Ricerca intelligente

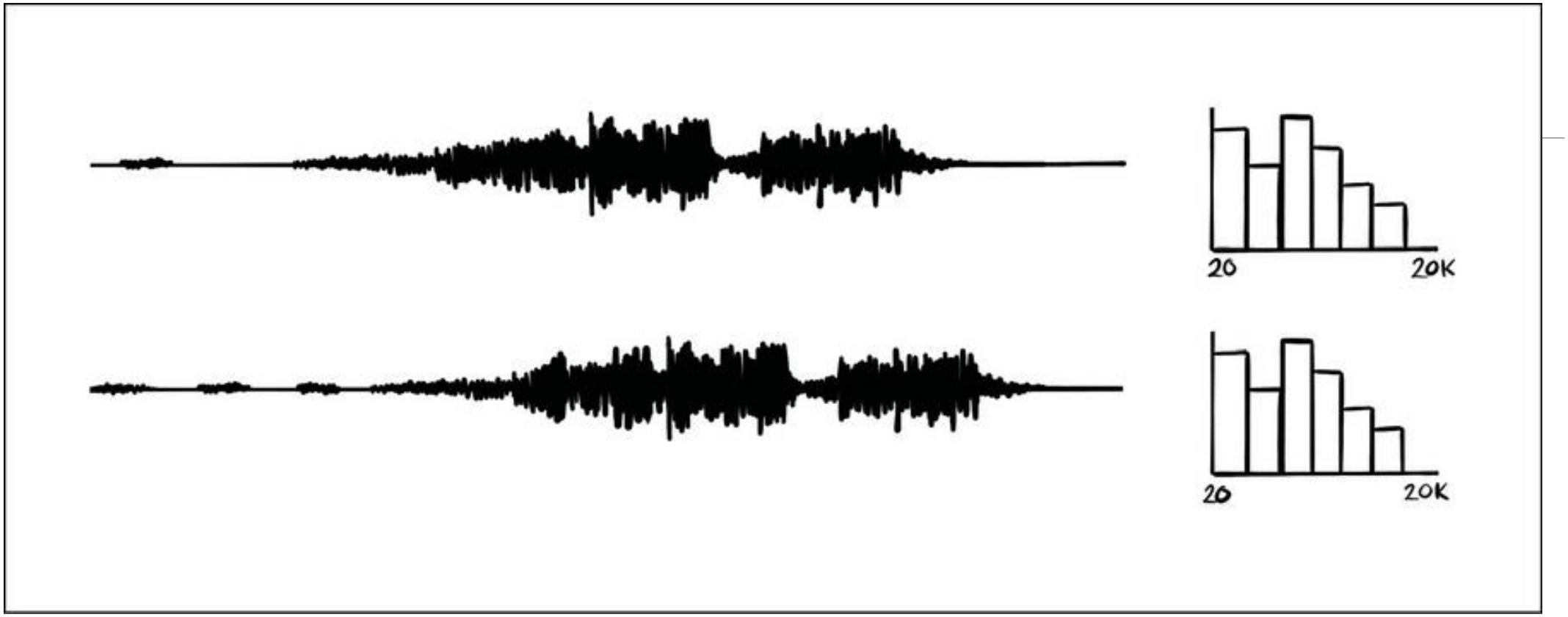
A CURA DI LINO POLO

Definizione di euristica: applicazione di scelte informate

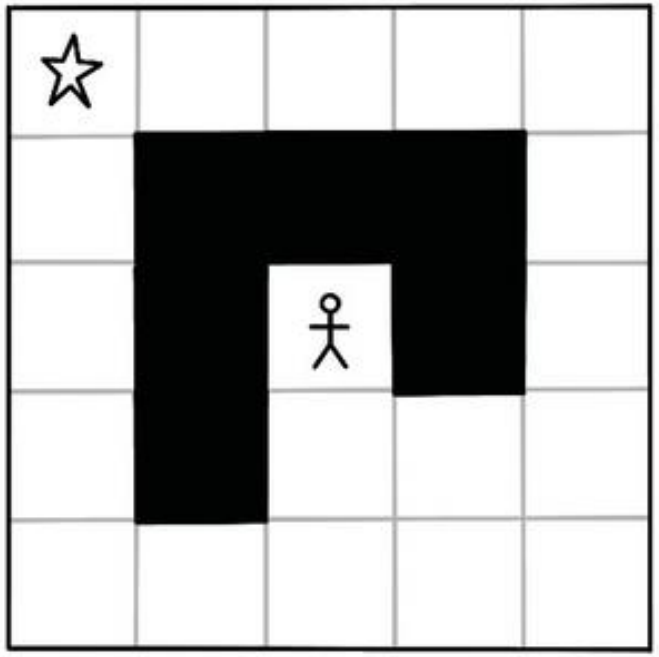
Ora che abbiamo un'idea di come funzionano gli algoritmi di ricerca non informati, possiamo esplorare come questi possono essere migliorati potendo contare su maggiori informazioni sul problema. A tale scopo, utilizziamo la ricerca informata. Si ha una ricerca informata quando l'algoritmo può contare su un contesto relativo al problema da risolvere. Le euristiche sono un modo per rappresentare questo contesto. Spesso chiamata regola empirica, un'euristica è una regola (o un insieme di regole) utilizzata per valutare uno stato. Può essere utilizzata per definire i criteri che uno stato deve soddisfare o per misurare le prestazioni di un determinato stato. Un'euristica viene utilizzata quando non esiste un metodo ben definito per trovare una soluzione ottimale. Un'euristica è un po' come un'ipotesi, e dovrebbe essere considerata più come una linea guida che come una verità scientifica relativa al problema che si deve risolvere.

Un altro esempio è la scrittura di algoritmi per risolvere un problema di navigazione GPS. L'euristica potrebbe essere "I buoni percorsi riducono al minimo il tempo nel traffico e minimizzano la distanza percorsa" oppure "I buoni