedy best first utilizza come criterio di espansione l'euristica $h(n)$, è completo se b e d sono finiti, non è ottimo, ha complessità temporale $O(b^m)$ e spaziale $O(b^m)$.

A* utilizza come criterio di espansione $g(n)+h(n)$ (ovvero la somma tra il costo per raggiungere il nodo n e l'euristica per raggiungere il goal dal nodo n), è completo, ottimo se usa un'euristica ammissibile; ha complessità temporale $O(b^{1+C/\epsilon})$ e spaziale $O(b^{1+C/\epsilon})$. In generale un'euristica $h(n)$ è ammissibile se $h(n) \leq h^*(n) \forall n$; un'euristica consistente (cioè per la quale vale la disuguaglianza triangolare) è anche ammissibile.

Weighted A* utilizza come criterio di espansione $g(n)+w \cdot h(n)$, per cui maggiore è w più l'algoritmo è greedy verso il goal; questo algoritmo vuole essere un compromesso tra velocità di esecuzione e ottimalità.

10.3.1 5D Planning

Un'alternativa all'architettura a due livelli è quella che considera la pianificazione nello spazio degli stati 5-dimensionale $\langle x, y, \theta, v, \omega \rangle$ tenendo in considerazione i vincoli cinematici del robot $|v1 - v2| \leq a_{trans}t$ e $|\omega_1 - \omega_2| \leq a_{rot}t$. Il problema di questo approccio sta nel fatto che lo spazio degli stati è enorme, non è possibile perciò effettuare la ricerca abbastanza velocemente: la soluzione che si introduce consiste nel restringere lo spazio di ricerca.

Si utilizza A* per effettuare una ricerca nello spazio bidimensionale $\langle x, y \rangle$, rimuovendo dunque dallo spazio di ricerca 5-dimensionale tutti quegli stati troppo lontani dal path trovato nella prima ricerca, per poi eseguire un'ulteriore ricerca A* nello spazio 5-dimensionale ristretto al solo canale intorno al percorso ottimo.

Deve fornire molto frequentemente i comandi per la movimentazione, nel caso la ricerca non termini in tempo si continua a seguire il piano trovato in precedenza. Questo approccio inoltre risolve anche il problema dei passaggi stretti che si aveva con Dynamic Window.

10.4 Motion Replanning

Attuando il planning nell'ambito della robotica, a livello pratico ci si rende conto che il planning è un processo che viene eseguito di continuo, in quanto il mondo, durante il movimento, può modificarsi nel tempo. Nel motion replanning, dunque, l'idea è qulla di effettuare una pianificazione incrementale che riutilizzi le informazioni ottenute negli step precedenti, così da non dover continuamente effettuare una ricerca da zero. Per mettere in pratica questa intuizione, esistono varie metodologie, ma quella che noi utilizzeremo è la **anytime heuristic search**.

10.4.1 Anytime Heuristic Search: Straw Man Approach

Una prima soluzione che si introduce tramite ricerca euristica anytime è quella di applicare continuamente weighted A* ad un percorso, avendo a disposizione un tempo di esecuzione T. La prima esecuzione sarà più greedy, ovvero avrà un w maggiore, e con l'avanzare del tempo tale peso diminuirà, fino a confluire a $w = 1$, cioè un A* classico. Tali esecuzioni saranno effettuate compatibilmente al tempo totale disponibile. Un esempio è in figura 21.

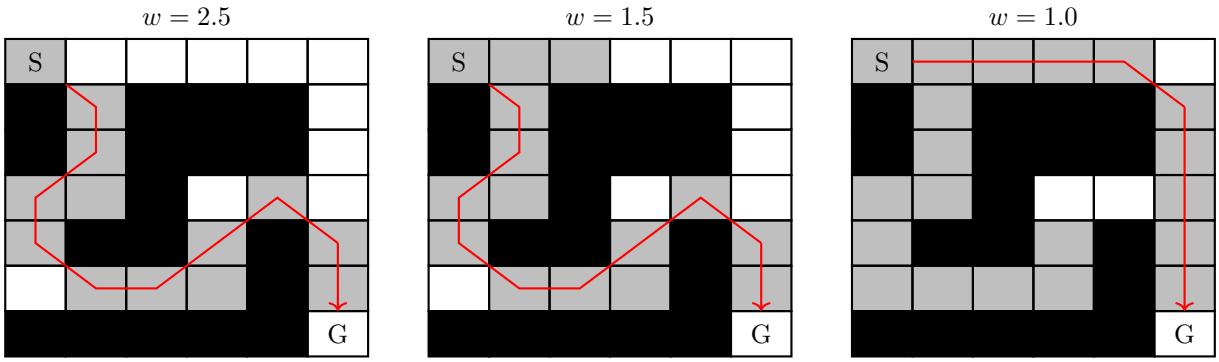


Figure 21: Esempio di applicazione ripetuta di Weighted A*.

Per quanto possa sembrare una buona idea, essa presenta alcuni difetti, ad esempio il valore di alcuni stati rimane il medesimo tra le varie interazioni, ma viene ricalcolato ad ognuna di tali interazioni, sprecando tempo computazionale. Si introduce quindi un algoritmo che ovvi a questa problematica.

10.4.2 ARA*

Per capire ARA* partiamo innanzitutto dall'algoritmo standard di A*, applicato al grafo in figura 22; per trovare tale percorso è stato applicato l'algoritmo 9.

Notiamo che arrivati nello stato finale (Grafo 4), essendo S_3 uno stato non ancora espanso, ovvero appartenente all'insieme OPEN, il suo valore g risulta un upper bound del valore ottimale. Per tutti gli altri stati, già espansi ed appartenenti all'insieme CLOSED, g è già il valore ottimale. A questo punto è possibile computare il percorso per arrivare al goal, conoscendo i valori veri delle distanze g. Con riferimento al paragrafo precedente, nel momento in cui, in un algoritmo Weighted A* si vuole diminuire i pesi, come si possono reutilizzare i risultati visti?

Inseriamo una variabile v per salvare tali risultati intermedi; così che il valore g di ogni stato sia ottenibile da tale v: $g(s') = \min_{s'' \in \text{pred}(s')} v(s'') + c(s'', s')$. Utilizziamo poi tali valori nell'algoritmo ComputerPath (9) rinnovato, ovvero l'algoritmo 10.

Si noti che l'insieme OPEN è un insieme di stati in cui $v(s) > g(s)$, gli stati con tale proprietà saranno definiti *overconsistenti*; gli altri stati avranno invece $v(s) = g(s)$ e saranno definiti stati *consistenti*. Una volta stabiliti i valori v saranno espansi gli stati overconsistenti sulla base del valore $f(s) = g(s) + h(s)$.

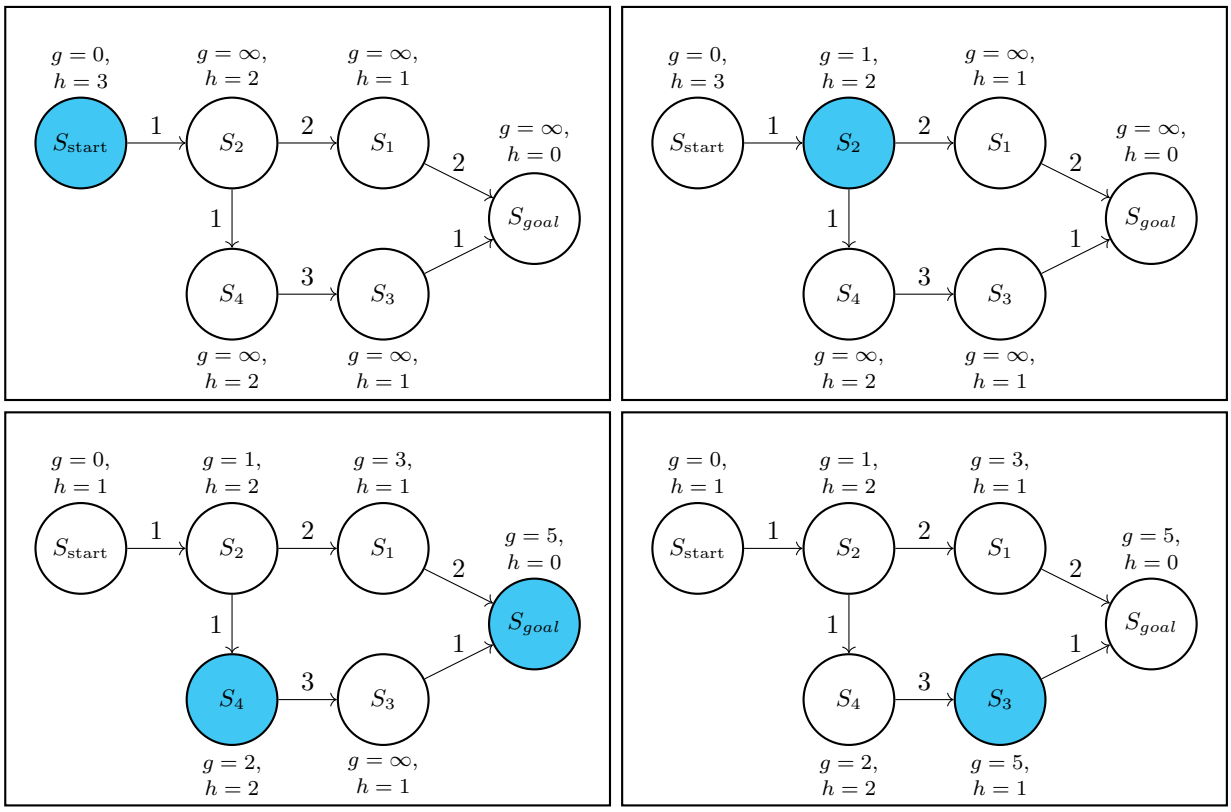


Figure 22: Esempio di grafo e di percorso trovato tramite A*.

Algorithm 9 ComputePath function

```
1: while s_goal is not expanded and OPEN ≠ ∅ do
2:     Remove s with the smallest f(s) = g(s) + h(s) from OPEN;
3:     Insert s into CLOSED;
4:     for all successors s' of s such that s' ∉ CLOSED do
5:         if g(s') > g(s) + c(s, s') then
6:             g(s') = g(s) + c(s, s');
7:             Insert s' into OPEN;
8:         end if
9:     end for
10: end while
```

Per fare sì che a seguito dell'algoritmo 10 vengano reutilizzati i valori v, si introduce la funzione **ComputePathwithReuse** (algoritmo 11). Sarà la combinazione di questi algoritmi a portare al risultato finale, utilizzando l'algoritmo 10 inizialmente e poi per svariate volte l'algoritmo 11.

Algorithm 10 ComputePath function with v-values

```
1: all v-values initially are ∞;
2: while s_goal is not expanded and OPEN ≠ ∅ do
3:     Remove s with the smallest f(s) = g(s) + h(s) from OPEN;
4:     Insert s into CLOSED;
5:     v(s) = g(s);
6:     for all successors s' of s such that s' ∉ CLOSED do
7:         if g(s') > g(s) + c(s, s') then
8:             g(s') = g(s) + c(s, s');
9:             Insert s' into OPEN;
10:        end if
11:    end for
12: end while
```

Algorithm 11 ComputePathwithReuse function

```
1: initialize OPEN with all overconsistent states;
2: while f(s_goal) > min f-value of state ∈ OPEN do
3:     Remove s with the smallest f(s) = g(s) + h(s) from OPEN;
4:     Insert s into CLOSED;
5:     v(s) = g(s);
6:     for all successors s' of s such that s' ∉ CLOSED do
7:         if g(s') > g(s) + c(s, s') then
8:             g(s') = g(s) + c(s, s');
9:             Insert s' into OPEN;
10:        end if
11:    end for
12: end while
```

Una volta che, come abbiamo visto, abbiamo fatto sì che A* reutilizzasse valori già computati, si vuole applicare il medesimo concetto a Weighted A*.

