

## Pianificazione discreta

In questa parte del corso si studia il problema della pianificazione discreta che viene utilizzato come caso introduttivo al problema generale della pianificazione. Nella pianificazione discreta infatti si ha, nella maggior parte dei casi, uno spazio di stato finito (nel caso peggiore numerabile). Per caratterizzare i problemi di pianificazione discreta non sono quindi necessari modelli geometrici o equazioni differenziali.

Il problema della pianificazione delle traiettorie viene formalizzato come problema di ricerca di cammini ammissibili su un grafo. Si illustreranno dapprima diversi metodi di ricerca di soluzioni ammissibili, in cui si è interessati a raggiungere la configurazione desiderata. Successivamente verranno descritti metodi per la ricerca di soluzioni ottime.

Fonte: “Planning Algorithm”, Steve M. LaValle, Cambridge University Press, 2006.

## Applicazione: Pianificazione del moto

Tra i metodi classici per la pianificazione del moto di robot in ambienti con ostacoli ci sono i metodi basati su una **roadmap** tra cui quelli basati sul **grafo di visibilità** (visibility graph methods).

In una formulazione semplificata, la pianificazione si fa nel piano, gli ostacoli sono poligonali, il robot é puntiforme o al più poligonale con orientazione costante. Detto  $C_{free}$  lo spazio di lavoro privo di ostacoli e quindi percorribile dal robot il metodo si basa sul teorema in cui si dimostra che esiste un cammino (semi-)libero da ostacoli tra ogni coppia di configurazioni  $q_{init}$ ,  $q_{goal}$  se e soltanto se esiste una spezzata contenuta strettamente in  $C_{free}$  con i vertici estremi in  $q_{init}$  e  $q_{goal}$  e gli altri vertici coincidenti con quelli degli ostacoli.

Una semplice conseguenza del teorema è che i cammini liberi da ostacoli si possono ricercare tra i segmenti che uniscono  $q_{init}$  e  $q_{goal}$  e i vertici degli ostacoli che siano “visibili” tra loro.

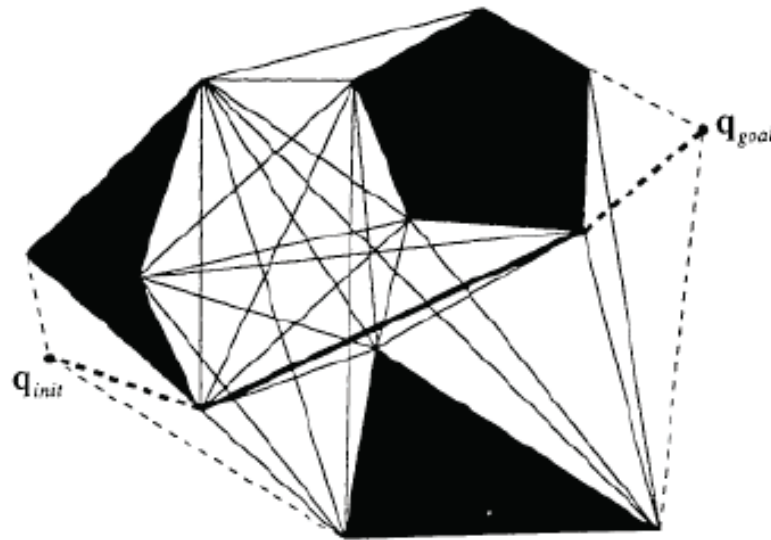


Figura 6: Grafo di visibilità

E' possibile quindi definire un grafo non orientato,  $G = (V, E)$ , con insieme dei vertici  $V$  coincidente con l'insieme formato da  $q_{init}$  e  $q_{goal}$  e tutti i vertici degli ostacoli poligonali. Si definiscono come archi i segmenti tra i nodi in  $V$  che sono lati dei poligoni o giacciono in  $C_{free}$ .

Si noti che il grafo si costruisce per la maggior parte senza  $q_{init}$  e  $q_{goal}$  che possono essere selezionati di volta in volta andando a modificare solo in parte  $V$

ed  $E$ . L'obiettivo è quello di determinare un cammino sul grafo da  $q_{init}$  a  $q_{goal}$ . Inoltre assegnando ad ogni arco il costo pari alla lunghezza del segmento che esso rappresenta, è possibile chiedersi quale sia il cammino di lunghezza minima tra le configurazioni iniziale e finale.

Sono possibili delle semplificazioni al grafo:

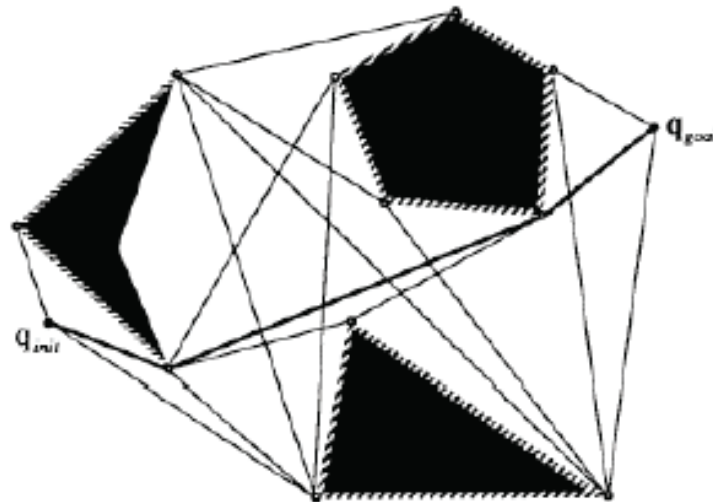


Figura 7: Grafo di visibilità ridotto

## Formalizzazione del problema

Sia  $\mathcal{X}$  lo *spazio di stato* discreto, cioè numerabile o anche finito. Sia  $\mathcal{U}$  lo *spazio dei controlli* ammissibili che in generale può dipendere dallo stato particolare in cui si trova il sistema. In tal caso lo spazio dei controlli associato ad uno stato viene indicato con  $\mathcal{U}(x)$  mentre lo spazio dei controlli risulta  $U = \cup_{x \in \mathcal{X}} \mathcal{U}(x)$ . Sia  $f : \mathcal{X} \times \mathcal{U}(x) \rightarrow \mathcal{X}$  la *funzione di transizione degli stati*, i.e. ogni controllo  $u$  applicato ad uno stato  $x$  produce un nuovo stato  $x' = f(x, u)$ . Sia infine  $\mathcal{X}_G \subset \mathcal{X}$  lo *spazio degli stati di arrivo*.

Lo scopo degli algoritmi di pianificazione è quindi quello di determinare una sequenza finita di controlli che trasformano uno stato iniziale  $x_I$  in uno stato in  $\mathcal{X}_G$ .

Spesso è conveniente modellizzare il problema della pianificazione ammissibile come il problema della ricerca di cammini su un grafo orientato. Si considerino infatti gli stati in  $\mathcal{X}$  come nodi di un grafo, un arco tra gli stati  $x$  e  $x'$  esiste nel grafo se e soltanto se esiste  $u \in \mathcal{U}(x)$  tale che  $x' = f(x, u)$ . Lo stato di partenza e gli stati di arrivo sono dei particolari nodi del grafo. Il problema della pianificazione ammissibile diventa quindi il problema di determinare se esiste un cammino sul grafo a partire dal nodo iniziale sino ai nodi finali.