percorsi riducono al minimo i pedaggi e massimizzano le buone condizioni stradali". Una scarsa euristica per un programma di navigazione GPS potrebbe cercare di ridurre al minimo la distanza in linea retta fra due punti. Questa euristica potrebbe funzionare per gli uccelli o gli aerei, ma non per chi deve camminare o guidare: siamo legati a strade, vie e sentieri, che aggirano edifici e ostacoli. L'euristica deve quindi essere sensata per il contesto in cui viene impiegata.

Un esempio: occorre controllare se una clip audio caricata è tratta da una libreria di contenuti protetti da copyright. Poiché le clip audio sono costituite da frequenze sonore, un modo per raggiungere questo obiettivo consiste nel ricercare ogni frammento del clip caricato in ogni clip contenuto nella libreria. Questo compito richiederà un'enorme quantità di calcolo. Un approccio primitivo per realizzare una ricerca migliore potrebbe consistere nel definire un'euristica che minimizzi la differenza di distribuzione delle frequenze fra le due clip, come mostrato nella Figura 3.1. Notate che le frequenze sono identiche, a parte la differenza temporale; non vi sono differenze nelle distribuzioni di frequenze. Questa soluzione potrebbe non essere perfetta, ma è un buon inizio verso la realizzazione di un algoritmo meno costoso dal punto di vista computazionale.