Utilizzando Weighted A*, ora è necessario considerare che i valori $g(s)$ di alcuni stati possano variare. Ciò accade perchè Weighted A* sovrastima maggiormente quanto più grande è il valore ϵ, e ciò porta l'algoritmo a trovare percorsi sub-ottimi. Abbassando il valore di ϵ può accadere che vengano trovati nuovi valori $g(s)$ per uno stato s minori del valore $g(s)$ calcolato in precedenza. Dunque per ogni successore ora sarà necessario ricalcolare il valore $g(s)$ a prescindere per ogni successore, anche se esso è già stato calcolato (e dunque si trova in CLOSED); proprio in questo caso, come già detto, si arriverà ad avere, in uno stato s, una situazione per cui $g(s) < v(s)$ in quanto $g(s)$ è stato decrementato, diventando inconsistente con il valore $v(s)$.

A questo punto, dobbiamo schermare le problematiche che ciò può causare: si introduce un insieme INCONS, che racchiude gli stati inconsistenti. Tali stati hanno necessità di avere un valore di $v(s)$ in linea con il nuovo $g(s)$; e dunque nelle iterazioni successive saranno inserite nella frontiera degli stati OPEN. L'algoritmo ARA* completo effettua questa operazione, come si può vedere nell'algoritmo 13.

La funzione già vista ComputePathWithReuse può essere efficiente solo con gli stati s per cui $v(s) \geq g(s)$, cioè gli overconsistenti o i consistenti. Nella pratica, però possono incombere ostacoli improvvisi che introducono stati underconsistenti, cioè per cui $v(s) < g(s)$. Supponendo per esempio che nella figura 22, l'arco $S_2 \rightarrow S_1$ assuma valore 4 invece di 2, la situazione cambierebbe, rompendo l'integrità della proprietà per cui $g(s') = \min_{s'' \in \text{pred}(s')} v(s'') + c(s'', s')$, essendo cambiato $c(s'', s')$.

Algorithm 12 ComputePathwithReuse function

```
1: function ComputePathWithReuse
2:     while f(s_goal) > min f-value of state ∈ OPEN do
3:         Remove s with the smallest f(s) = g(s) + h(s) from OPEN;
4:         Insert s into CLOSED;
5:         v(s) = g(s);
6:         for all successors s' of s do
7:             if g(s') > g(s) + c(s, s') then
8:                 g(s') = g(s) + c(s, s');
9:                 if s' ∉ CLOSED then
10:                    insert s' into OPEN;
11:                else
12:                    insert s' into INCONS;
13:                end if
14:            end if
15:        end for
16:    end while
17: end function
```

Algorithm 13 Anytime Repairing A*

```
1: set ε to large value;
2: g(s_start) = 0; v-values of all states are set to ∞; OPEN = {s_start};
3: while ε > 1 do
4:     CLOSED = {}; INCONS = {};
5:     ComputePathWithReuse();
6:     publish current ε suboptimal solution;
7:     decrease ε;
8:     initialize OPEN = OPEN ∪ INCONS;
9: end while
```

Dunque, dall'esempio sopracitato, sarà necessario aggiornare il valore $g(S_1) = 5 \neq 3$, portando alla disuguaglianza $v(S_1) < g(S_1)$, visto $v(S_1) = 3$.

A questo punto, per risolvere il problema, è necessario effettuare due passaggi:

- per rendere lo stato consistente o overconsistente si imposta $v(S_1) = \infty$;
- aggiornare tutti i valori delle g dei successori di S_1, nel nostro caso S_{goal}, ed aggiornare $v(s)$ di tali stati come al punto precedente.

10.4.3 D* Lite

D* Lite è un algoritmo di replanning ottimale, più semplice dell'algoritmo D* da cui deriva. Si applica a problemi goal-driven in cui si assume che tutte le celle siano attraversabili (free space assumption): nel momento in cui una mossa è impedita dalla presenza di un ostacolo avviene la ripianificazione, che può tuttavia portare a soluzioni subottime. Rispetto a ARA* si passa da una ricerca forward ad una backward, cioè la ricerca parte dal goal verso la posizione attuale, cambia quindi il significato dell'euristica e della funzione di costo $g(s)$:

$$g^*(s) = \begin{cases} 0 & \text{se } s = s_{goal} \\ \min_{s' \in succ(s)} c(s, s') + g^*(s') & \text{altrimenti.} \end{cases}$$

In questo modo cambia la modalità con cui si aggiornano le varie $g(s)$ quando i sensori rilevano nuovi ostacoli. Infatti i nodi più vicini alla radice, ovvero il goal, rimarrano invariati, fino a raggiungere il

cambiamento rilevato; al contrario, con una ricerca forward, sarebbe stato necessario aggiornare tutti i valori.

L'algoritmo è applicabile anche in forma anytime, facendo varie run ed aggiornando i pesi ϵ similarmente ad ARA*; l'algoritmo è il seguente (algoritmo 14):

Algorithm 14 Anytime D*

```
1: set ε to large value;
2: repeat
3:     ComputePathWithReuse();
4:     publish ε-suboptimal path;
5:     follow the path until map is updated with new sensor information;
6:     update the corresponding edge costs;
7:     set s_start to the current state of the agent;
8:     if significant changes were observed then
9:         increase ε or replan from scratch;
10:    else
11:        decrease ε;
12:    end if
13: until goal is reached
```

10.5 Rapidly Exploring Random Trees (RRT)

Rapidly Exploring Random Trees è una tecnica la cui idea principale è quella di espandere incrementalmentalmente un albero da una configurazione iniziale q_0, ovvero la radice dell'albero.

In questo caso non esiste una frontiera, ma l'albero cresce in continuazione facendo sampling nel C-space; questa tecnica risulta utile quando si ha a che fare con domini in cui la funzione di transizione fra stati non è nota. La probabilità di trovare un cammino tra lo stato iniziale e lo stato goal al tende a 1 all'infinito, cioè quando il numero di sample tende all'infinito.

L'algoritmo di RRT (algoritmo 15) funziona effettuando sampling di uno stato x, trova lo stato x_{near} più vicino appartenente all'albero già costruito e prende uno stato x_{new} tra x_{near} e x raggiungibile tramite un controllo opportuno da x_{near}. Graficamente si può vedere un esempio in figura 23.

Questo è un algoritmo generico, che può essere declinato in vari modi, come illustrato nelle sezioni successive.

Per pianificare un percorso utilizzando RRT si effettua il processo seguente:

1. Si avvia RRT a q_I.
2. Ogni n iterazioni, viene forzato $q_{rand} = q_G$
3. Se si raggiunge q_G, il problema è risolto.

Se venisse continuamente scelto q_G il risultato sarebbe quello di incappare in numerosi ostacoli esplorando lo spazio, sprecando tempo computazionale. In generale sono algoritmi semplici da implementare e per i quali vi è un buon bilanciamento tra esplorazione e greedyness, ma per cui la convergenza in generale è ignota.

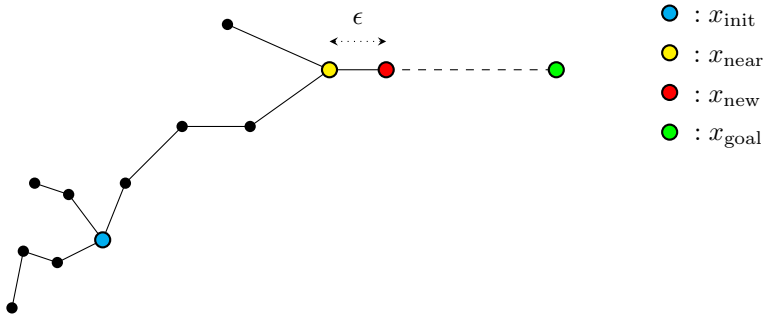


Figure 23: Esempio grafico di RRT.

Algorithm 15 BUILD_RRT(x_{init})

function EXTEND($\mathcal{T}, x$)
    $x_{near} \leftarrow$ NEAREST_NEIGHBOR($x, \mathcal{T}$);
    if NEW_STATE($x, x_{near}, x_{new}, u_{new}$) **then**
        $\mathcal{T}$. add_vertex(x_{new});
        $\mathcal{T}$. add_edge($x_{near}, x_{new}, u_{new}$);
        if $x_{new} = x$ **then**
            return *Reached*;
        else
            return *Advanced*;
        end if
    end if
    return *Trapped*
end function

$\mathcal{T}$. init(x_{init});
for $k = 1$ **to** K **do**
    $x_{rand} \leftarrow$ RANDOM_STATE();
    EXTEND($\mathcal{T}, x_{rand}$);
end for
return $\mathcal{T}$

10.5.1 RRT-Connect

Connect è una variante in cui non si utilizza un punto intermedio x_{new} ma si cerca di raggiungere direttamente x da x_{near}; se questo non è possibile viene rieffettuato sampling.

Algorithm 16 RRT_CONNECT(q_{init}, q_{goal})

```
1: function CONNECT(T, x)
2:     repeat
3:         S ← EXTEND(T, x_rand);
4:     until not (S = Advanced)
5:     return S;
6: end function

7: T_a.init(q_init); T_b.init(q_goal);
8: for k = 1 to K do
9:     q_rand ← RANDOM_CONFIG();
10:    if not EXTEND(T_a, q_rand) = Trapped then
11:        if CONNECT(T_b, q_new) = Reached then
12:            return PATH(T_a, T_b);
13:        end if
14:    end if
15:    SWAP(T_a, T_b)
16: end for
17: return Failure
```

10.5.2 RRT Goal Bias

RRT Goal Bias utilizza un sampling non uniforme che è più propenso verso il goal; si può decidere di applicare questo approccio solo su una porzione delle iterazioni.

10.5.3 RRT-Bidirectional

RRT-Bidirectional ha come idea principale di fare crescere in contemporanea due diversi alberi, uno dallo stato iniziale e l'altro dallo stato goal, finché essi non si incontrano in un punto. Di tanto in tanto anziché effettuare sampling per trovare x lo si impone a goal in modo da vedere se questo sia raggiungibile.

10.5.4 RRT*

RRT* è una variante che prende ispirazione da A*, che tende a soluzioni ottime. L'idea consiste nel connettere x_{new} con il nodo più vicino, esso viene collegato al nodo x_{min} che minimizza la distanza da x_{start} a x_{new}. L'algoritmo si trova sulle slide ma presumo non sia necessario studiarlo.

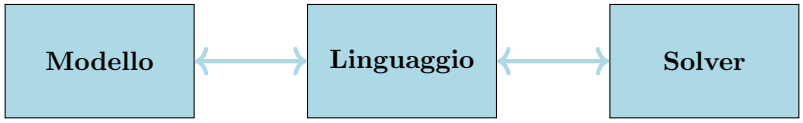
11 Task Planning

L'obiettivo del task planning è capire come il robot organizzi i suoi compiti ad alto livello, senza approfondire come avvengano i compiti più specifici. Le possibili modalità di applicazione del task planning sono principalemnte tre:

- **approccio programming-based**: il piano è predeterminato, ovvero la missione del robot è decisa a priori; questa non è una modalità predittiva, ma è solamente esecutiva;
- **approccio learning based (reinforcement learning)**: vengono utilizzate delle policy, cioè funzioni che guidano verso l'obiettivo, questo approccio può essere utile ma non offre una visione completa in quanto alcune scelte possono avere dei "punti oscuri", cioè non si capisce la motivazione di alcune azioni;
- **approccio model-based (pianificazione automatica)**: in questo approccio si computa un piano a partire da un modello del mondo e da un insieme di azioni che l'agente può compiere.

Noi ci concentreremo principalmente sulla pianificazione automatica; in generale si puo dire che *il planning è il ragionamento dell'azione*, ovvero un processo deliberativo che sceglie ed organizza le azioni sulla base dell'output atteso, per raggiungere alcuni obiettivi nel modo migliore possibile.