## Esempio di pianificazione discreta: il labirinto

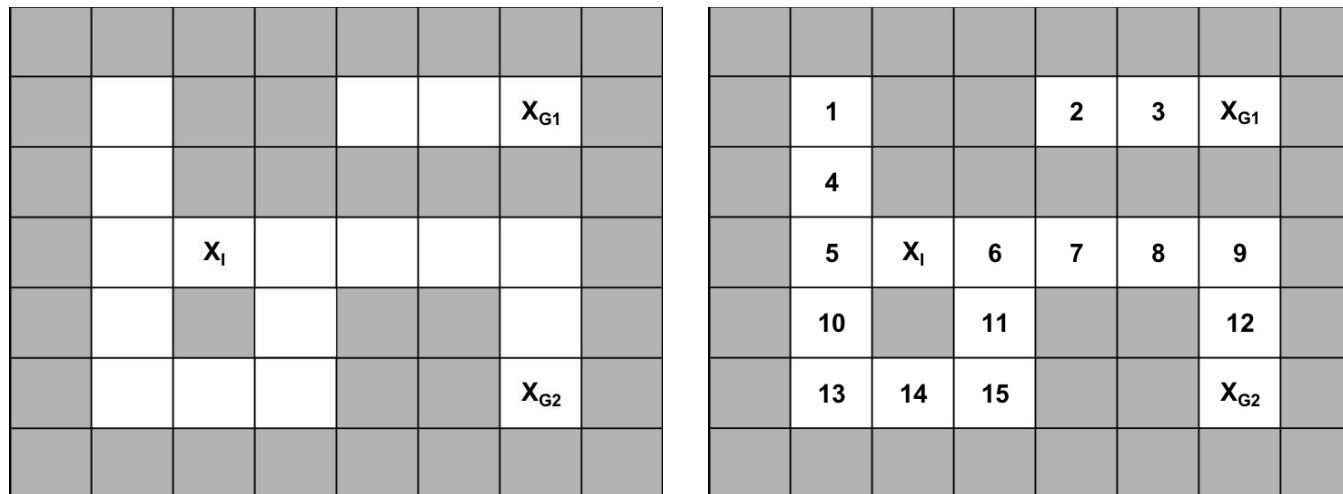


Figura 8: Esempio di pianificazione discreta

Si consideri il labirinto riportato in figura 3 per il quale si vuole trovare una traiettoria dallo stato iniziale  $x_I$  sino allo stato finale  $x_{G_1}$  o allo stato finale  $x_{G_2}$ . Si procede col numerare tutti gli stati del problema.

Ogni stato del sistema viene rappresentato, come riportato in figura 4, come un nodo del grafo. Infine si determinano gli archi che appartengono al grafo.

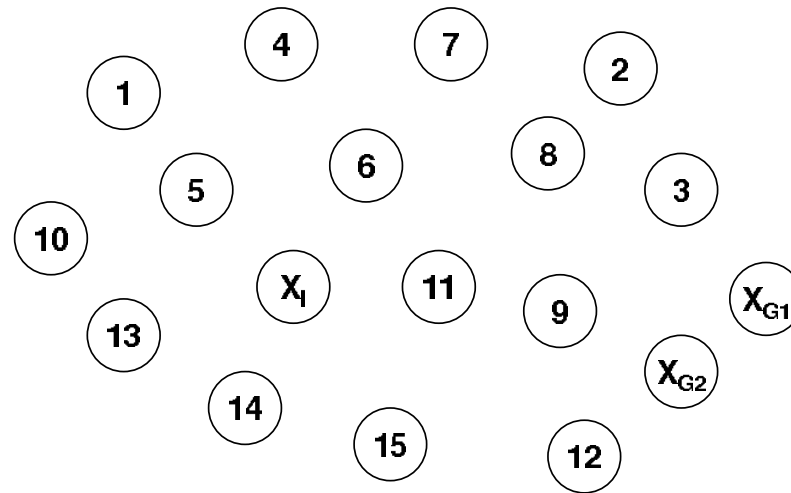


Figura 9: Ad ogni stato discreto si associa un nodo del grafo

Per l'esempio considerato, il grafo associato al problema della pianificazione discreta ammissibile è riportato in figura 5.



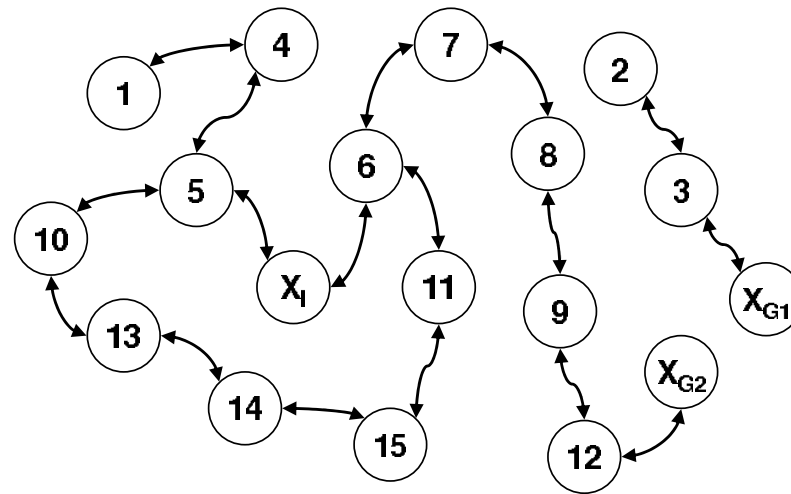


Figura 10: Si determinano quali sono gli archi presenti nel grafo

Si noti che non esiste una traiettoria ammissibile che porti lo stato  $x_I$  nello stato  $x_{G_1}$ , così come non esiste un cammino sul grafo che connetta i rispettivi nodi. Lo stato  $x_{G_2}$  è invece raggiungibile tramite due diverse traiettorie, così come esistono due cammini distinti sul grafo che connettono il nodo  $x_I$  con il nodo  $x_{G_2}$ .

## Ricerca di traiettorie ammissibili

Presentiamo ora alcuni metodi di ricerca su grafi per trovare la soluzione al problema delle traiettorie ammissibili. Ogni algoritmo di ricerca sul grafo deve essere *sistematico*. Se il grafo è di dimensione finita questo vuol dire che l'algoritmo deve visitare ogni stato raggiungibile e quindi deve essere in grado di determinare in tempo finito se esiste o meno una soluzione. Per essere sistematico l'algoritmo deve tenere traccia degli stati già visitati in modo tale da evitare di visitare ciclicamente gli stessi stati.

Nel caso di grafi infiniti si richiede agli algoritmi di ricerca che se la soluzione esista questa venga trovata in tempo finito altrimenti si accetta che l'algoritmo possa procedere per un tempo infinito.

## Metodi di ricerca in avanti

Lo schema generale per gli algoritmi di ricerca su grafo risulta

### Algoritmo di ricerca in avanti

```
1 Q.Insert( $x_I$ )
2 while Q not empty do
3      $x \leftarrow$  Q.GetFirst()
4     if  $x \in \mathcal{X}_G$ 
5         return SUCCESS
6     forall  $u \in \mathcal{U}(x)$ 
7          $x' \leftarrow f(x, u)$ 
8         if  $x'$  not visited
9             Mark  $x'$  as visited
10            Q.Insert( $x'$ )
11        else
12            Resolve duplicate  $x'$ 
13 return FAILURE
```

Tale algoritmo è soltanto in grado di determinare se una soluzione esiste o meno. Se si vuole anche ottenere una traiettoria ammissibile è necessario associare allo stato  $x'$  lo stato genitore  $x$ .

Durante la ricerca nel grafo ci sono tre diversi tipi di stato:

**Non visitato:** Stato che non è ancora stato visitato. Inizialmente ogni stato, con l'esclusione di  $x_I$ , è di questo tipo.

**Morto:** Stato che è stato visitato e per il quale ogni possibile stato successivo è stato visitato. Lo stato successivo di  $x$  è uno stato  $x'$  per cui esiste un ingresso  $u \in \mathcal{U}(x)$  tale che  $x' = f(x, u)$ .

**Vivo:** Stato che è già stato visitato ma ancora non tutti gli stati successivi lo sono stati. Inizialmente l'unico stato vivo è  $x_I$ .