20
20K
20
20K

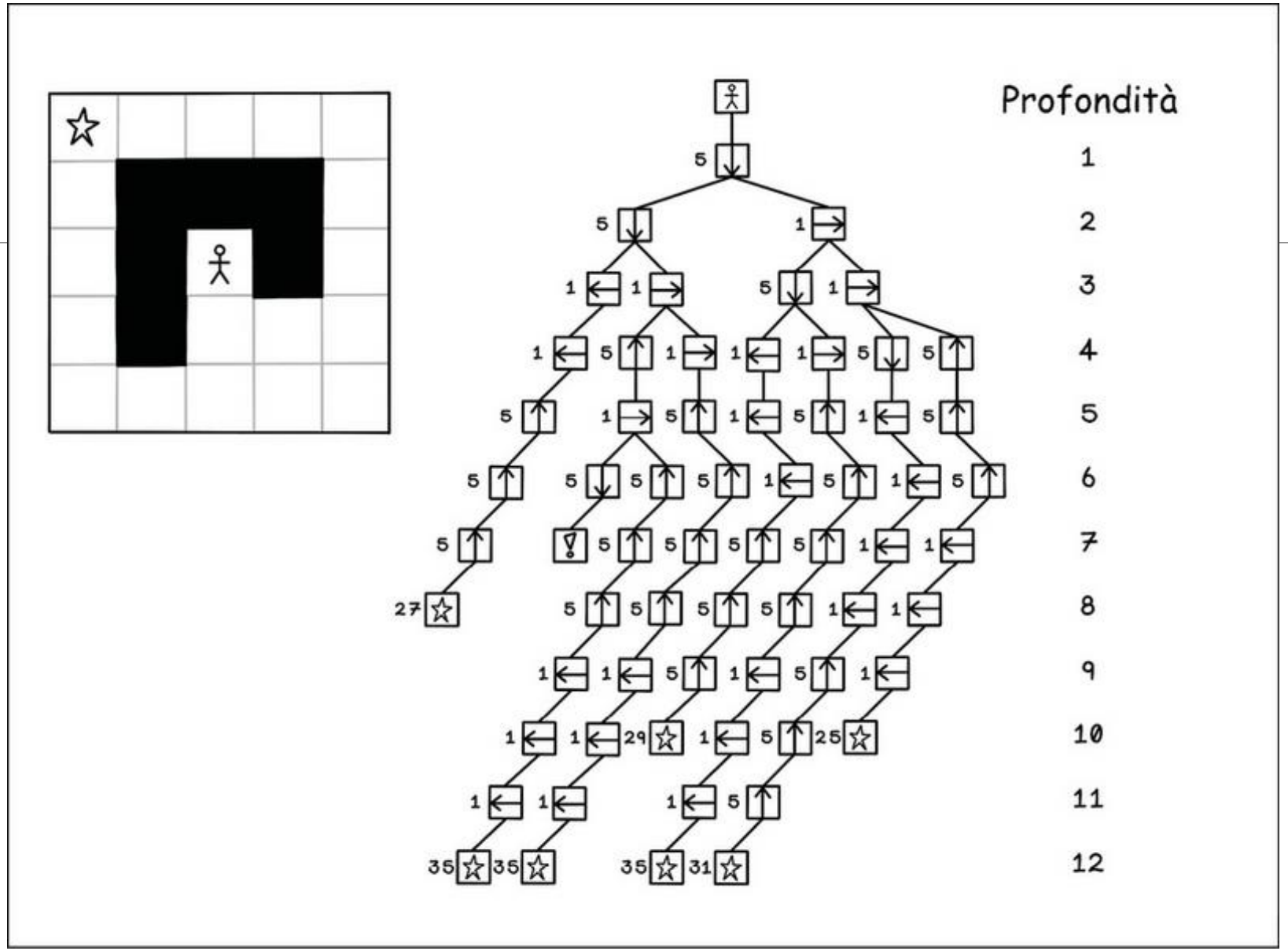
Le euristiche sono specifiche del contesto, e una buona euristica può aiutare a ottimizzare sensibilmente le soluzioni. Lo scenario del labirinto del Capitolo 2 verrà adattato per dimostrare il concetto di creazione di euristiche, introducendo una dinamica interessante. Invece di trattare tutti i movimenti allo stesso modo e valutare le migliori soluzioni esclusivamente in base ai percorsi con meno azioni (quindi con una profondità ridotta nell'albero), ai movimenti in direzioni differenti ora possiamo associare costi differenti. C'è stata come una strana alterazione della forza di gravità nel nostro labirinto, e spostarsi a nord o a sud ora costa cinque volte di più che spostarsi a est o a ovest.



	COSTO
↑	5
↓	5
←	1
→	1

Nel nuovo scenario del labirinto, i fattori che influenzano il miglior percorso possibile verso l'obiettivo sono il numero di azioni intraprese e la somma dei costi di ciascuna azione di tale percorso.

Nella Figura tutti i possibili percorsi sono rappresentati in un albero, per evidenziare le opzioni disponibili, indicando i costi delle rispettive azioni. Ancora una volta, questo esempio riguarda solo lo spazio di ricerca di questo specifico labirinto e non è applicabile a scenari della vita reale. L'algoritmo genererà l'albero come parte


Profondità
1
2
3
4
5
6
7
8
9
10
11
12
27
29
25
35
35
35
31

Un’euristica per il problema del labirinto può essere definita come segue: “I buoni percorsi minimizzano il costo del movimento e minimizzano le mosse totali per raggiungere l’obiettivo”. Questa semplice euristica aiuta a determinare quali nodi vengono visitati, perché stiamo applicando alcune conoscenze del dominio per risolvere il problema.

Esperimento mentale: dato il seguente scenario, quale euristica potete immaginare?