Per attuare questa strategia, abbiamo tre ingredienti principali: un **modello** (avente degli stati ed una **funzione di transizione** per una sua rappresentazione, similarmente ad un problema di ricerca), un **linguaggio** per esprimere la modellazione, ed infine un **risolvitore** per risolvere il problema scritto nel determinato linguaggio con un certo modello.



11.1 PDDL

Per rappresentare il modello ed il problema noi utilizziamo esiste il linguaggio PDDL, che attualemente è lo standard de facto per la rappresentazione di questo tipo di modelli, che utilizza la logica del primo ordine. PDDL utilizza due principali file per esprimere il modello ed il problema: le transizioni (azioni) e i fatti (predicati) sono espressi nel *domain file* mentre il mondo con i relativi oggetti, lo stato iniziale ed il goal sono rappresentati nel *model file*.

PDDL è strettamente legato al *classical planning*, ovvero un problema di pianificazione esprimibile in una tupla $\langle \boldsymbol{F}, \boldsymbol{I}, \boldsymbol{G}, \boldsymbol{A} \rangle$, rispettivamente:

- $\boldsymbol{F}$: insieme dei fatti;
- $\boldsymbol{I}$: sottoinsieme contenente fatti iniziali ($I \subseteq F$);
- $\boldsymbol{G}$: stati goal, dati da una formula su F.
- $\boldsymbol{A}$: insieme delle azioni, in cui ogni azione è una coppia $\langle \text{pre}(a), \text{eff}(a) \rangle$, ovvero le precondizioni di una azione affinchè essa si verifichi e gli effetti della stessa azione;

La soluzione del classical planning consiste in un insieme di azioni che portano da I a G. Possiamo vedere il classical planning come l'istanza concreta del problema generico espresso tramite PDDL, cioè appunto il piano da I a G concreto, che si ottiene tramite l'operazione di "grounding" sul problema.

Il termine grounding indica il processo di trasformazione di azioni e predicati "astratti" (contenenti parametri) in istanze concrete (senza parametri) a partire dagli oggetti e dalle costanti specificati nel problema; in altre parole, si passa da operatori e predicati con variabili a azioni e predicati completamente istanziati con i nomi degli oggetti reali.

Ad esempio, data l'azione `move ?x ?y` e gli stati s_1 ed s_2, il grounding porta alle possibili combinazioni:

-move s_1, s_1
-move s_1, s_2
-move s_2, s_1
-move s_2, s_2

L'operazione di grounding è esponenziale in quanto dati n stati e k parametri, la complessità è $\mathcal{O}(n^k)$ (di fatto però essa diminuisce in quanto di fatto vengono poi considerate le azioni possibili, ad esempio moves_1, s_1 è una azione di fatto insensata e quindi normalmente non sarebbe da considerare).

11.1.1 Pianificazione Numerica

Una delle problematiche di PDDL è che esso esprime fatti ma non quantità, quindi ci si chiede come è possibile esprimere quantità numeriche in generale e come sono modellabili. L'idea è quindi di modificare PDDL affinchè possa comprendere le nozioni numeriche.

- I predicati rimangono equivalenti.
- Le **azioni** indicano cosa l'agente può fare e quando, e cosa succede se lo fa, cioè assegna valori numerici, ed assegna precondizioni a partire da formule numeriche.
- Utilizza **funzioni**, variabili di stato numeriche che assumono valori in $\mathcal{Q}$, che possono essere definite in maniera parametrica.
- Agli **stati iniziali** possono essere assegnate variabili numeriche.
- Ai **goal** vengono assegnate condizioni tali per cui essere considerati stati goal.

Questo tipo di PDDL, avente variabili di stato numeriche, si associa al numeric planning, ovvero una tupla $\langle \boldsymbol{F}, \boldsymbol{X}, \boldsymbol{I}, \boldsymbol{G}, \boldsymbol{A} \rangle$:

- $\boldsymbol{F}$: insieme dei fatti, come accade nel classical planning;
- $\boldsymbol{X}$: insieme delle variabili numeriche;
- $\boldsymbol{I}$: sottoinsieme contenente fatti iniziali ($I \subseteq F \times X$);
- $\boldsymbol{A}$: insieme delle azioni, in cui ogni azione è una coppia $\langle \text{pre}(a), \text{eff}(a) \rangle$, come nel classical planning, con l'aggiunta di termini numerici;
- $\boldsymbol{G}$: stati goal, dati da una formula su con termini di preposizioni e numerici.

La soluzione del planning numerico consiste, come nel planning classico, in un insieme di azioni che portano da I a G.

11.2 Problemi di Pianificazione

Nonostante esistano varie modalità di risoluzione dei problemi di pianificazione, noi ci concentreremo sulla ricerca nello spazio degli stati forward, cioè in avanti (dagli stati iniziali verso il goal). La ricerca forward ha alcuni elementi fondamentali:

- viene utilizzato il sistema delle transazioni: due stati s_1 ed s_2 sono connessi se esiste una azione a che porta da s_1 ad s_2;
- come spiegato precedentemente, non viene effettuato il grounding completo, ma vengono considerate solo gli stati e le azioni possibili;
- i grafi che rappresentano lo spazio non vengono forniti in input (necessiterebbero di troppo spazio e sarebbe eccessivamente complesso crearli), bensì vengono generati in maniera procedurale;
- non si vuole visitare uno stato più di una volta, dunque si scartano i cicli.

Naturalmente usare algoritmi di ricerca che esplorino l'intero spazio è troppo costoso e potrebbe anche non finire mai; per questo è necessario direzionare la ricerca utilizzando un'euristica approriata. Nella pianificazione automatica, infatti, si possono derivare euristiche indipendenti dal dominio.

La chiave per ottenere buone euristiche sta nel rilassamento del problema, cioè la risoluzione di un problema più semplice, liberato da alcuni vincoli (ad esempio la rimozione degli effetti negativi), che porti ad una soluzione non esatta, ma che possa per lo meno guidare la ricerca, come succede ad esempio in A*. La vastità dello spazio del problema rilassato è maggiore, e una soluzione rilassata non per forza è una soluzione del problema vero; al contrario, una soluzione al problema completo è anche soluzione del rilassato.

Esistono due principali tipi di rilassamento: interval-based relaxation e subgoaling-based relaxation.

11.2.1 Interval-based Relaxation

L'idea generica è quella di mappare gli stati con intervalli che rappresentano più stati possibili, ed il rilassamento consiste nella continua espansione di tali intervalli.
Ad esempio, dato lo stato $s = \{x = 0, y = 0\}$, esso viene mappato con gli intervalli: $s^+ = \{x = [0,0], y = [0,0]\}$. Quando poi viene applicata un'azione, ad esempio $s + [x+ = 1]$, si ottiene $\{x = [0,1], y = [0,0]\}$. Applicando invece $s + [x = 5]$, si ottiene $\{x = [0,5], y = [0,0]\}$, in quanto gli intervalli sono tutti uniti e non presentano buchi. Naturalmente questa idea comporta di dover definire delle operazioni più o meno complicate sugli intervalli, riportate di seguito:

$$x + y = [x^- + y^-, x^+ + y^+]; (\text{e.g.}[0,10] + [5,10] = [5,20])$$
$$x - y = [x^- - y^+, x^+ - y^-];$$
$$x \times y = [\min(x^-y^-, x^-y^+, x^+y^-, x^+y^+), \max(x^-y^-, x^-y^+, x^+y^-, x^+y^+)];$$
$$x \div y = [\min(x^- \div y^-, x^- \div y^+, x^+ \div y^-, x^+ \div y^+), \max(x^- \div y^-, x^- \div y^+, x^+ \div y^-, x^+ \div y^+)]$$

Dunque, per applicare questi concetti al problema iniziale, vediamo lo stato come un insieme di possibili intervalli per ogni variabile, e per ogni stato, quando si applica un azione, gli effetti sono computati tramite **unione convessa** $\sqcup$ (cioè un intervallo avente il minimo dei minimo ed il massimo tra i massimi degli intervalli che si stanno unendo).

Ne consegue che, dato un problema di planning che noi vogliamo rilassare attraverso gli intervalli, non saremo noi a dover eliminare vincoli specifici, ma la convex union in sè espande gli intervalli dei possibili stati. Dunque gli effetti negativi, citati in precedenza, non vengono tolti a mano, ma è la convex union di per sè a farlo, vista la continua espansione che fornisce l'operazione stessa.

Per utilizzare concretamente questa idea, innanzitutto si costruisce un problema rilassato: dato un problema di pianificazione $\Pi = (s_0, A, G, X)$, $\Pi^+ = (s_0^+, A^+, G^+, X^+)$ è il suo rilassamento interval-based, in cui:

- s_0^+ è l'equivalente dello stato iniziale s_0 del problema iniziale, ma espresso come insieme di intervalli.
- A^+ è sintatticamente equivalente ad A, ma differisce nella semantica, viste le motivazioni seguenti.
  1. Il successore è stabilito via **convex union** $\sqcup$.
  2. Le precondizioni sono soddifatte in modo rilassato: $(\exists v \in [\underline{x}, \bar{x}] : v \models pre(a))$, ovvero data una precondizione che necessitava del valore v in un certo stato, tale valore deve appartenere all'intervallo attuale per soddisfare le precondizioni.
- G^+ sintatticamente equivalente a G, considerando che le precondizioni sono trattate come descritto al punto precedente.
- $X^+ \equiv X$. Le variabili non cambiano, ma le espressioni seguono l'algebra degli intervalli.

Stabilito il problema, bisogna rispondere a due domande.

Reachability Analisys: è possibile raggiungere il goal?

Actual Heuristic Computation: se si, quanto è distante in termini di azioni nel problema rilassato?

Reachability Analisys Per risolvere in maniera efficiente la prima domanda, visto che potrebbe essere possibile dover applicare molte azioni nonostante il rilassamento, si effettuano alcune osservazioni: se una condizione è soddifatta al livello i, lo sarà anche nei livelli j con $j > i$, in quanto le azioni possibili da applicare non decrescono.

Si introduce dunque il criterio maggiormente importante per il numeric planning nella reachability analisys: **asymptotic reachability**. Questo concetto vuole portare le azioni all'infinito, basandosi sul fatto che se una azione è già stata applicata una volta, ed essa ha, per esempio, allargato l'intervallo in modo positivo, questa azione può essere protratta un infinito numero di volte, fino a che il valore assunto non sarà appunto infinito. Vediamo un esempio.

Dato un effetto numerico di una azione, cioè $x := x + f(x)$ con $f(x)$ intervallo, si arriva ad un intervallo del tipo $[m, M]$; constatato questo fatto, M se è positivo arriva, con la continua applicazione dell'azione ad essere $+\infty$, mentre se negativo $-\infty$.

A livello più generale, data una azione, ricavo due azioni possibili, vista $f(x)$ che può essere positiva oppure negativa, che possono essere "trasportate" all'infinito. Per questo si introduce l'insieme dei supporters Ω, di cardinalità doppia rispetto all'insieme delle azioni $\mathbf{A}$ in cui da ogni azione se ne ricavano due "proiettate all'infinito". Tale insieme è la base di partenza per l'algoritmo di reachability analisys, **ARPG** (algoritmo 17).

Algorithm 17 Asymptotic Relaxed Planning Graph (ARPG)

Input: Π^{++}
Output: Is $\mathcal{G}$ reachable?
$\Omega =$ supporters of $\mathcal{A}$
$s^+ = s_0^+$
$\mathcal{S} = \{a \in \Omega : s^+ \models \text{pre}(a)\}$
while $\mathcal{S} \neq \emptyset$ and $s^+ \not\models \mathcal{G}$ **do**
    $s^+ = \text{succ}^+(s^+, \mathcal{S})$
    $\Omega = \Omega \setminus \mathcal{S}$
    $\mathcal{S} = \{a \in \Omega : s^+ \models \text{pre}(a)\}$
end while
return $s^+ \models \mathcal{G}$

In questo algoritmo, inoltre, troviamo la funzione $succ^+$, che calcola gli stati raggiunti dagli stati s^+ con le azioni $\mathcal{S}$, ovvero dalle azioni "asintotiche"; dunque ogni azione dell'insieme dei supporters Ω viene calcolata una sola volta.