L'insieme degli stati vivi viene ordinato secondo una coda di priorità  $Q$ . L'unica differenza sostanziale tra i vari algoritmi di ricerca è la funzione utilizzata per ordinare la coda  $Q$ . La coda più utilizzata è quella FIFO (First-In First-Out) per la quale lo stato che è stato inserito per primo nella coda è anche il primo ad essere scelto quando  $Q.GetFirst()$  viene chiamata.

## **Algoritmi Breadth First**

Con la scelta di tipo FIFO per la coda  $Q$ , si ottengono algoritmi di tipo “breadth first” in quanto si vanno ad esplorare tutti nodi di un certo livello di profondità prima di andare ad esaminare nodi di profondità più elevata. In altre parole si procede in modo uniforme nell’esplorare il grafo in profondità. Inoltre, con questa politica di selezione dei nodi di  $Q$ , la prima soluzione trovata è quella che usa il minor numero di passi dal nodo origine.

## **Algoritmi Depth First**

Una scelta di tipo LIFO (Last-In First-Out) si ha invece una esplorazione del grafo che procede in profondità a partire dal nodo iniziale. Si scelgono questo tipo di algoritmi se dall’applicazione particolare sembra più efficiente andare a cercare la soluzione in cammini particolarmente lunghi.

Alcuni algoritmi richiedono di calcolare un certo costo e associarlo ad ogni stato. Se uno stesso stato viene raggiunto più volte il costo potrebbe essere modificato. Questo costo può essere utilizzato per ordinare la coda  $Q$  oppure per consentire la ricostruzione della traiettoria una volta che l'algoritmo è completato. Un algoritmo di questo tipo è l'algoritmo di Dijkstra.

## Algoritmo di Dijkstra

Un algoritmo che consente di trovare sia le traiettorie che le traiettorie ottime è l'algoritmo di Dijkstra. Questo algoritmo consente la ricerca di cammini minimi su grafi a partire da un nodo sorgente ed è una particolare forma di programmazione dinamica.

Si supponga che, nella rappresentazione a grafo del problema della pianificazione del moto, ad ogni arco  $e \in E$  sia associato un costo  $l(e)$  non negativo. Tale costo rappresenta il costo per applicare l'azione  $u$  che porta uno stato  $x$  in  $x' = f(x, u)$ . Per questo motivo il costo  $l(e)$  si rappresenta anche come  $l(x, u)$ . Il costo totale del cammino sul grafo risulta quindi pari alla somma dei costi sugli archi che a partire dal nodo iniziale portano a quello finale.

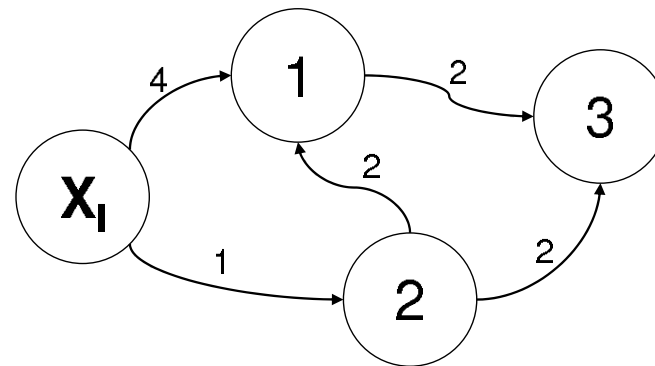
La coda di priorità  $Q$  è ordinata secondo una funzione  $C : \mathcal{X} \rightarrow [0, \infty]$  detta *cost-to-come*. Per ogni stato  $x$ ,  $C^*(x)$  viene detta *cost-to-come* ottima di cammini che partono dallo stato iniziale  $x_I$  e arrivano sino a  $x$ . Ovviamente, il costo ottimo si ottiene considerando il minimo tra i costi di tutti i cammini che connettono  $x_I$  con  $x$ .

Il *cost-to-come* viene calcolato incrementalmente durante l'esecuzione dell'algoritmo di ricerca. Inizialmente si ha che  $C^*(x_I) = 0$ . Sia  $x' = f(x, u)$  allora  $C(x') = C^*(x) + l(e)$ , dove  $e$  è l'arco che connette il nodo  $x$  con il nodo  $x'$ . In questo caso  $C(x')$  rappresenta il *cost-to-come* conosciuto per  $x'$  che non necessariamente è il valore ottimo. Quello che può succedere è infatti che, come previsto dalla riga 12 dell'algoritmo generale di ricerca in avanti, il nodo  $x'$  è già in  $Q$  e viene raggiunto attraverso un nuovo cammino a costo minore. In tal caso il costo  $C(x')$  viene aggiornato e  $Q$  viene riordinato di conseguenza.



Una volta che il nodo  $x'$  viene restituito dalla funzione  $Q.GetFirst()$  allora diventa un nodo morto e per induzione è possibile far vedere che non è possibile raggiungere il nodo a costo minore del corrente valore  $C(x')$ . Infatti, per induzione, il caso base è dato da  $C^*(x_I) = 0$  e supponiamo che tutti i nodi morti hanno associato il proprio cost-to-come ottimo che non può più essere modificato. Sia  $x$  il nodo restituito dalla funzione  $Q.GetFirst()$  vogliamo far vedere che il valore  $C(x)$  è ottimo. Si consideri infatti un altro cammino che prima di arrivare ad  $x$  passi attraverso  $\tilde{x}$ . Se  $\tilde{x}$  è un nodo morto allora il cammino è già stato preso in considerazione e il suo costo è già stato considerato nel calcolo di  $C(x)$ . Se  $\tilde{x}$  non è morto ha comunque un costo  $C(\tilde{x}) \geq C(x)$  perchè non è stato restituito dalla funzione  $Q.GetFirst()$ . In questo caso, essendo il costo dell'arco tra  $\tilde{x}$  e  $x$  positivo si ha che il cammino per arrivare in  $x$  attraverso  $\tilde{x}$  ha un costo più elevato di  $C(x)$ . Concludendo se il nodo  $x$  è morto  $C(x) = C^*(x)$ .

**Esempio 11.** *Si consideri il grafo riportato in figura 6.*



*Figura 11: Determinare i cammini minimi a partire da  $X_I$ .*

*Inizialmente  $Q = \{x_I\}$  e  $C^*(x_I) = 0$ .*

*I nodi successori di  $x_I$  sono 1 e 2 che hanno associato un cost-to-come pari a  $C(1) = 0 + 4 = 4$  e  $C(2) = 0 + 1 = 1$ , da cui  $Q = \{2, 1\}$ .*

*Il nodo da esaminare è quindi 2 i cui successori sono 1 e 3 mentre il costo del nodo 3 risulta  $C(3) = C(2) + 2 = 3$  il costo di 1 va aggiornato in quanto il costo attuale è 4 mentre, al passo corrente,  $C(1) = C(2) + 2 = 3$ .*

*Avendo esaminato tutti i nodi figli, si elimina 2 dalla coda e si pone  $C^*(2) = 1$ .*

*Si ha ora che  $Q = \{1, 3\}$  e si esamina il nodo 1 il cui unico successore è 3 per il quale non si deve aggiornare il costo in quanto risulta maggiore del costo attuale, i.e.  $C(3) = C(1) + 1 = 4 > 3$ .*

*Si elimina 1 dalla coda e si pone  $C^*(1) = 3$ .*

*Infine  $Q = \{3\}$  e quindi  $C^*(3) = 3$ . Si sono così ottenuti tutti i cammini minimi a partire da  $x_I$ .*