Minatori differenti sono specializzati in tipi di estrazione differenti: diamanti, oro, platino… Tutti i minatori sono produttivi in qualsiasi miniera, ma sono più efficaci nelle miniere in cui sono specializzati. In un'area vi sono varie miniere di diamanti, oro e platino, e i magazzini si trovano a distanze differenti fra le miniere. Se il problema è quello di distribuire i minatori per massimizzare la loro efficienza e ridurre i tempi di spostamento, quale potrebbe essere un'euristica?

Esperimento mentale: una possibile soluzione

Un’euristica sensata includerebbe l’assegnazione di ogni minatore a una miniera in cui è specializzato e l’incarico di recarsi al magazzino più vicino a quella miniera. Ciò può anche essere interpretato come minimizzare le assegnazioni di minatori a miniere in cui non sono specializzati e minimizzare anche la distanza da percorrere per arrivare ai magazzini.

Ricerca informata: ricerca di soluzioni con una guida

La ricerca informata, o ricerca euristica, è un algoritmo che utilizza entrambi gli approcci di ricerca (in ampiezza e in profondità) combinati con una certa intelligenza. La ricerca è guidata dall'euristica, data una certa conoscenza del problema in questione.

Possiamo impiegare diversi algoritmi di ricerca informata, a seconda della natura del problema, fra i quali la ricerca Greedy (nota anche come Best-first). L'algoritmo di ricerca informata più utilizzato e utile, tuttavia, è A*.

La ricerca A*

L'algoritmo di ricerca A* (pronunciato "A star") di solito migliora le prestazioni stimando l'euristica per minimizzare il costo del prossimo nodo visitato.

Il costo totale viene calcolato con due metriche: la distanza totale dal nodo iniziale al nodo corrente e il costo stimato del passaggio a un determinato nodo utilizzando un'euristica. Quando l'obiettivo è quello di ridurre al minimo i costi, un valore inferiore indica una soluzione che offre prestazioni migliori (Figura 3.4).

$$\mathbf{f}(n) = \mathbf{g}(n) + \mathbf{h}(n)$$

g(n): costo del percorso dal nodo iniziale al nodo n.

h(n): costo della funzione euristica per il nodo n.

f(n): costo del percorso dal nodo iniziale al nodo n, più costo della funzione euristica per il nodo n.

Il seguente schema di elaborazione è un esempio astratto di come un albero venga visitato utilizzando l'euristica per guidare la nostra ricerca. L'attenzione si concentra sui calcoli euristici per i diversi nodi dell'albero.

La ricerca in ampiezza visita tutti i nodi a un certo livello di profondità, prima di passare alla profondità successiva. La ricerca in profondità visita tutti i nodi fino alla profondità massima, prima di tornare alla radice e visitare il percorso successivo. Una ricerca A* è differente, in quanto non ha uno schema predefinito da seguire; i nodi vengono visitati in un ordine basato sui loro costi euristici. Notate che l'algoritmo non conosce i costi di tutti i nodi. I costi vengono calcolati a mano a mano che l'albero viene esplorato o generato, e ogni nodo visitato viene aggiunto a uno stack. Questo ci permette di ignorare i nodi che costano più dei nodi già visitati, risparmiando tempo di calcolo

Esaminiamo il flusso dell'algoritmo di ricerca A*.

• Aggiungi il nodo radice allo stack. L'algoritmo di ricerca A* può essere implementato con uno stack in cui per primo viene elaborato l'ultimo oggetto aggiunto (Last-In, First-Out o LIFO). Il primo passaggio consiste nell'aggiungere allo stack il nodo radice.

• Lo stack è vuoto? Se lo stack è vuoto e non è stato restituito alcun percorso nel Passaggio 8 dell'algoritmo, non esiste alcun percorso verso l'obiettivo. Se ci sono ancora nodi nello stack, l'algoritmo può continuare la sua ricerca.

• Restituisci "Nessun percorso verso l'obiettivo". Questo passaggio è l'unica possibile uscita dall'algoritmo se non esiste alcun percorso verso l'obiettivo.

• Estrai il nodo dallo stack come nodo corrente. Estraendo l'oggetto successivo dallo stack e impostandolo come nodo corrente, possiamo esplorarne le possibilità.

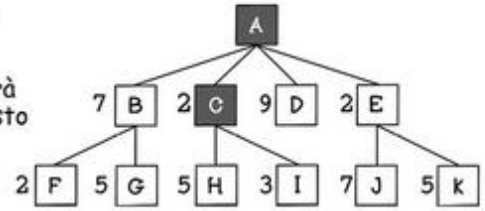
• Il nodo corrente è già stato visitato? Se il nodo corrente non è stato visitato, non è stato ancora esplorato e può essere elaborato ora.