Actual Heuristic Computation Ci si domanda a questo punto come sia possibile calcolare le distanze in termini di azioni. Per questo viene introdotto l'algoritmo **AIBR** (algoritmo 18).
Questo algoritmo gode di alcune proprietà.

Termination AIBR termina in un numero finito di passi.

Safeness AIBR ritorna *unsatisfied* solo per problemi effettivamente senza soluzione.

Efficiency AIBR lavora in tempo polinomiale.

Vanno effettuate alcune specifiche sull'algoritmo.

- $\mathcal{A}'$ è un multiset, non un normale insieme: ciò implica che internamente ad $\mathcal{A}'$ possono esservi ripetizioni di elementi (cioè delle azioni possono essere compiute n volte).
- Per azioni *relaxed applicable* si intendono le azioni che sono applicabili a partire da uno stato di intervalli.

Algorithm 18 Compute AIBR Estimate

Input: Π^{++}
Output: Integer
$\mathcal{A}' = \emptyset$
$s = s_0^+$
loop
  for all $a \in \mathcal{A}$ **do**
    if a is *relaxed applicable* in s **then**
      $s = \text{succ}^+(s, a)$
      $\mathcal{A}' = \mathcal{A}' \cup \{a\}$
      if $s \models \mathcal{G}$ **then**
        return $|\mathcal{A}'|$
      end if
    end if
  end for
end loop

- Le azioni non hanno uno specifico ordine, che porta l'algoritmo a sovrastimare pesantemente (infatti l'euristica non è ammissibile); nonostante ciò tutte le azioni sono considerate e solitamente applicate, in quanto è effettuato un ciclo `for all` all'interno di un ciclo `while` (motivo per cui $\mathcal{A}'$ è un multiset: ci sono ripetizioni di azioni).
- Di fatto è utile ricavare non solo $|\mathcal{A}'|$, ma anche le azioni di tale insieme; esso è infatti un buon punto di partenza per effettuare ricerche come breadth-first e varie.

Mostriamo un esempio di utilizzo di AIBR.

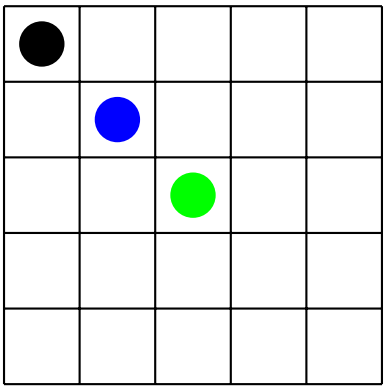


Figure 24: Griglia di esempio per l'attuazione di AIBR.

Rappresentiamo una griglia 5×5, dove alla posizione $(1, 1)$ abbiamo il robot (cerchio nero), alla posizione $(2, 2)$ abbiamo un rubinetto (cerchio blu), ed infine in posizione $(3, 3)$ abbiamo una pianta(cerchio verde). L'obiettivo del robot è quello di innaffiare la pianta, prendendo l'acqua dal rubinetto durante il processo.

Innanzitutto definiamo l'insieme delle azioni possibili $\mathcal{A}$: il movimento nelle quattro direzioni (E east, W west, N nord, S, south), poi il caricamento dell'acqua, L_w ed infine l'innaffiamento P_w. Per dare definizioni formali con precondizioni ed effetti definiamo l'insieme delle variabili X. Nelle variabili utilizziamo i pedici r, t, p, rispettivamente per robot, rubinetto e pianta, e la variabile w che indica l'acqua

che sta portando il robot. Si avrà poi uno stato iniziale I ed un goal G.

$$\begin{aligned}
\mathcal{A} &= \{E, W, N, S, L_w, P_w\} \\
X &= \{x_r, y_r, x_t, y_t, x_p, y_p, w, w_p\} \\
\text{pre}(E) &= (x_r < 4) \\
\text{eff}(E) &= (x_r = x_r + 1) \\
\text{pre}(W) &= (x_r > 0) \\
\text{eff}(W) &= (x_r = x_r - 1) \\
\text{pre}(N) &= (y_r > 0) \\
\text{eff}(N) &= (y_r = y_r - 1) \\
\text{pre}(S) &= (y_r < 4) \\
\text{eff}(S) &= (y_r = y_r + 1) \\
\text{pre}(L_w) &= (x_r = x_t) \wedge (y_r = y_t) \\
\text{eff}(L_w) &= (w = w + 1) \\
\text{pre}(P_w) &= (w > 0) \wedge (x_r = x_p) \wedge (y_r = y_p) \\
\text{eff}(P_w) &= (w = w - 1) \wedge (w_p = w_p + 1) \\
I &= \{x_r = 0, y_r = 0, w = 0, w_p = 0, x_t = 1, y_t = 1, x_p = 2, y_p = 2\} \\
G &= \{w_p = 2\}
\end{aligned}$$

Stabilite le condizioni iniziali cerchiamo di stabilire l'euristica tramite AIBR, ipotizzando di avere già fatto l'analisi asintotica con ARPG e che essa abbia dato risultato positivo. Il primo passo è trasformare I in I^+, che diventerà il primo stato s_0; esso è lo stato iniziale trasformato in intervalli, così da poter utilizzare il rilassamento per intervalli. Per comodità le variabili x_t, y_t, x_p, y_p che sono fisse non verranno riportate. In tale stato nessuna azione sarà stata effettuata, dunque $\mathcal{A}$ sarà vuoto.

$$s_0 = I^+ : \{x_r^+ = [0,0], y_r^+ = [0,0], w^+ = [0,0], w_p^+ = [0,0]\}, \qquad \mathcal{A} = \emptyset$$

Da questo stato, viste le azioni a disposizione e le loro precondizioni, solo due sono quelle possibili da questo stato, cioè E e S. Scegliamo E.

$$s_1 = \{x_r^+ = [0,1], y_r^+ = [0,0], w^+ = [0,0], w_p^+ = [0,0]\}, \qquad \mathcal{A} = \{E\}$$

Ora le x_r possibili sono 0 ed 1 contemporaneamente, in quanto il rilassamento ad intervalli considera possibili entrambe le possibilità. Scegliamo poi di effettuare l'azione S.

$$s_2 = \{x_r^+ = [0,1], y_r^+ = [0,1], w^+ = [0,0], w_p^+ = [0,0]\}, \qquad \mathcal{A} = \{E, S\}$$

Trovandoci nella posizione ideale per riempire l'acqua, in quanto $x_t = 1 \in x_r \wedge y_t = 1 \in y_r$, e dunque le precondizioni sono soddisfatte, scegliamo di riempire di acqua il robot con l'azione L_w.

$$s_3 = \{x_r^+ = [0,1], y_r^+ = [0,1], w^+ = [0,1], w_p^+ = [0,0]\}, \qquad \mathcal{A} = \{E, S, P_w\}$$

Di seguito applichiamo le azioni E, S, P_w, L_w, P_w, ottenendo rispettivamente gli stati s_4, s_5, s_6, s_7, s_8; quest'ultimo stato rappresenterà un goal e quindi è possibile fermarci. Si noti che l'ordine delle azioni è scelto arbitrariamente da noi, e dunque per motivi di brevità dell'esercizio non sono state applicate tutte come prevederebbe AIBR. Se avessimo applicato l'algoritmo alla lettere l'insieme $\mathcal{A}$ sarebbe stato notevolmente più ampio.

$$s_4 = \{x_r^+ = [0,2], y_r^+ = [0,1], w^+ = [0,1], w_p^+ = [0,0]\}, \qquad \mathcal{A} = \{E, S, L_w, E\}$$
$$s_5 = \{x_r^+ = [0,2], y_r^+ = [0,2], w^+ = [0,1], w_p^+ = [0,0]\}, \qquad \mathcal{A} = \{E, S, L_w, E, S\}$$
$$s_6 = \{x_r^+ = [0,2], y_r^+ = [0,2], w^+ = [0,1], w_p^+ = [0,1]\}, \qquad \mathcal{A} = \{E, S, L_w, E, S, P_w\}$$
$$s_7 = \{x_r^+ = [0,2], y_r^+ = [0,2], w^+ = [0,2], w_p^+ = [0,1]\}, \qquad \mathcal{A} = \{E, S, L_w, E, S, P_w, L_w\}$$
$$s_8 = \{x_r^+ = [0,2], y_r^+ = [0,2], w^+ = [0,1], w_p^+ = [0,2]\}, \qquad \mathcal{A} = \{E, S, P_w, E, S, P_w, L_w, P_w\}$$

11.2.2 Subgoal-based Relaxation

In questo rilassamento l'idea è quella di partire dal goal, e vedere tutte le sue precondizioni come diversi sottogoal indipendenti, e reiterare il processo per tali sottogoal fino ad arrivare allo stato iniziale; la direzione del rilassamento è dunque backward e non forward.

Uno dei processi chiave di questo rilassamento è la regressione, ovvero l'operazione che a partire da una azione a e da una formula ψ ritorna una formula ψ', che permette di raggiungere ψ tramite l'azione a. Questa operazione è uno degli ingredienti base perchè permette di passare di sottogoal in sottogoal, scorrendo le azioni backward.

Formalmente, dato il dominio delle formule Γ, quello delle azioni A e quello degli stati S, la regressione *regr* è una funzione $\Gamma \times A \Rightarrow \Gamma$ tale che:

$$\forall \psi \in \Gamma, \forall a \in A, \forall s \in S, s \models regr(\psi, a) \Leftrightarrow s[a] \models \psi$$

L'idea alla base non risulta complessa, se pur la sua applicazione sia semplice nel classical planning ma trovi alcune difficoltà nel planning numerico. Nel planning classico, infatti, data ψ formula atomica, ad esempio `innaffiata(pianta)`, ed una azione a addittiva, ad esempio `annaffia`, la regressione restituisce le precondizioni dell'azione a: $regr(\psi, a) = \text{pre}(a)$; seguendo l'esempio, immaginiamo `soleggiata(pianta)`.

Tale esempio analizza però una singola azione. In generale, nella pianficazione, questo rilassamento va applicato in un percorso tra stati, e dunque tale idea deve essere reiterata, per questo si introduce una formula ricorsiva: la formula introdotta riguarda il classical planning.

$$h^{\max}(s, \psi) := \begin{cases} 0 & \text{if } s \models \psi \\ \min\limits_{a \in \text{ach}(s,\psi)} (h^{\max}(s, \text{pre}(a)) + c(a)) & \text{if } \psi \text{ is a PC(atomic)} \\ \max\limits_{\psi' \in \psi} h^{\max}(s, \psi') & \text{if } \psi \text{ is } \wedge \end{cases}$$

In questa formula la funzione ach denota l'insieme degli *achiever*, ovvero l'insieme delle azioni che a partire da uno stato s permettono di raggiungere la formula atomica ψ (infatti siamo nel caso in cui ψ deve essere atomica). Utilizzando tale insieme con la funzione minimo, otteniamo l'azione che tra tutte permette di raggiungere la condizione ψ al minor costo. Nel caso invece non avessimo una formula atomica, applichiamo ricorsivamente la formula ma con la funzione massimo; in questo modo l'euristica si può dimostrare essere ammissibile, portando ottimalità nel classical planning.