**Esempio 12.** Si consideri il grafo riportato in figura 7.

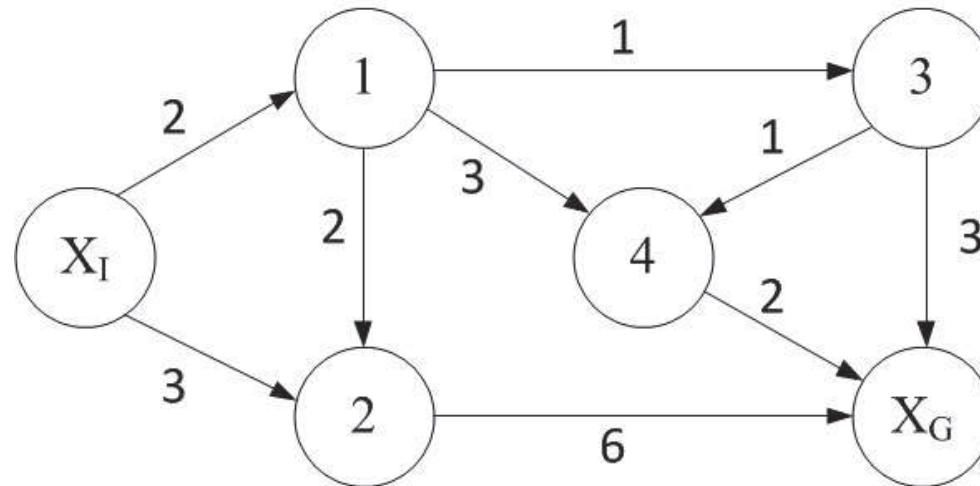


Figura 12: Determinare i cammini minimi a partire da  $X_I$ .

Inizialmente  $Q = \{x_I\}$  e  $C^*(x_I) = 0$ .

I nodi successori di  $x_I$  sono 1 e 2 che hanno associato un cost-to-come pari a  $C(1) = 2$  e  $C(2) = 3$ , da cui  $Q = \{1, 2\}$  e il cammino attuale per arrivare ai nodi 1 e 2 passa attraverso  $X_I$ . Pertanto si pone  $p(1) = p(2) = X_I$  (nodi genitori).

*Il nodo da esaminare è quindi 1 i cui successori sono 2, 3 e 4. Il nodo 2 è un nodo vivo quindi già visitato ma il costo trovato in questo passo ( $C(2) = 2 + 2 > 3$ ) è più elevato di quello noto in precedenza e quindi non viene aggiornato.  $C(3) = 2 + 1 = 3$  e  $C(4) = 2 + 3 = 5$ . Si toglie il nodo 1 dalla coda e si inseriscono i nuovi nodi in base al costo da cui  $Q = \{2, 3, 4\}$ . Si pone inoltre  $C^*(1) = 2$  e  $p(3) = p(4) = 1$ .*

*Si esamina ora il nodo 2 il cui unico successore è  $X_G$ ,  $C(X_G) = 3 + 6 = 9$ ,  $C^*(2) = 3$  e  $Q = \{3, 4, X_G\}$  con  $p(X_G) = 2$ .*

*Si considera il nodo 3 con successori 4 e  $X_G$ . Il nodo 4 è già stato visitato e il costo attuale  $C(4) = 3 + 1 = 4 < 5$  è minore di quello precedentemente trovato, si aggiorna quindi il nodo genitore  $p(4) = 3$ . Anche il nodo  $X_G$  è stato visitato e ora ha un costo minore  $C(X_G) = 3 + 3 = 6 < 9$ , si aggiorna quindi anche il nodo genitore  $p(X_G) = 3$ . Infine  $C^*(3) = 3$  e  $Q = \{4, X_G\}$ .*

*Si esamina il nodo 4 con successore  $X_G$ . Il nodo  $X_G$  è già stato visitato e con un costo pari a quello corrente  $C(X_G) = 4 + 2 = 6$ . In questo caso, non è necessario aggiornare costi o genitori. Si ottiene  $C^*(4) = 4$  e  $Q = \{X_G\}$ .*

*Come ultimo passo si ottiene  $X_G$  e quindi  $C^*(X_G) = 6$  e il cammino minimo da  $X_I$  a  $X_G$  passa attraverso 1, 3 e 4.*

*per il quale non si deve aggiornare il costo in quanto risulta maggiore del costo attuale, i.e.  $C(3) = C(1) + 1 = 4 > 3$ .*

*Si elimina 1 dalla coda e si pone  $C^*(1) = 3$ .*

*Infine  $Q = \{3\}$  e quindi  $C^*(3) = 3$ . Si sono così ottenuti tutti i cammini minimi a partire da  $x_I$ .*

## Algoritmo A-Star

L'algoritmo  $A^*$  è una variante della programmazione dinamica che tenta di ridurre il numero totale di stati esplorati incorporando una stima euristica del costo per raggiungere lo stato finale da un dato stato. Sia  $C(x)$  il cost-to-come da  $x_I$  a  $x$  e  $G(x)$  il cost-to-go da  $x$  a qualche stato in  $\mathcal{X}_G$  (stati di arrivo). Sebbene  $C^*(x)$  si possa calcolare in modo incrementale tramite la programmazione dinamica, non ci sono possibilità di conoscere in anticipo il valore del cost-to-go ottimo  $G^*$ . È comunque spesso possibile ottenere una buona stima inferiore di  $G^*$ , l'obiettivo è quello di trovare una stima inferiore ( $\hat{G}^*(x)$ ) che sia più vicina possibile a  $G^*$ .

La differenza tra l'algoritmo  $A^*$  e Dijkstra sta nella funzione utilizzata per ordinare la coda  $Q$ . Nell'algoritmo  $A^*$  si usa la somma  $C(x') + \hat{G}^*(x')$ , pertanto la coda di priorità è ordinata in base alle stime del costo ottimo tra  $x_I$  e  $\mathcal{X}_G$ . Se effettivamente  $\hat{G}^*(x)$  è una stima inferiore del cost-to-go ottimo per ogni  $x \in \mathcal{X}$  allora l'algoritmo  $A^*$  trova cammini minimi. Se  $\hat{G}^*(x) = 0$  per ogni  $x \in \mathcal{X}$  allora l'algoritmo  $A^*$  coincide con l'algoritmo di Dijkstra.

Per l'esempio del labirinto una stima inferiore del costo si ha considerando una indicizzazione delle caselle sotto forma di matrice. Siano rispettivamente  $ij$  e  $hk$  gli indici di  $x$  e  $x_G$ , la stima inferiore della distanza tra  $x$  e  $x_G$  è data da  $|i - h| + |j - k|$  (percorso minimo se non ci fossero i muri nel labirinto).



## Altri metodi di ricerca generali

Analizziamo ora due altri particolari algoritmi di ricerca, un metodo di ricerca all'indietro e un approccio bidirezionale.

### Algoritmo di ricerca all'indietro

Si consideri il caso in cui ci sia un solo stato di arrivo  $x_G$ , può essere più efficiente procedere all'indietro a partire da  $x_G$  per arrivare a  $x_I$ .

Sia  $\mathcal{U}^{-1} = \{(x, u) | x \in \mathcal{X}, u \in \mathcal{U}\}$  l'insieme di tutte le coppie stato-ingresso, questo può anche essere visto come il dominio della funzione di transizione  $f$ . Sia invece  $\mathcal{U}^{-1}(x') = \{(x, u) \in \mathcal{U}^{-1} | x' = f(x, u)\}$ . Per semplicità denotiamo con  $u^{-1}$  una coppia stato-ingresso in qualche  $\mathcal{U}^{-1}(x')$ . Per ogni  $u^{-1} \in \mathcal{U}^{-1}(x')$  esiste un unico stato  $x \in \mathcal{X}$ . Sia  $f^{-1}$  una *funzione di transizione degli stati all'indietro*, possiamo scrivere  $x = f^{-1}(x', u^{-1})$ . Lo schema per questo algoritmo è simile a quello proposto in 11.