• Contrassegna il nodo corrente come visitato. Questo passaggio indica che questo nodo è stato visitato, per impedire un'elaborazione ripetuta non necessaria.

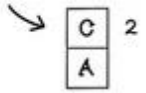
• Obiettivo raggiunto? Questo passaggio determina se il fratello corrente contiene l'obiettivo che l'algoritmo sta cercando.

• Restituisci il percorso utilizzando il nodo corrente. Facendo riferimento al genitore del nodo corrente, quindi al genitore di quel nodo e così via, viene descritto il percorso dall'obiettivo alla radice. Il nodo radice sarà un nodo senza genitore.

Dato un albero che rappresenta i nodi e i loro punteggi euristici, A* visiterà il primo figlio al costo minore. In questo caso è il nodo C, che ha costo 2.

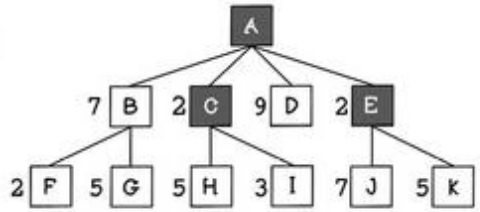


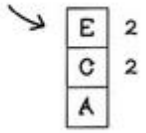
Sequenza di elaborazione dello stack.



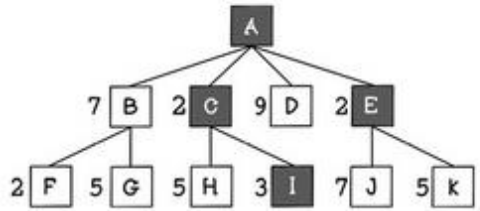
Quando due nodi hanno lo stesso costo, viene selezionato il nodo il cui costo è stato calcolato per primo.

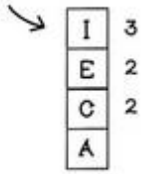
Poiché anche E ha un costo pari a 2 ed è figlio di A, quello sarà il successivo nodo visitato.





Poi A* visiterà il nodo al costo più basso fra i figli di A o i figli dei nodi che ha già visitato.

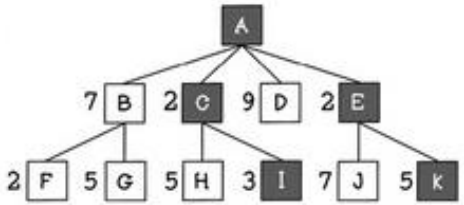




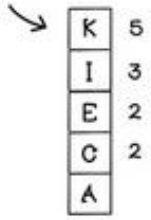
• Il nodo corrente ha un altro fratello? Se il nodo corrente ha più mosse possibili da fare nell'esempio del labirinto, quella mossa può essere aggiunta per essere elaborata. In caso contrario, l'algoritmo può tornare al Passaggio 2, in cui può essere elaborato l'oggetto successivo nello stack, se non è vuoto. La natura dello stack LIFO consente all'algoritmo di elaborare tutti i nodi fino alla profondità di un nodo foglia, prima di tornare indietro per visitare altri figli del nodo radice.

• Ordina lo stack per costo crescente. Quando lo stack viene ordinato in base al costo di ciascun nodo nello stack ascendente, viene elaborato il nodo con il costo più basso, cosa che consente di visitare sempre il nodo più economico.

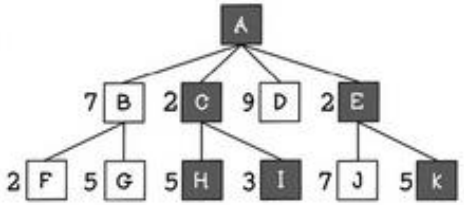
Il successivo nodo col costo più basso è K, un figlio di E.



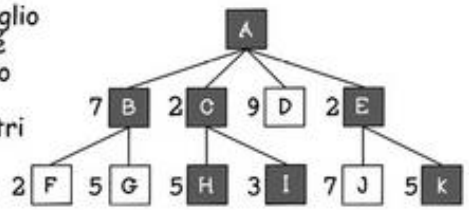
Sequenza di elaborazione dello stack.