Nel numerical planning bisogna però estendere il concetto espresso: la complessità dell'applicazione nel numerical planning è tale per cui si utilizza una sottocategoria di problemi: i simple numerical planning problems. In questo modo possiamo definire una euristica subgoal-based efficace. Un concetto fondamentale per questa categoria di problemi è la *simple numeric condition*: una condizione numerica si dice semplice se è lineare e tutte le azioni che interagiscono con essa aumentano o diminuiscono di una costante; si parla cioè di una condizione del tipo $x = x + k$ con k costante numerica.

Un problema di numerical planning semplice sarà dunque un problema in cui tutte le condizioni sono semplici. In questa tipologia di problemi la regressione differisce da quella vista prima, infatti nelle formule, ogni variabile è sostituita in modo tale che essa risulti essere stata modificata dall'azione.

Poniamo un semplice esempio: data $\psi : x + y \geq 10$, una azione a con effetto $\text{eff}(a) : x = x + 1$, ed una azione b con effetto $\text{eff}(b) : x = x - 1$, otterremo

$$regr(\psi, a) = (x + 1) + y \geq 10$$
$$regr(\psi, b) = (x - 1) + y \geq 10$$

e dunque:

$$regr(\psi, a) = x + y \geq 9$$
$$regr(\psi, b) = x + y \geq 11$$

Ne consegue che l'azione a è un possibile achiever per la formula ψ, mentre l'azione b non lo è. Tali osservazioni sono **indipendenti** dallo stato in cui le azioni sono applicate. Immaginando di essere nello stato $x = 0, y = 0$, dovremo effettuare 10 volte l'azione a per ottenere la formula ψ.

All'atto pratico questo conteggio può risultare complesso, ma necessario per stabilire l'euristica. Tale conteggio è espresso dalla funzione $\widehat{\text{rep}}(a, \psi, s)$:

$$\widehat{\text{rep}}(a, \psi, s) = \begin{cases} 0 & \text{if } s \models \psi \\ -\dfrac{[\xi_\psi]^s}{N_{\psi,a}} & \text{else if } a \in \text{ach}(\psi) \\ \infty & \text{otherwise} \end{cases}$$

dove, per essere brevi, ξ_ψ e $N_{\psi,a}$ sono due termini complessi e supercazzolati che indicano rispettivamente la valutazione della condizione nello stato s ed il contributo dell'azione su tutte il variabili che sono in ψ.

Questa formula ha però come scopo pricipale quello di indicare come sia possibile contare il numero di azioni a per passare da uno stato s ad una formula desiderata ψ nel simple numerical planning, così da poter ottenere una ulteriore formula ricorsiva, analoga a quella del classical planning.

$$h^{\max}_{\text{hbd}}(s, \psi) \begin{cases} 0 & \text{if } s \models \psi \\ \min\limits_{a \in \text{ach}(s,\psi)} \left(h^{\max}_{\text{hbd}}(s, \text{pre}(a)) + c(a)\right) & \text{if } \psi \in \text{PC} \\ \min\limits_{a \in \text{ach}(s,\psi)} \left(\widehat{\text{rep}}(a, \psi, s) \cdot c(a) + h^{\text{add}}_{\text{hbd}}(\text{pre}(a))\right) & \text{if } \psi \in \text{SC} \\ \max\limits_{\psi' \in \psi} h^{\text{add}}_{\text{hbd}}(s, \psi') & \text{if } \psi \text{ is } \wedge \end{cases}$$

Questa formulazione, contrariamente a quanto accade nei rilassamenti per intervalli, utilizza un meccanismo di scelta dell'azione intelligente, in quanto cerca di scegliere le azioni con il minimo costo nei casi $\psi \in \text{PC}$ (ovvero una formula atomica, con soli valori booleani) e $\psi \in \text{SC}$ (ovvero una formula non solamente atomica, ma che comprende variabili numeriche).

Tale formula non presenta però le stesse caratteristiche della corrispettiva nel classical planning, infatti esistono controesempi che la rendono non ammissibile. Tali controesempi si basano sul fatto che scegliendo una determinata azione come miglior achiever (in quanto minimo) non vengono considerate interazioni positive tra le azioni che partecipano al raggiungimento di una determinata formula.

Per rendere ammissibile tale formula viene cambiata la casistica $\psi \in \text{SC}$, anche se esistono variabili della stessa casistica inammissibili che nella pratica funzionano bene.

Un miglioramento ulteriore sono le *helpful actions*, ovvero azioni che sono ritenuti utili in un certo stato per raggiungere la formula desiderata, cioè delle azioni che sono volte a considerare le varie iterazioni positive tra azioni che solitamente non sono considerate. La variante con helpful actions risulta generalmente più efficace.

12 Task Planning nel Tempo

Spesso, la dimensione temporale è cruciale nell'ambito della pianificazioni, in quanto può essere possibile che certe azioni siano effettuabili solo in determinati lassi di tempo. Risulta dunque la necessità di poter rappresentare vincoli temporali all'interno dei problemi di pianificazione. Sono diversi gli strumenti che danno questa possibilità: Timeline, ANML, ma soprattutto, quelli trattati nel corso, PDDL 2.1 e PDDL+.

12.1 PDDL+

PDDL+ è una versione di PDDL che aggiunge agli elementi già trattati (predicati, azioni, funzioni) due nuove possibilità: i **processi** e gli **eventi**. Fino a questo momento, abbiamo sempre trattato i problemi di planning in maniera centrica rispetto all'agente, senza particolare focus sul mondo esterno; esso però, cambia nel tempo a causa di fattori esterni e a causa di interazioni dell'agente col mondo. Questi due elementi hanno dunque lo scopo di integrare il mondo esterno nel planning.

In particolare, i processi trattano cosa succede nel corso del tempo: per esempio un agente può attivare indirettamente un processo attraverso un' azione, che porta alcuni effetti, oppure un processo potrebbe essere un clock che scandisce il tempo; il processo si protrae poi nel tempo.

Gli eventi, invece, sono cambiamenti causati da particolari condizioni raggiunte, che possono ad esempio cambiare il valore di alcuni predicati.

Vediamo un esempio in cui vogliamo definire il movimento di un mars-rover, che deve raccogliere informazioni in alcuni siti su marte e trasmettere le informazioni raccolte. Il mars rover, tra le azioni a sua disposizione, potrà muoversi, ma tale movimento non sarà definito come semplice azione da un punto x_0 ad un punto x_1, bensì come un movimento nel tempo tra gli stessi punti. Il mars-rover può inoltre collezionare informazioni una volta raggiunto un sito.

```
(:types
    site
)

(:predicates
    (at ?x -site)
    (picture_taken_at ?x -site)
    (sample_taken_at ?x -site)
    (connected ?x -site)
    (recharge_point ?x -site)
    (moving ?x ?y)
    (transmitted)
)

(:functions
    (info ?x -site)
    (battery_level)
    (distance ?x ?y -site)
    (missing_distance ?x ?y -site)
    (info_acquired)
    (info_wanted)
    (min_time ?x -site)
    (max_time ?x -site)
    (clock)
)

(:process clock
    :parameters ()
    :effect (and (increase (clock) (* #t 1.0)))
)
```

```
(:action transmit_data
    :parameters (?x -site)
    :precondition (and
        (> (info_required) (info_wanted))
        (> (clock) (min_time ?x))
        (< (clock) (max_time ?x))
        (at ?x)
    )
    :effect (and (transmitted))
)

(:process moving
    :parameters (?x - site ?y - site)
    :precondition (and
        (moving ?x ?y)
        (> (battery_level) 0)
    )
    :effect (and
        (decrease (missing_distance ?x ?y) (* #t 1.0))
        (decrease (battery_level) (* #t 1.0))
    )
)

(:action start-move
    :parameters (?x - site ?y - site)
    :precondition (and (or (connected ?x ?y) (connected ?y ?x)) (at ?x))
    :effect (and (not (at ?x)) (moving ?x ?y)
        (assign (missing_distance ?x ?y) (/ (distance ?x ?y) 2))
    )
)

(:event end_move
    :parameters (?x - site ?y - site)
    :precondition (and
        (> (missing_distance ?x ?y) -0.01)
        (< (missing_distance ?x ?y) 0.01)
        (moving ?x ?y)
    )
    :effect (and
        (at ?y)
        (not (moving ?x ?y))
    )
)

(:action take_picture
    :parameters (?x - site)
    :precondition (and (at ?x))
    :effect (and
        (picture_taken_at ?x)
        (increase (info) (info ?x))
        (assign (info ?x) 0)
    )
)

(:action collect_sample
    :parameters (?x - site)
```

```
    :precondition (and (at ?x))
    :effect (and (sample_taken_at ?x)
        (increase (info) (info ?x))
        (assign (info ?x) 0)
    )
)

(:action recharge
    :parameters (?x - site)
    :precondition (and
        (recharge_point ?x)
        (at ?x)
    )
    :effect (and (increase (battery_level) 1))
)
```

In questo esempio vediamo che, a fianco delle classici predicati, funzioni ed azioni, abbiamo i processi `clock` e `moving`, con l'evento `end_move`.

Il primo rappresenta lo scorrere del tempo, e la forma `(increase (clock) (* #t 1.0))` indica che il clock aumenta di 1 (più in particolare, stiamo affermando che la derivata dt aumenta di 1, cioè $x_{t+\Delta t} = x_t + 1$ in quanto $dx/dt = 1$).

Il letterale speciale `#t` è usato per indicare il periodo di tempo in cui l'azione viene eseguita; utilizzando `#t` è possibile accedere a punti temporali arbitrari all'interno dell'intervallo di esecuzione (questa frase è un onesto copia ed incolla dal paper originale di PDDL+; per come la interpreto io, `#t` è il tempo scandito internamente dal planner, e durante un processo si ha che il Δt, ovvero `#t` è molto piccolo e viene aumentato o decrementato numerose volte; internamente al planner, dunque `#t` non aumenta volta per volta di 1, ma aumenta di piccoli valori molte volte, per lo meno a livello figurativo da utilizzatore).

Il secondo processo invece si protrae qualora l'agente si stia già muovendo (fattore causato dall'azione `start-move`) e vi sia un livello di batteria sufficiente: la distanza rimanente si riduce continuamente.

Infine l'effetto è dato dal momento in cui la distanza rimanente è pressochè 0, e dunque il movimento è terminato.

Il piano risultante non sarà una sequenza di azioni come nel planning classico, ma sarà dato da una coppia (tempo, azione) in quanto ogni azione necessita di essere eseguita in un momento specifico, che sia consistente con i processi e gli eventi dell'ambiente circostante, fino al raggiungimento del goal; ad esempio `1.0:(start-move s0 s1)` indica che al tempo 1 si compie l'azione di inizio del movimento da s0 ad s1.

12.2 Hybrid Planning via PDDL+

Un sistema di planning ibrido è rappresentabile attraverso una tupla $\Pi = \langle X, F, I, G, A, P, E \rangle$, dove:

- $\boldsymbol{X}$: insieme delle variabili numeriche;
- $\boldsymbol{F}$: insieme dei fatti;
- $\boldsymbol{I}$: sottoinsieme contenente fatti iniziali ($I \subseteq F \times X$);
- $\boldsymbol{G}$: stati goal, dati da una formula;
- $\boldsymbol{A}$: insieme delle azioni, in cui ogni azione è una coppia $\langle \text{pre}(a), \text{eff}(a) \rangle$;
- $\boldsymbol{P}$: insieme dei processi, con la stessa struttura delle azioni e l'aggiunta che essi inducono cambiamenti continui che non sotto il controllo degli agenti;
- $\boldsymbol{E}$: insieme degli eventi, con la stessa struttura di una azione;

Questo problema risulta ulteriormente più difficile da risolvere rispetto al planning numerico, in quanto una serie di variabili ulteriori si aggiungono; anche in questi casi è possibile fare ricerca nello spazio degli stati. In particolare noi vedremo la **State-Space Search through Decision Epoch**:

- si parte dallo stato iniziale,
- da esso, vi sono due possibilità ad ogni punto della ricerca: fare una determinata azione (che provoca una transizione istantanea deterministica da uno stato ad un altro), o non fare nulla (*waiting*), facendo così scorrere il tempo per un determinato Δt in caso ve ne sia necessità (così facendo i processi attivi avanzano),
- si controllano gli eventi prima e dopo ogni transizione,
- la principale idea alla base dell'euristica è di trasformare processi ed eventi in azioni, così da rilassare il problema e ricondurci ad una forma già nota, che sovrastima il problema in generale e su cui è possibile utilizzare ulteriori rilassamenti come ad esempio intervalli o subgoal.