**Ricerca all'indietro**

```

1 Q.Insert( $x_G$ )
2 while Q not empty do
3      $x' \leftarrow$  Q.GetFirst()
4     if  $x = x_I$ 
5         return SUCCESS
6     forall  $u^{-1} \in \mathcal{U}^{-1}(x)$ 
7          $x \leftarrow f^{-1}(x, u^{-1})$ 
8         if  $x$  not visited
9             Mark  $x$  as visited
10            Q.Insert( $x$ )
11        else
12            Resolve duplicate  $x$ 
13 return FAILURE

```

**Algoritmo bidirezionale**

Negli algoritmi bidirezionali un albero viene generato a partire dallo stato iniziale  $x_I$  e uno a partire dallo stato finale  $x_G$ . La ricerca termina con successo quando i due alberi si incontrano, mentre fallisce se una delle code di priorità si esaurisce. Lo schema di questi algoritmi è la combinazione dei due schemi già riportati.

## Pianificazione ottima discreta

Consideriamo ora il caso in cui non si è semplicemente interessati a trovare una traiettoria ammissibile ma si vuole determinare la traiettoria ammissibile che ottimizza un dato criterio come il tempo di percorrenza o l'energia consumata. Per poter estendere la formalizzazione descritta precedentemente devono essere introdotti altri concetti:

- 1 un indice associato al passo della pianificazione;
- 2 una funzione costo che rappresenta il costo accumulato durante l'esecuzione della traiettoria;
- 3 una azione di terminazione dell'algoritmo.

La programmazione dinamica risulta essere un ottimo strumento per cercare di minimizzare il costo della manovra corrente cercando però di mitigare i possibili costi (indesiderati) elevati futuri. Si applica a sistemi con dinamica discreta e con funzioni costo additive nel tempo.

## Programmazione Dinamica

Sistema dinamico  $x_{k+1} = f_k(x_k, u_k, w_k)$ ,  $k = 0, \dots, N - 1$ ,  $u_k$  controllo,  $w_k$  disturbo o rumore e  $N$  orizzonte temporale.

Il costo totale in  $N$  passi è:

$$g(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k, w_k)$$

dove  $g_k(x_k, u_k, w_k)$  è il costo del passo  $k$ -esimo.

### Problema base:

Si consideri il sistema  $x_{k+1} = f_k(x_k, u_k, w_k)$ ,  $k = 0, \dots, N - 1$ ,  $x_k \in S_k$ , con  $u_k \in C_k$  controllo e  $w_k \in D_k$  disturbo. Il controllo può dipendere dallo stato  $u_k \in U_k(x_k) \subseteq C_k$ . I disturbi  $w_k$  sono caratterizzati da una distribuzione di probabilità  $P_k(\cdot | x_k, u_k)$  che è indipendente da  $w_0, \dots, w_{k-1}$ .

La legge di controllo è una sequenza di funzioni (politiche ammissibili)

$\pi = \{\mu_0, \dots, \mu_{N-1}\}$  tale che  $u_k = \mu_k(x_k) \in U_k(x_k) \subseteq C_k$  e

$x_{k+1} = f_k(x_k, \mu_k(x_k), w_k)$ .

Il costo atteso è  $J_\pi(x_0) = E_{w_k, k=0, \dots, N-1} \{g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k)\}$ .

La politica ottima  $\pi^*$ :  $J_{\pi^*}(x_0) = \min_{\pi \in \Pi} J_{\pi}(x_0)$  dove  $\Pi$  è l'insieme delle politiche ammissibili.  $J^*$  è la funzione costo ottima.

Minimizzazione in anello aperto: selezionare i controlli  $u_0, \dots, u_{N-1}$  all'istante iniziale.

Minimizzazione in anello chiuso: selezionare una politica  $\mu_0, \dots, \mu_{N-1}$  che applica il controllo  $u_k = \mu(x_k)$  all'istante  $k$  in base alla conoscenza dello stato corrente  $x_k$ .

## Principio di ottimalità (Bellman 1957)

“Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.”

“Una politica ottima ha la proprietà che, in un certo stato del sistema, essa è indipendente dal modo con il quale lo stato è stato generato, cioè le decisioni rimanenti devono costituire una politica ottima rispetto allo stato risultante dalle decisioni precedenti”

Sia  $\pi^* = \{\mu_0^*, \dots, \mu_{N-1}^*\}$  una politica ottima per il problema base e si supponga che utilizzando  $\pi^*$  si attraversi con probabilità non nulla lo stato  $x_i$  al tempo  $i$ . Si consideri il sotto-problema che consiste nel voler minimizzare il “cost-to-go” dallo stato  $x_i$  da  $i$  ad  $N$ :

$$E \left\{ g_N(x_N) + \sum_{k=i}^{N-1} g_k(x_k, \mu_k(x_k), w_k) \right\}.$$

La politica (troncata)  $\{\mu_i^*, \dots, \mu_{N-1}^*\}$  è ottima per il sottoproblema.

Una diretta conseguenza è che la strategia ottima si può costruire procedendo a ritroso ed estendendo una strategia ottima per il sotto-problema di “coda” dei passi precedente.

### Proposizione

Per ogni stato iniziale  $x_0$  il valore  $J^*(x_0)$  del problema base è uguale a  $J_0(x_0)$  ottenuto nell'ultimo passo della seguente procedura iterativa che procede a ritroso nel tempo da  $N - 1$  a 0:

$$\begin{aligned}
 J_N(x_N) &= g_N(x_N) \\
 J_k(x_k) &= \min_{u_k \in U_k(x_k)} E_{w_k} \{g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k))\} \\
 k &= N - 1, \dots, 0.
 \end{aligned}$$

Se  $u_k^* = \mu_k^*(x_k)$  minimizza il funzionale per ogni  $x_k$  e  $k$  allora la strategia  $\pi^* = \{\mu_0^*, \dots, \mu_{N-1}^*\}$  è ottima.

Schema di dimostrazione della proposizione precedente:

Per ogni politica ammissibile  $\pi = \{\mu_0, \dots, \mu_{N-1}\}$  sia  $\pi^k = \{\mu_k, \dots, \mu_{N-1}\}$  con  $k = 0, \dots, N - 1$ . Si indica con  $J_k^*(x_k)$  il costo ottimo per il problema a  $N - k$  passi con condizione iniziale  $x_k$  al tempo iniziale  $k$  e tempo finale  $N$ :

$$J_k^*(x_k) = \min_{\pi^k} E_{w_k, \dots, w_{N-1}} \left\{ g_N(x_N) + \sum_{i=k}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\}.$$

Per induzione vogliamo provare che i valori delle iterazioni coincidono corrispondono all'ottimo del sottoproblema relativo. In altre parole vogliamo provare che  $J_k(x_k) = J_k^*(x_k)$  per  $k = 0, \dots, N$ . Procediamo per induzione a ritroso.

Nel caso iniziale si ha per definizione  $J_N^*(x_N) = g_N(x_N)$  e  $J_N(x_N) = g_N(x_N)$  e quindi la tesi è vera per  $k = N$ . Si supponga per l'induzione che la tesi sia vera per  $k + 1$  e che quindi  $J_{k+1}(x_{k+1}) = J_{k+1}^*(x_{k+1})$  per ogni  $x_{k+1}$ .



Sia  $\pi^k = (\mu_k, \pi^{k+1})$ , si ha

$$\begin{aligned}
J_k^*(x_k) &= \min_{(\mu_k, \pi^{k+1})} E_{w_k, \dots, w_{N-1}} \left\{ g_k(x_k, \mu_k(u_k), w_k) + g_N(x_N) + \sum_{i=k+1}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\} = \\
&= \min_{\mu_k} E_{w_k} \left\{ g_k(x_k, \mu_k(u_k), w_k) + \min_{\pi^{k+1}} E_{w_{k+1}, \dots, w_{N-1}} \left\{ g_N(x_N) + \sum_{i=k+1}^{N-1} g_i(x_i, \mu_i(x_i), w_i) \right\} \right\} = \\
&= \min_{\mu_k} E_{w_k} \{ g_k(x_k, \mu_k(u_k), w_k) + J_{k+1}^*(f_k(x_k, \mu_k(u_k), w_k)) \} = \\
&= \min_{\mu_k} E_{w_k} \{ g_k(x_k, \mu_k(u_k), w_k) + J_{k+1}(f_k(x_k, \mu_k(u_k), w_k)) \} = J_k(x_k)
\end{aligned}$$

in questo modo la tesi è verificata per  $k$  e la proposizione è dimostrata.

Vediamo ora un esempio di soluzione di un problema di controllo ottimo con la programmazione dinamica.

### **Esempio:**

Si consideri del materiale che deve passare attraverso due forni in sequenza. Sia  $x_0$  la temperatura iniziale del materiale e  $x_i$   $i = 1, 2$  la temperatura del materiale in uscita dal forno  $i$ . Siano inoltre  $u_{i-1}$  le temperature che vengono erogate dal forno  $i - 1 = 0, 1$  e si supponga per semplicità che non siano limitate. La temperatura in uscita da un forno dipende dalla temperatura di ingresso e dalla temperatura erogata come segue:

$$x_{k+1} = (1 - a)x_k + au_k, \quad k = 0, 1$$

dove  $a \in (0, 1)$  è un parametro. L'obiettivo è quello di determinare le temperature dei forni  $u_i$  in modo tale da avere una temperatura finale vicino a  $T$  e minimizzando l'energia spesa. In altre parole si vuole minimizzare il costo  $r(x_2 - T)^2 + u_0^2 + u_1^2$ , con  $r > 0$ .

Si vuole quindi risolvere il problema con la programmazione dinamica con orizzonte temporale  $N = 2$ , si noti che  $g_i(x_i, u_i, w_i) = u_i^2$  con  $i = 1, 2$ . Si ha

$J_2(x_2) = r(x_2 - T)^2$  come passo iniziale dell'iterazione (a questo punto dell'iterazione non si conosce ancora  $x_2$ ).

Per  $k = 1$ :  $J_1(x_1) = \min_{u_1} [u_1^2 + J_2(x_2)] = \min_{u_1} [u_1^2 + r((1 - a)x_1 + au_1 - T)^2]$ .

Per calcolare il controllo  $u_1$  ottimo è sufficiente derivare l'argomento della minimizzazione e calcolarne una soluzione nel caso di derivata nulla:

$\frac{\partial}{\partial u_1} u_1^2 + r((1 - a)x_1 + au_1 - T)^2 = 2u_1 + ra((1 - a)x_1 + au_1 - T) = 0$  da cui segue

$$\mu_1^*(x_1) = u_1 = \frac{ra(T - (1 - a)x_1)}{1 + ra^2}.$$

In altre parole si ottiene la legge di controllo ottima per la determinazione della temperatura del secondo forno.

Utilizzando la  $u_1$  così ottenuta si ottiene  $J_1(x_1) = \frac{r((1-a)x_1 - T)^2}{1+ra^2}$ .

Per  $k = 0$ :  $J_0(x_0) = \min_{u_0} [u_0^2 + J_1(x_1)] = \min_{u_0} [u_0^2 + J_1(x_1)] = \min_{u_0} [u_0^2 + J_1((1 - a)x_0 + au_0)] = \min_{u_0} \left[ u_0^2 + \frac{r((1-a)((1-a)x_0 + au_0) - T)^2}{1+ra^2} \right]$ . Anche in questo caso, per calcolare il controllo  $u_0$  ottimo è sufficiente derivare l'argomento della minimizzazione e calcolarne una soluzione nel caso di derivata nulla. In questo caso si ottiene che la legge di controllo ottima per la

determinazione della temperatura del primo forno risulta

$$\mu_0^*(x_0) = u_0 = \frac{r(1-a)a(T - (1-a)^2x_0)}{1 + ra^2(1 + (1-a)^2)}.$$

Utilizzando la  $u_0$  così ottenuta si ottiene  $J_0(x_0) = \frac{r((1-a)^2x_0 - T)^2}{1 + ra^2(1 + (1-a)^2)}$ .

Data la temperatura iniziale del materiale si possono infine calcolare gli ingressi e gli stati del sistema.

I conti in questo caso sono semplici grazie al fatto che il sistema è lineare e la funzione costo è quadratica.

## Pianificazione discreta ottima a lunghezza fissata

Sia  $\pi_N$  una *pianificazione a  $N$  passi*, i.e. una sequenza  $(u_1, u_2, \dots, u_N)$  di  $N$  azioni di controllo. Dati  $\pi_N$  e  $x_I$  è possibile ricostruire, tramite la funzione di transizione  $f$ , la sequenza degli stati  $x_I, x_2, \dots, x_{N+1}$  dove  $x_{k+1} = f(x_k, u_k)$ .

Sia  $L$  una funzione costo a valori reali che sia additiva rispetto al passo  $N$  della pianificazione e che sia definita su una pianificazione a  $N$  passi  $\pi_N$ . Se si denota con  $F$  il *passo finale*,  $F = N + 1$  si ha che la *funzione costo* è data da

$$L(\pi_N) = \sum_{k=1}^N l(x_k, u_k) + l_F(x_F).$$

Il termine  $l_F(x_F)$  è tale che  $l_F(x_F) = 0$  se  $x_F \in \mathcal{X}_G$  e  $l_F(x_F) = \infty$  altrimenti. In questo modo  $L$  è definita su ogni traiettoria, quelli che non consentono di raggiungere stati in  $\mathcal{X}_G$  vengono penalizzati da un costo infinito.

Se si è interessati soltanto a traiettorie ammissibili di lunghezza  $N$  è sufficiente porre  $l(x, u) \equiv 0$ . Se invece si vuole minimizzare il numero di passi è sufficiente porre  $l(x, u) \equiv 1$ .

Gli algoritmi che descriveremo in seguito si basano sul principio di ottimalità secondo il quale ogni porzione di una traiettoria ottima è essa stessa ottima. Questo principio induce un algoritmo di tipo iterativo detto *value iteration* in grado di risolvere una vasta gamma di problemi e non soltanto quelli discussi in questo corso (ad esempio pianificazione con incertezze stocastiche o con misure degli stati non perfette). L'idea è quella di calcolare iterativamente le funzioni cost-to-go e cost-to-come ottime.

### Backward value iteration

Sia  $G_k^*$  per  $1 \leq k \leq F$  il costo che si accumula dal passo  $k$  al passo  $F$  lungo un cammino ottimo:

$$G_k^*(x_k) = \min_{u_k, \dots, u_N} \left\{ \sum_{i=k}^N l(x_i, u_i) + l_F(x_F) \right\}. \quad (9)$$

Per  $k = F = N + 1$  si ha  $G_F^*(x_F) = l_F(x_F)$ . Partendo da  $x_F$  si ottiene

$$G_N^*(x_N) = \min_{u_N} \{l(x_N, u_N) + l_F(x_F)\} = \min_{u_N} \{l(x_N, u_N) + G_F^*(f(x_N, u_N))\},$$

che risulta facilmente calcolabile per ogni  $x_N \in \mathcal{X}$ .

Mostriamo ora come calcolare  $G_k^*$  a partire da  $G_{k+1}^*$ . La 1 può essere riscritta nel seguente modo

$$G_k^*(x_k) = \min_{u_k} \min_{u_{k+1}, \dots, u_N} \left\{ l(x_k, u_k) + \sum_{i=k+1}^N l(x_i, u_i) + l_F(x_F) \right\} \quad (10)$$

$$= \min_{u_k} \left[ l(x_k, u_k) + \min_{u_{k+1}, \dots, u_N} \left\{ \sum_{i=k+1}^N l(x_i, u_i) + l_F(x_F) \right\} \right] \quad (11)$$

$$= \min_{u_k} \{ l(x_k, u_k) + G_{k+1}^*(x_{k+1}) \}. \quad (12)$$

L'ultimo membro della catena di uguaglianze dipende soltanto da  $x_k$ ,  $u_k$  e  $G_{k+1}^*$ . Si prosegue sino ad ottenere  $G_1^*$  da cui poi si ottiene  $G^*(x_I)$  che rappresenta il costo ottimo per andare da  $x_G$  a  $x_I$ .