13 Planning con Incertezza

Quando all'effettivo un robot si trova in un ambiente e necessita di pianificare, muoversi, compiere azioni e processi, l'ambiente e le azioni non sono totalmente deterministiche: infatti è possibile incorrere in guasti dell'agente, domini non totalmente osservabili, e ostacoli improvvisi nell'ambiente. L'idea che si vuole implementare è quella di modellare l'incertezza e gestirla.

In generale dobbiamo trattare con due concetti, *incertezza* ed *osservabilità*.

La prima consiste in effetti di azioni che possono essere non deterministici oppure stocastici; nel primo caso si ha che, data una azione, non esistono modelli per qualificare l'output di tale azione, che potrebbe essere vario; nel secondo caso si ha che è nota la distribuzione di probabilità per qualificare l'output.

Per quanto riguarda l'osservabilità, si parla di cosa si può osservare nel mondo: tutto, ovvero si ha una conoscenza completa in ogni momento, parziale, ovvero solo alcuni fattori sono osservabili, oppure nulla, nel senso che è impossibile trarre informazioni.

Si possono dunque classificare le tecniche attuabili a seconda dell'oservabilità e dell'incertezza del problema:

Tipo di incertezza / osservabilità	**Osservabilità totale**	**Osservabilità parziale**	**Osservabilità nulla**
Non deterministico	FOND	POND	Conformant Planning
Stocastico	MDP	POMDP	Conformant Probabilistic Planning

13.1 Full Observability Non Deterministic (FOND)

Definiamo un problema FOND attraverso:

- $\boldsymbol{F}$: insieme di variabili booleane
- $\boldsymbol{S}$: insieme di tutte le possibili combinazioni di F
- $\boldsymbol{A}$: tuple di tipo ⟨name, pre, eff⟩, dove name è una stringa che rappresenta il nome dell'azione, pre è una formula definita sulle variabili F, ed eff è un insieme di assegnamenti,
- $\boldsymbol{G}$: goal, una formula definita sulle variabili F.

Vista la formalizzazione e la natura del problema, non è possibile definire la soluzione come una serie di azioni, essendo tali non deterministiche. La soluzione consiste invece in una policy, ovvero una funzione parziale (in quanto non mappa tutti gli stati ma un sottoinsieme, ciò è indicato dal simbolo $\rightharpoonup$) che che mappa gli stati e le azioni: $\pi : S \rightharpoonup A$.

Esistono due modi di cercare policy.
La prima è una modalità online, detta **reattiva**, dove vengono eseguite le azioni, e vengono confrontate con le azioni aspettate; nel caso ciò non avvenga, si ripianifica se necessario.
La seconda modalità è offline, è detta **proattiva**, e deve tenere conto di tutti i possibili imprevisti; ciò comporta una maggiore quantità e tempo di calcolo necessaria, anche se poi questa policy è definitiva e non viene modificata.

13.1.1 Approccio Reattivo

Affinchè l'approccio reattivo sia attuabile, è necessario chiedersi come sia possibile monitorare l'esecuzione del piano e rilevare imprevisti in modo efficiente. Supponiamo un algoritmo del tipo:
1. → Apply an action.
2. → Observe the environment.
3. → If the state is not how it was planned, replan.
4. → Goal reached? Finish!
5. → Otherwise goto 1.

Questo approccio risulterebbe corretto ma inefficiente, infatti è evidente un che replanning fatto a prescindere non sia efficiente: se infatti fossimo vicini allo stato in cui dovremmo essere e dovessimo fare solo un breve ricalcolo, utilizzeremo molte computazioni inutilmente. In questo caso, possiamo infatti spostare il focus dagli stati esatti agli stati rilevanti per arrivare al goal.

Attraverso la regressione, già vista, infatti è possibile stabilire dei **kernel**, uno strumento necessario a capire quali sono gli stati rilevanti. Visto un piano $\pi = \langle a_0, a_1, \ldots, a_{n_1} \rangle$ ed un goal G, è possibile computare n kernels, ovvero $K = \langle K_0, K_1, \ldots, K_{n-1} \rangle$ tali che $\forall s \in S$:
- $s \models K_0$ iff $s[a_0, a_1, \ldots, a_{n-1}] \models G$
- $s \models K_1$ iff $s[a_1, a_2, \ldots, a_{n-1}] \models G$
- ...
- $s \models K_{n-1}$ iff $s[a_{n-1}] \models G$

Questo indica che uno stato appartiene dunque ad un kernel i, se eseguendo le azioni $i, \ldots, n-1$ della policy π in maniera deterministica , noi raggiungeremmo il goal. I kernel sono chiaramente calcolabili tramite regressione, e risultano utili in quanto, se interrompendo un piano nel punto i, ci troviamo in uno stato $s \in K_i$, allora in realtà non serve eseguire replanning. Come già visto, la regressione fatta nel classical planning è semplice, mentre risulta più complessa qualore la si attui nel numerical planning, anche se è possibile.

Esitono ulteriori tecniche per il replanning, ad esempio LPG-ADAPT, SHERPA, oppure, nel temporal planning, si usa una simple temporal network, dove ad ogni azione eseguita si verifica la consistenza della rete ed in caso si effettua un replan.

13.1.2 Approccio Proattivo

Per quanto riguarda l'approccio offline, esistono diverse tecniche: si può anticipare il replanning per ogni possibile stato (oneroso, poco efficiente), è possibile compilare il problema in un problema SAT, oppure, come approccio che analizzeremo, è possibile utilizzare grafici AND/OR con Policy Space Search.
La ricerca AND/OR utilizza grafi nei quali vi sono due possibili tipi di nodi.

- **Nodi OR**: è possibile scegliere una delle possibili azioni disponibili per arrivare al goal.
- **Nodi AND**: bisogna procedere analizzando tutti i possibili outcome dell'azione, di fatto stiamo protraendo tutti i possibili effetti di una azione.

Una soluzione per questi grafi è rappresentata da un sottoalbero che raggiunge il goal ad ogni foglia, scegliendo una possibile azione per nodo OR, e seguendo tutte le possibili uscite nei nodi AND (esempio in figura 25). Nell'ambito del non determinismo, possiamo dire che questo sottoalbero rappresenta una policy: in ogni stato abbiamo la migliore azione disponibile da intraprendere, e dunque abbiamo una funzione che ad ogni stato associa una azione.

I grafi AND/OR, in generale cercano soluzioni di tipo cyclic strong. Questo tipo di soluzioni è una famiglia di soluzioni cicliche. A livello intuitivo, se avessi un ciclo, si creerebbe un loop infinito per cui da uno stato tornerei ad esso infinite volte pur seguendo la policy, e ciò porterebbe a soluzioni errate.

Di fatto stiamo però facendo ipotesi su un ambiente non deterministico: in una strong cyclic solution infatti, assumiamo che per un limite infinito, ogni azione debba accadere,in quanto nessuna azione ha probabilità di accadere nulla. Ciò avviene in quanto l'ambiente è "fair" e pertanto si vuole indicare che non è avverso, nel senso che vuole evitare che una azione accada, ma al contrario è onesto, e dunque se una azione ha la minima probabilità di accadere, se riptetuta un numero sufficiente di volte, essa accadrà.

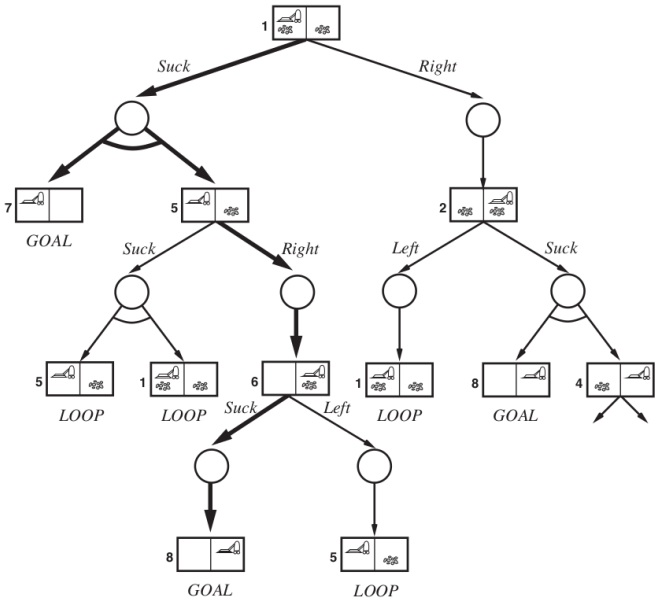


Figure 25: I primi due livelli dell'albero di ricerca per il mondo erratic vacuum del Russel/Norvig. La soluzione trovata è indicata dalle linee in grassetto.

Algorithm 19 AND-OR-GRAPH-SEARCH

```
1: function AND-OR-GRAPH-SEARCH(problem) return a conditional plan, or failure
2:     return OR-SEARCH(problem.Initial-State, problem, [])
3: end function

4: function OR-SEARCH(state, problem, path) return a conditional plan, or failure
5:     if problem.Goal-Test(state) then
6:         return empty plan
7:     else if state is on path then
8:         return failure
9:     end if
10:    for all action in problem.Actions(state) do
11:        plan ← AND-SEARCH(Results(state, action), problem, [state | path])
12:        if plan ≠ failure then
13:            return [action | plan]
14:        end if
15:    end for
16:    return failure
17: end function

18: function AND-SEARCH(states, problem, path) return a conditional plan, or failure
19:    for all s_i in states do
20:        plan_i ← OR-SEARCH(s_i, problem, path)
21:        if plan_i = failure then
22:            return failure
23:        end if
24:    end for
25:    return conditional plan:
26:    [if s_1 then plan_1 else if s_2 then plan_2 else ... else plan_n]
27: end function
```

Questo comporta che la ripetizione di una azione a lungo termine porti l'effetto desiderato. Le cyclic strong solutions sono inoltre più semplici da trovare, per questo utilizzate da molti pianificatori FOND. A livello più formale diciamo che ciò che cerchiamo è una policy closed e proper.

- **Closed**: se e solo se per ogni stato raggiunto dalla policy π esiste almeno una azione ad esso associata.
- **Proper**: se e solo se per ogni stato raggiunto dalla policy π esiste almeno un percorso di stati della policy che porta al goal.

Una policy π è detta **strong** se e solo se è closed, proper e aciclica.
Una policy π è detta **strong cyclic** se e solo se è closed e proper.

Stabiliti i requisiti e la forma della soluzione, è necessario comprendere come sia possibile creare una soluzione. Anche in questo caso la policy è parziale, in quanto solo alcuni stati sono mappati. L'idea è quella di compiere una best-first search sulle policy ammissibili: si ha un insieme OPEN con delle policy, ognuna valutata sulla base di una funzione euristica f che ne valuta il costo. Dall'insieme OPEN si espande la policy con valore f minore, fino al raggiungimento di una policy strong cyclic. Nel caso una policy di questo tipo non sia trovata, si effettua una espansione: viene selezionato uno stato s non ancora raggiunto dalla policy con ogni sua azione possibile. L'algoritmo risultante è l'algoritmo 20.

Algorithm 20 AND*

```
1: π_I := ∅, Open := {π_I}
2: while Open ≠ ∅ do
3:     Remove from Open some policy π with least f(π)
4:     if π is closed and proper then
5:         return π                                          ▷ strong-cyclic
6:     end if
7:     if π is not closed then
8:         Select a state s from Out_∼(π)
9:         for all action a applicable to s do
10:            Insert π' = π ∪ {s ↦ a} in Open
11:        end for
12:    end if
13: end while
14: return ⊥                                                 ▷ unsolvable task
```

Risulta chiaro che in questo algoritmo uno dei punti chiave è trovare una euristica adeguata per esprimere la funzione f. La soluzione è fingere che il problema sia deterministico: ciò rappresenta un rilassamento in quanto, una policy che ha costo n in un ambiente deterministico avrà un costo $m \geq n$ in un ambiente non deterministico, risultando dunque ammissibile.

Possono esistere vari casi di policy: quella che ottimizza il best case, e quella che ottimizza il worst case.

13.1.3 Continuos Replanning

Continuos replanning consiste in un mix di tecniche attive e proattive. Viene utilizzato un planner, detto PRP che fa una ricerca completa. Non è un argomento affrontato nel dettaglio, l'idea alla base è comunque la **All Outcome Determinization (AOC)**. L'idea alla base è che, data una azione a, che per esempio può portare sia in uno stato s_0 che in uno stato s_1, e dunque: $eff(a) = \{\{s_0\}, \{s_1\}\}$, con AOC si arrivi al risultato per cui l'azione a si sdoppia in a_1 ed a_2, con effetti rispettivamente $eff(a_1) = \{s_0\}$ ed $eff(a_2) = \{s_1\}$. Il planner sceglierà poi l'outcome: è normale che se non esiste una soluzione con questo rilassamento, nemmeno il problema originale avrà soluzione.

13.2 Conformant Planning

In questa tipologia di problemi oltre al non determinismo si ha osservabilità nulla: di fatto abbiamo incertezza nello stato iniziale, non abbimo sensori utili ad indicare lo stato in cui ci troviamo. Ne consegue

che il piano risultante non è più una policy (non sapendo in quale stato ci troviamo non sapremmo che azione applicare), bensì una sequenza di azioni. Nel conformant planning deterministico, invece si hanno diverse assunzioni:
- lo stato iniziale è noto solo parzialmente;
- gli effetti son deterministici ma condizionali: gli effetti dell'azione a non sono semplicemente un insieme fisso di addizioni e cancellazioni atomiche, ma sono abbinati a delle condizioni, cioè se la condizione c è vera, allora si applica l'effetto e, cioè $eff(a) = \{c \Rightarrow e\}$;
– non è possibile effettuare osservazioni.

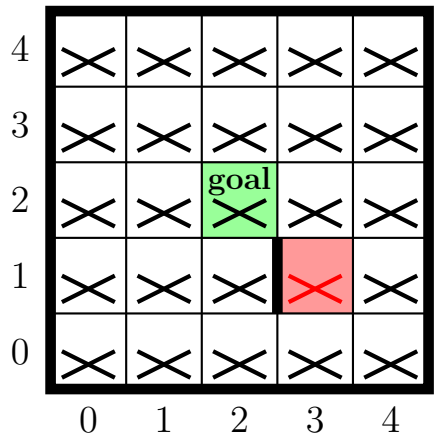


Figure 26: Esempio di problema di deterministic conformant planning

Un esempio di problema è quello in figura 26, dove è possibile muoversi nelle quattro direzioni, e si vuole raggiungere il goal G in posizione (2,2). Si ha un muro tra la posizione (1,2) e (1,3) che impedisce di andare oltre, inoltre la stessa casella (1,3) indica la morte dell'agente e dunque necessita di essere evitata. Una volta colpito un muro non si può sfondare, e dunque muoversi in direzione del muro implica che l'agente rimanga fermo. Chiaramente, la posizione iniziale non è nota.

La soluzione corrisponde all sequenza di azioni:
- 5 volte destra (finendo nelle possibili coordinate $\{(0,4),(1,2),(2,4),(3,4),(4,4)\}$),
- 5 volte nord (finendo nelle possibili coordinate $\{(4,2),(4,4)\}$),
- 2 due volte destra (finendo sicuramente in $\{(4,4)\}$),
- stabilita una posizione unica valida si può arrivare al goal: 2 volte sinistra e 2 volte giù.

Ci si domanda in generale come sia possibile risolvere problemi di questo tipo. Una possibile soluzione è il planning attraverso i **belief state**: esiste un algoritmo detto **CEGAR (CounterExample Guided Abstraction Refinement)**. Esso consiste in:

1. Partiamo dall'insieme di tutti i possibili stati iniziali, definendo un'astrazione su tali stati e comprimendo insieme più stati concreti in "blocchi" indistinti.
2. Formuliamo un problema di classical planning che funzioni per tali stati.
3. Troviamo un piano π per il problema di classical planning.
4. Proviamo ad eseguire il piano per ogni stato.
5. Troviamo (eventualmente) uno stato del belief per il quale π non è valido, esso è un controesempio.
6. Se non ci sono controesempi, il problema è risolto.
7. Computiamo un nuovo piano.
8. Ripetiamo.

L'algoritmo completo è l'algoritmo 21.

Algorithm 21 Conformant Planning Algorithm.

1: **input:** planning problem $\mathbb{P}$
2: $\mathcal{B} := \bot$ ▷ Initial sample of the belief state
3: $\pi := \varepsilon$ ▷ Empty plan—is a plan for belief state $\bot$
4: **loop**
5: **if** π is a solution for $\mathbb{P}$ **then**
6: **return** π ▷ Plan found
7: **end if**
8: Let q be a counter-example
9: $\mathcal{B} := \mathcal{B} \vee q$
10: Compute new plan π for $\mathbb{P}_{\mathcal{B}}$
11: **if** no such π exists **then**
12: **return** no plan exists
13: **end if**
14: **end loop**

14 Pianificazione con MDP

Nel caso sia disponibile una dstribuzione probabilistica, il problema diventa un **Markov Decision Process (MDP)**. In questo caso, il problema di ottimizzazione viene descritto dalla tupla $\langle S, s_0, s^*, A, Pr, C\rangle$, dove:

- S è l'insieme degli stati,
- s_0 è lo stato iniziale,
- s^* è l'insieme di stati goal,
- A è l'insieme delle azioni,
- Pr è la funzione di transizione, che descrive la probabilità di passare da uno stato s ad uno stato s' in seguito all'esecuzione di una certa azione a, per cui è esprimibile come $Pr : S \times A \times S \to [0, 1]$, dove $Pr(s' \mid s, a)$ è la probabilità di passare dallo stato s allo stato s' eseguendo l'azione a,
- C è la funzione di costo, che assegna un costo a ciascuna azione eseguita in uno stato, per cui $C : S \times A \to \mathbb{R}_{\geq 0}$, dove $C(s, a)$ è il costo dell'azione a nello stato s.

Come nel caso dei problemi FOND, anche gli MDP hanno una policy (parziale) come soluzione: $\pi : S \rightharpoonup A$. La policy ottima è la policy che minimizza il costo atteso, ovvero:

$$\pi^* = \arg\min_{\pi} E[\sum C(s, a) \text{ to reach } s^*].$$

A questo punto ci si chiede come sia possibile calcolare tale policy e, più in generale, come calcolare il costo atteso di una policy.

Policy Evaluation è la tecnica che risponde a questa domanda: si utilizza una funzione $V(s)$ che rappresenta il costo atteso per raggiungere lo stato goal s^* a partire dallo stato s, similarmente ad una funzione euristica. La funzione di valore viene calcolata in modo ricorsivo, utlizzando la funzione di costo C, la funzione di transizione Pr, la policy π, che indica l'azione da eseguire in uno stato s con la notazione $\pi(s)$:

$$V^\pi(s) = C(s, \pi(s)) + \sum_{s' \in S} Pr(s'|s, \pi(s)) \cdot V^\pi(s')$$

Per trovare in generale il valore atteso di uno stato s, $V(s)$, si utilizza **Value Iteration**, un processo la cui idea è, a partire dagli stati goal (con $V(s) = 0$), dalla conoscenza della funzione di costo e delle transizioni, di calcolare i valori V attesi ripetendo il processo fino a convergenza:

$$V(s) = \begin{cases} 0 & \text{if } s \in s^* \\ \min_{a \in A(s)} C(s, a) + \sum_{s' \in S} Pr(s'|s, a)V(s') & \text{otherwise} \end{cases}$$

L'idea si applica attraverso l'algoritmo 22.

I valori V dunque si riaggiornano continuamente, essendo che si parte dai valori vicini al goal e che poi, considerando tutte le azioni possibili, si occupa lo spazio intero. Gli stati più vicini al goal arriveranno più velocemente a valori veritieri.

Tale dispendiosità del processo lo rende poco applicabile, per cui si introduce l'idea di **Policy Iteration**: si parte con una policy iniziale (si può trovare ad esempio tramite una ricerca greedy o simili), e con i valori V associati a tale policy (possono essere calcolati tramite value iteration, ma senza dover scegliere l'azione a costo minimo, ma scegliendo l'azione dettata dalla policy, così si riduce la complessità dell'algoritmo da $\mathcal{O}(as^2)$ ad $\mathcal{O}(s^2)$) si cerca un modo di migliorare la policy. Si introduce quindi la funzione Q, che indica il costo atteso per raggiungere il goal a partire da un certo stato e da una certa azione:

$$Q^\pi(s, a) = C(s, a) + \sum_{s' \in S} Pr(s'|s, a) \cdot V^\pi(s')$$

Algorithm 22 Value Iteration

Input: Value function stored in vector V, with $V(s) = 0$ for goal states s
repeat
  flag := **true**
  for each non-goal state s **do**
    new-value := $\min_{a \in A(s)} \left[c(a, s) + \sum_{s' \in S} P_a(s'|s) V(s') \right]$
    if $|V(s) - \text{new-value}| \geq \epsilon$ **then**
      $V(s)$:= new-value
      flag := **false**
    end if
  end for
until flag = **true**

Algorithm 23 Policy Iteration

Starts with proper policy $\pi(s)$;
repeat
  Compute V^π; ▷ Value Iteration
  Store V^π in vector V; ▷ $V \leftarrow V^\pi(s) = c(s, \pi(s)) + \sum_{s'} P(s'|s, \pi(s)) V^\pi(s')$
  Let $\pi' = \pi$ be the new policy initialized to π;
  Let change = false;
  for all state $s \in S$ **do**
    for all action $a \in A(s)$ **do**
      $Q(s, a) = C(s, a) + \sum_{s' \in S} Pr(s'|s, a) \cdot V(s')$;
    end for
    Let $Q = min_{a \in A(s)} Q(s, a)$;
    if $Q < V(s)$ **then**
      $\pi'(s) = \arg\min_{a \in A(s)} Q(s, a)$;
      change = true;
    end if
  end for
  if change = true **then**
    $\pi = \pi'$;
  end if
until change = false
return π

14.1 Reinforcement Learning

Il **Reinforcement Learning** è un approccio per risolvere MDP simile a quello per la risoluzione di MDP goal-based, ma che utilizza reward invece di costi e massimizzazione invece di minimizzazione. In questo caso un MDP è definito dalla tupla $\langle S, s_0, s^*, A, T, R \rangle$

- S è l'insieme degli stati,
- s_0 è lo stato iniziale,
- s^* un possibile insieme di stati finali,
- A è l'insieme delle azioni,
- T è la funzione di transizione, che descrive la probabilità di passare da uno stato s ad uno stato s' in seguito all'esecuzione di una certa azione a, per cui è esprimibile come $T : S \times A \times S \to [0, 1]$, dove $Pr(s' \mid s, a)$ è la probabilità di passare dallo stato s allo stato s' eseguendo l'azione a,

- R è la funzione di reward, che assegna un reward a ciascuna azione eseguita in uno stato, per cui $R : S \times A \times S \to \mathbb{R}_{\geq 0}$, dove $R(s, a, s')$ è il reward dell'azione a nello stato s per arrivare in s'.