Per ottenere anche la sequenza di controlli ottima è necessario mantenere in memoria il controllo che minimizza la 2.



**Esempio 1:** Dato il grafo in figura 8, si consideri  $x_I = a$ ,  $\mathcal{X}_G = \{d\}$  e  $N = 4$ .

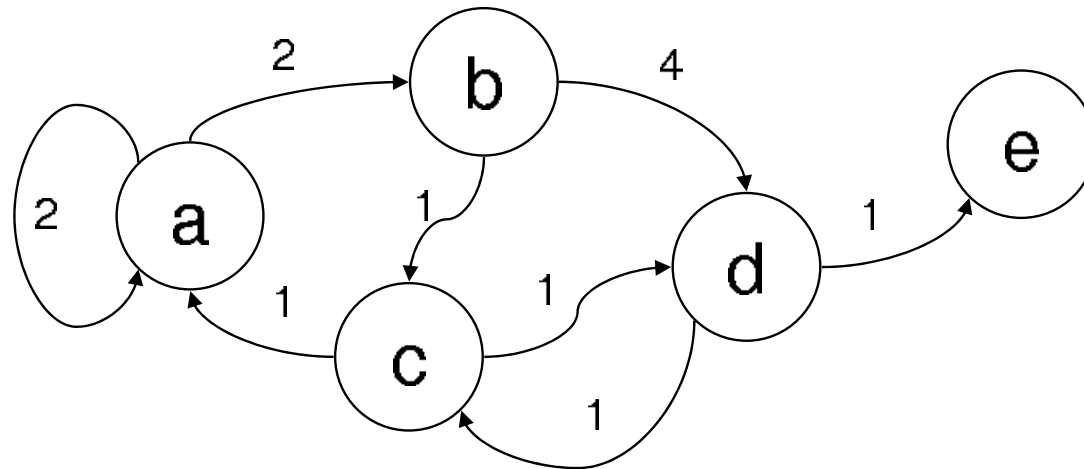


Figura 13: Determinare il cammino a costo minimo di 4 passi.

Calcolando all'indietro il cost-to-go ottimo si ottiene la seguente tabella.

	a	b	c	d	e
$G_5^*$	$\infty$	$\infty$	$\infty$	0	$\infty$
$G_4^*$	$\infty$	4	1	$\infty$	$\infty$
$G_3^*$	6	2	$\infty$	2	$\infty$
$G_2^*$	4	6	3	$\infty$	$\infty$
$G_1^*$	6	4	5	4	$\infty$

Infatti inizialmente  $N + 1 = 5$  e  $x_F = d$  per cui  $G_5^*(d) = 0$  e tutti gli altri valgono  $\infty$ . Il nodo  $d$  in un passo si può raggiungere soltanto dai nodi  $b$  e  $c$  pertanto si aggiornano i rispettivi valori di  $G_4^*$  a 4 e 1 rispettivamente. Questi nodi sono raggiungibili in un passo (e quindi  $d$  lo è in due passi) dai nodi  $a$ ,  $b$  e  $d$ . Si procede in questo modo sino ad ottenere  $G_1^*$ . In figura 9 sono riportati i vari passi con i costi dei cammini. I costi ottimi per ogni passo sono riportati in grassetto. Il cost-to-go del nodo  $e$  non viene mai aggiornato e pertanto rimane infinito. Infatti dal nodo  $e$  non è possibile raggiungere mai il nodo  $d$  e tanto meno in 4 passi.

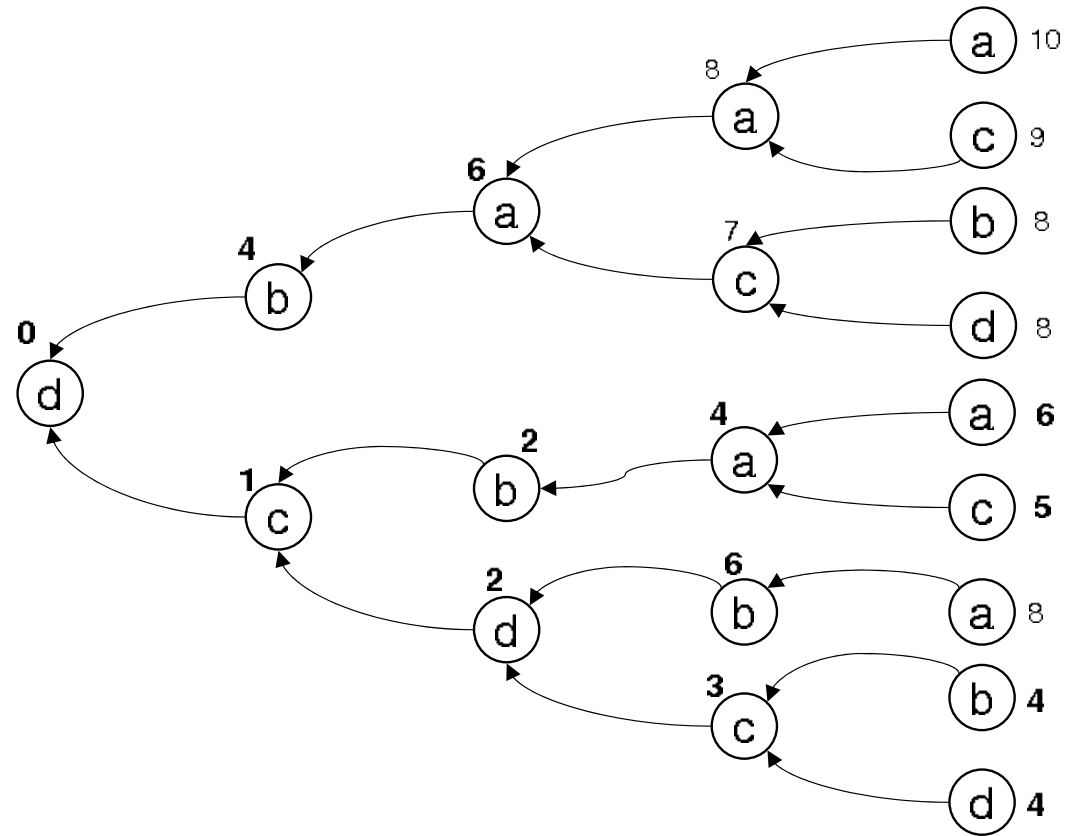


Figura 14: Procedimento per il calcolo del cammino a costo minimo di 4 passi all'indietro.

### Forward value iteration

L'iterazione in avanti può essere usata per calcolare tutti i cammini ottimi per raggiungere ogni stato di  $\mathcal{X}$  a partire da un  $x_I$  fissato. Sia  $C_k^*$  il *cost-to-come* ottimo dal passo 1 al passo  $k$  ottimizzato su tutti i cammini di  $k - 1$  passi. Per escludere cammini che non partono da  $x_1 = x_I$  si pone  $C_1^*(x_I) = l_I(x_I)$  dove  $l_I(x_I) = 0$  mentre  $l_I(x) = \infty$  per  $x \neq x_I$ . In generale il cost-to-come ottimo è dato da

$$C_k^*(x_k) = \min_{u_1, \dots, u_{k-1}} \left\{ l_I(x_1) + \sum_{i=1}^{k-1} l(x_i, u_i) \right\} \quad (13)$$

e al passo finale  $F$  si ha  $C_F^*(x_F) = \min_{u_1, \dots, u_N} \left\{ l_I(x_1) + \sum_{i=1}^N l(x_i, u_i) \right\}$ .