Inoltre, è necessario inserire un valore di discount $\gamma \in [0, 1]$ che, dato un percorso di stati e azioni fa diminuire esponenzialmente l'utilità U:

$$U([s_0, a_0, s_1, a_1, s_2, \ldots]) = R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \ldots$$

Visto questo cambio della tipologia di problema, vediamo come è possibile riscrivere le funzioni V e Q.

$$V(s) = \max_{a \in A(s)} \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma V(s')]$$

$$Q(s, a) = \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma \max_{a' \in A(s')} Q(s', a')]$$

Nel preambolo abbiamo sempre considerato di conoscere a priori le funzione T ed R, ma nella realtà questo non è quasi mai possibile. Per questo dobbiamo esporre nuove tecniche.

14.1.1 Model-based Reinforcement Learning

In questo caso l'idea è quella di imparare un modello basandoci sull'esperienza dell'agente, scoprendo dunque le funzione R e T attraverso l'esperienza. Imparate tali funzioni, è possibile usare le formule viste in precedenza.
Nel reinforcement learning model-based è necessario ripetere per un certo numero delle azioni e verificarne sia l'esito che il reward.

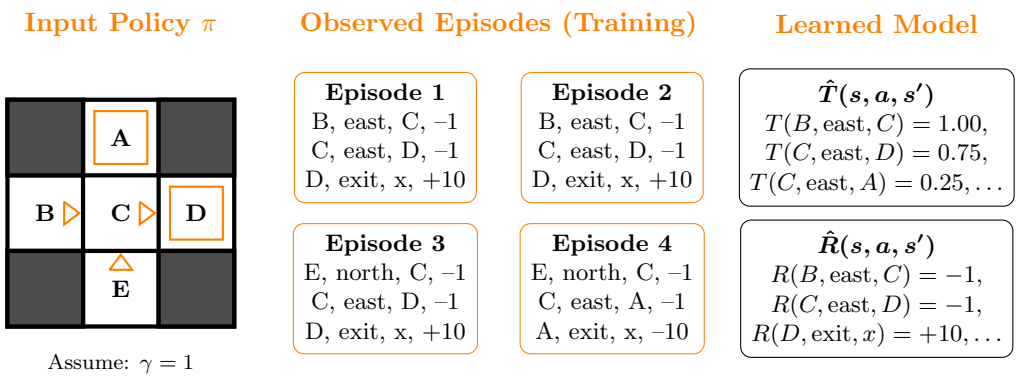


Figure 27: Esempio di apprendimento model-based utilizzando l'esperienza.

In figura 27 abbiamo un esempio: la funzione di transizione T per l'azione east nel punto C verso il punto D, ovvero $T(C, \text{east}, D)$ ha probabilità 0.75 in quanto applicando l'azione east nel punto C siamo finiti tre volte in D ed una volta in A, tanto che tale punto ha valore: $T(C, \text{east}, A) = 0.25$. Un ragionamento analogo è applicabile per la funzione R.

Questo approccio risulta semplice e pratico, oltre che adattabile in molti contesti, anche se è difficilmente applicabile in spazi di grandi dimensioni, in quanto salvare ed utilizzare tuple stato-azione per ogni stato per un numero elevato di volte, risulta dispendioso.

14.1.2 Model-Free Learning

In questo caso l'obiettivo è diverso, non si vuole imparare il modello, ma si vogliono apprendere direttamente i valori attesi degli stati V e di Q. In questo caso si parla di *direct evaluation*, in quanto viene scelta una policy, e successivamente le azioni sono eseguite dall'agente, che apprende dirattemente.

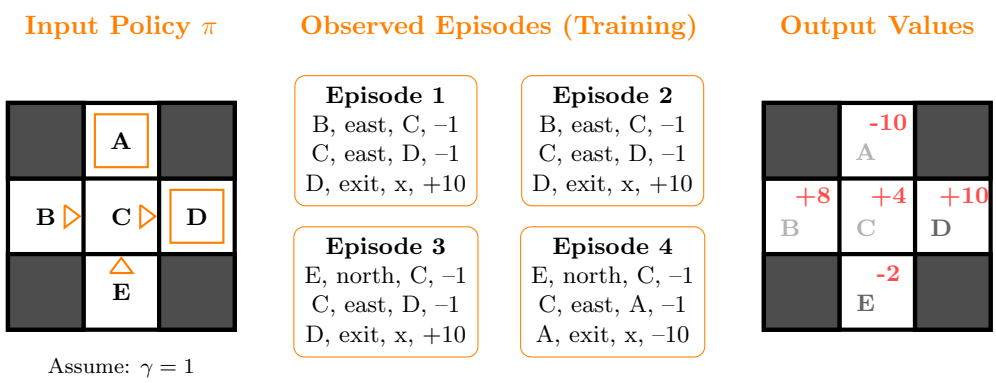


Figure 28: Esempio di direct evaluation.

Come si può vedere in figura 28 ad ogni stato è associato un certo reward a seguito di una esplorazione avvenuta per 4 episodi. Ogni azione effettuata in D ha avuto reward 10, quindi la media ottenuta per D ha valore 10; lo stesso vale per lo stato A. Invece B ed E, se pur la policy voglia portarli entrambi in C, hanno valori diversi, questo è dato dal fatto che il reward atteso per lo stato E è $V(E) = [(-1-1+10)+(-1-1-10)]/2 = -2$, mentre $V(B) = [(-1-1+10)+(-1-1+10)]/2 = 8$. Questo risultato deriva dall'esperienza dell'agente, che nel caso 4 ha ricevuto un reward negativo, pur seguendo la policy, perchè l'azione east in C lo ha portato nello stato A.
Con un apprendimento model-based, chiaramente, se le funzioni $T(B, \text{east}, C)$ e $T(E, \text{north}, C)$ avessero la stessa probabilità (nel nostro esempio sarebbe uguale e pari ad 1) i valori V avrebbero avuto lo stesso valore.

Questo approccio risulta molto semplice ed efficace, ma purtroppo non tiene conto delle connessioni tra stati (motivo per cui $V(E) \neq V(B)$) e richiede che gli stati siano imparati separatamente attraverso l'esecuzione pratica, il che richiede un tempo notevole.

14.1.3 Sample-based Policy Evaluation

Per porre rimedio alle problematiche degli approcci al problema precedenti, possiamo notare che la policy evaluation rimane uno strumento utile, in quanto esplora le connessioni tra stati completamente:

$$V_0^\pi(s) = 0$$
$$V_{k+1}^\pi(s) \leftarrow \sum_{s' \in S} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')].$$

Il problema di questo approccio è la mancata conoscenza delle funzioni T ed R.

Il concetto chiave che può risolvere questo problema è il sampling: se noi infatti provassimo a seguire la policy per un determinato numero di volte, riusciremmo ad estrarre i valori attesi degli stati V osservando i reward ottenuti in determinati stati seguendo una policy e contando le transizioni avvenute in uno stato verso un altro cercando di seguire la suddetta policy.

In particolare, risulta molto efficace effettuare un continuo aggiornamento dei valori V basato sull' esperienza, piuttosto di una serie di prove nel seguire la policy, che richiederebbe un tempo notevole. Definiamo un sample di $V(s)$ ed un conseguente aggiornamento.

$$\text{Sample of } V(s): \quad sample_{V(s)^\pi} = R(s, \pi(s), s') + \gamma V^\pi(s')$$
$$\text{Update of } V(s): \quad V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + (\alpha)sample.$$

Queste equazioni rappresentanpo una media esponenziale sul valore atteso $V(s)$ seguendo la policy π. Tale tipologia di aggiornamento permette infatti di far sì che la funzione V pesi, tramite il peso α gli ultimi sample, dando maggiore importanza ad essi (che si presume essere più precisi) piuttosto che ai primi. Inoltre α può anche essere impostato come fattore decrescente per arrivare ad una convergenza.

Questo processo è detto **temporal difference learning**. Vediamo un esempio di applicazione in figura 29, simile a quelli visti in precedenza.

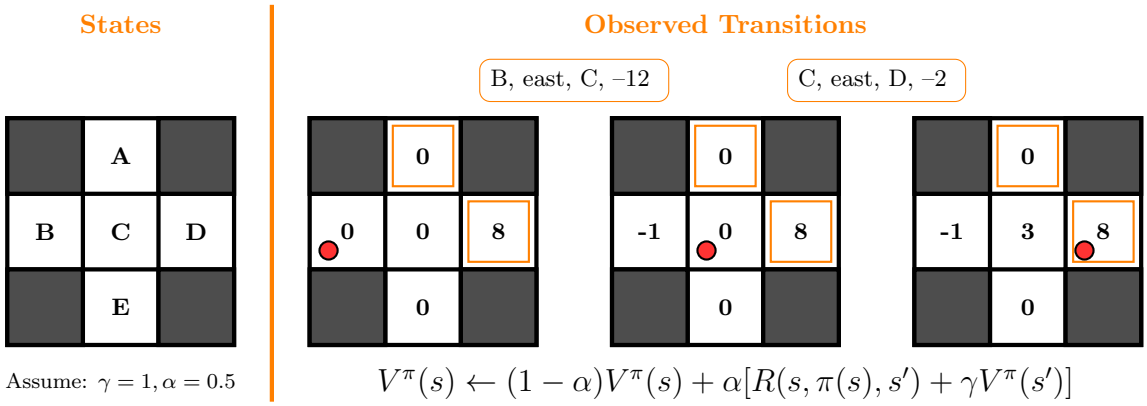


Figure 29: Esempio di applicazione di temporal difference learning.

Con l'approccio di temporal difference learning ci stiamo avvicinando alla soluzione del problema, ma rimane ancora un aspetto fondamentale da considerare: la policy in questo caso rimane predefinita, è necessario trovare un modo di aggiornarla, così da sfruttare al meglio i valori V trovati.

L'idea per ovviare alla problematica del temporal difference learning è quella di utilizzare i valori attesi Q invece di V, incorporando anche le azioni con questo metodo (**q-learning**). Applichiamo dunque un ragionamento analogo a quello effettuato in precedenza; a partire da:

$$Q_0^\pi(s,a) = 0$$
$$Q_{k+1}^\pi(s,a) \leftarrow \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma \max_{a' \in A(s')} Q_k(s',a')].$$

Gli step per imparare una Q-value $Q(s,a)$ sono i seguenti.

- Ricezione di un sample, ovvero una tupla $\langle s, a, s', r \rangle$, a partire da un vecchio valore $Q(s,a)$.
- Calcolare un nuovo sample: $sample = R(s,a,s') + \gamma \max_{a' \in A(s')} Q(s',a')$.
- Effettuare la media on going: $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)sample$.

Il q-learning ha la notevole proprietà di convergere ad ottimalità; di fatto però essa è complicata da raggiungere. Bisogna infatti esplorare abbastanza lo spazio, diminuire il learning rate (ma non troppo velocemente) e trovare un buon compromesso tra exploration ed exploitation, ovvero tra la strada già percorsa numerose volte e nuovi percorsi che potrebbero potenzialmente portare ad errori.

L'esplorazione può avvenire in diversi modi, ma uno particolarmente efficace sono le *exploration functions*. L'idea alla base è quella di esplorare aree la cui bontà non è ancora stata stabilita, e per cui eventalmente è necessario bloccare l'esplorazione. Data infatti l'utilità stimata u ed il contatore di visiste di uno stato n, si può utilizzare la funzione $f(u,n) = u + k/n$, con k costante. Un update che sfrutta questa caratteristica sarà dunque il calcolo del sample:

$$sample = R(s,a,s') + \gamma \max_{a' \in A(s')} f(Q(s',a'), N(s',a'))$$

con $N(s,a)$ numero di volte in cui il Q-state $Q(s,a)$ è stato visitato. Si può notare che il possibile incremento "bonus" per la possibile esplorazione viene propagato a ritroso verso gli stati precedenti.