Per ottenere una formula ricorsiva supponiamo di conoscere  $C_{k-1}^*$ :

$$C_k^*(x_k) = \min_{u^{-1} \in \mathcal{U}^{-1}(x_k)} \left\{ C_{k-1}^*(x_{k-1}) + l(x_{k-1}, u_{k-1}) \right\}, \quad (14)$$

dove  $x_{k-1} = f^{-1}(x_k, u_k^{-1})$  e  $u_{k-1} \in \mathcal{U}(x_{k-1})$  è l'ingresso a cui corrisponde  $u_k^{-1} \in \mathcal{U}^{-1}(x_k)$ .

**Esempio 2:** Si consideri nuovamente il grafo in figura 8 ma si ponga  $x_I = a$  e  $N = 4$ . Le funzioni cost-to-come ottime calcolate con l'iterazione in avanti sono riportate nella seguente tabella.

	a	b	c	d	e
$C_1^*$	0	$\infty$	$\infty$	$\infty$	$\infty$
$C_2^*$	2	2	$\infty$	$\infty$	$\infty$
$C_3^*$	4	4	3	6	$\infty$
$C_4^*$	4	6	5	4	7
$C_5^*$	6	6	5	6	5

In figura 10 sono riportati i vari passi con i costi dei cammini. I costi ottimi per ogni passo sono riportati in grassetto.

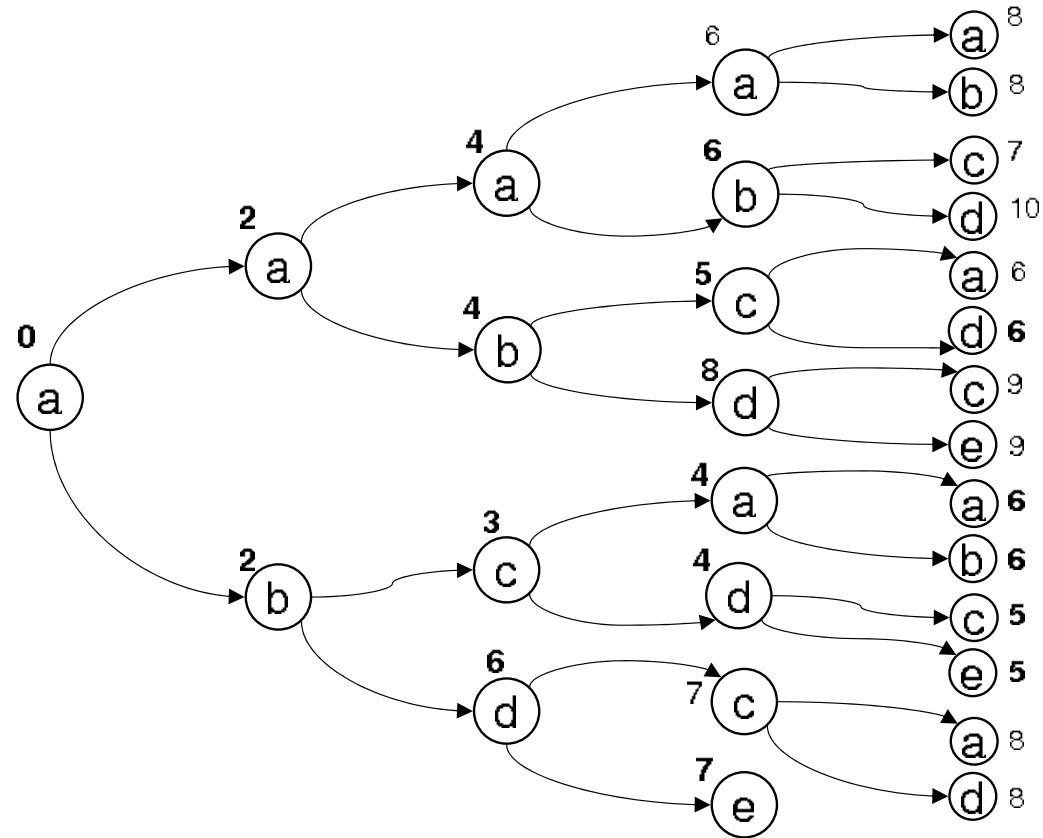


Figura 15: Procedimento per il calcolo del cammino a costo minimo di 4 passi in avanti.

## Pianificazione discreta ottima a lunghezza variabile

I metodi visti nel paragrafo precedente si possono facilmente estendere al calcolo di traiettorie ottime di lunghezza non specificata. È necessario introdurre una *azione terminale*  $u_T$  tale che quando applicata ad uno stato  $x_k$  lo stato non cambierà per i passi successivi. In altre parole per ogni  $i \geq k$ ,  $u_i = u_T$ ,  $x_i = x_k$  e  $l(x_i, u_T) = 0$ . Con questa azione terminale è possibile utilizzare i metodi iterativi introdotti precedentemente con passo  $N$  e ottenere cammini ottimi di lunghezza minore o uguale a  $N$ . Infatti se si ha un cammino ottimo in 2 passi si può applicare per tre passi successivi l'azione terminale e si ottiene un cammino ottimo in 5 passi per lo stesso stato.

Il passo successivo da compiere, nella estensione degli algoritmi, è quello di eliminare la dipendenza da  $N$ . Se si considera l'iterazione all'indietro, una volta calcolato  $G_1^*$  è possibile continuare il procedimento e calcolare  $G_0^*$ ,  $G_{-1}^*$  eccetera.

Il procedimento si interrompe nel momento in cui la situazione diventa *stazionaria* e cioè quando per ogni  $i \leq k$   $G_{i-1}^*(x) = G_i^*(x)$  per tutti gli stati  $x \in \mathcal{X}$ . Nel momento in cui ci si trova in una situazione stazionaria il cost-to-go non dipende più dal particolare passo  $k$ . È importante osservare che se la funzione  $l(x, u)$  non è mai negativa allora prima o poi ci si trova in condizione di stazionarietà.

Lo stesso risultato si ottiene nel caso di iterazione in avanti. Inoltre con entrambi gli approcci è possibile ottenere tramite l'algoritmo anche la sequenza dei controlli che determina la traiettoria ottima.



**Esempio 3:** Si consideri nuovamente il grafo in figura 8, le funzioni cost-to-come ottime calcolate con l'iterazione all'indietro sono riportate nella seguente tabella.

	a	b	c	d	e
$G_0^*$	$\infty$	$\infty$	$\infty$	0	$\infty$
$G_{-1}^*$	$\infty$	4	1	0	$\infty$
$G_{-2}^*$	6	2	1	0	$\infty$
$G_{-3}^*$	4	2	1	0	$\infty$
$G_{-4}^*$	4	2	1	0	$\infty$

Si noti che dopo 4 passi si è in condizione di stazionarietà.

Nel caso dell'iterazione in avanti si consideri  $x_I = b$ :

	a	b	c	d	e
$C_1^*$	$\infty$	0	$\infty$	$\infty$	$\infty$
$C_2^*$	$\infty$	0	1	4	$\infty$
$C_3^*$	2	0	1	2	5
$C_4^*$	2	0	1	2	3
$C^*$	2	0	1	2	3

Anche in questo caso dopo 4 passi si è in condizione di stazionarietà.

## **Algoritmo di Dijkstra e programmazione dinamica**

Il metodo iterativo in avanti e l'algoritmo di Dijkstra sono molto simili. In particolare però l'algoritmo di Dijkstra etichetta come morti i nodi per i quali il costo non verrà modificato e tali nodi non verranno più rivisitati dall'algoritmo. In generale però l'algoritmo di Dijkstra può risultare computazionalmente costoso nella gestione delle code.