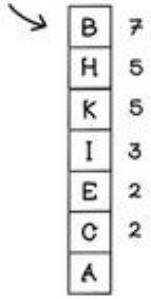


Il successivo nodo col costo più basso è H, un figlio di C.



Viene visitato un figlio diretto di A, poiché ha il costo più basso fra i figli di A e i figli di tutti gli altri nodi visitati.

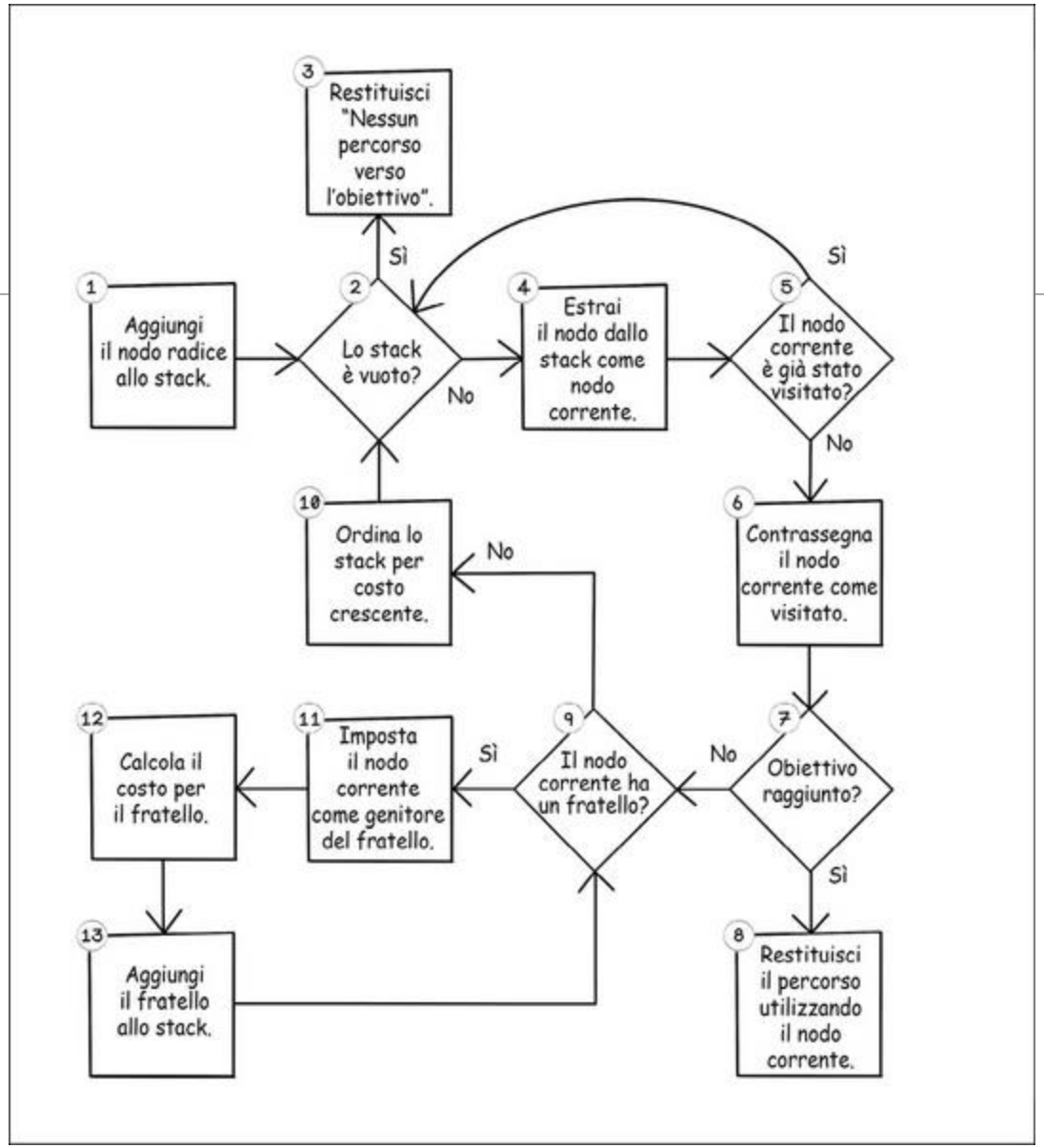




I nodi che costano più dell'attuale percorso a costo più basso verso la soluzione possono essere ignorati, poiché i percorsi verso la soluzione che attraversano questi nodi saranno più costosi.

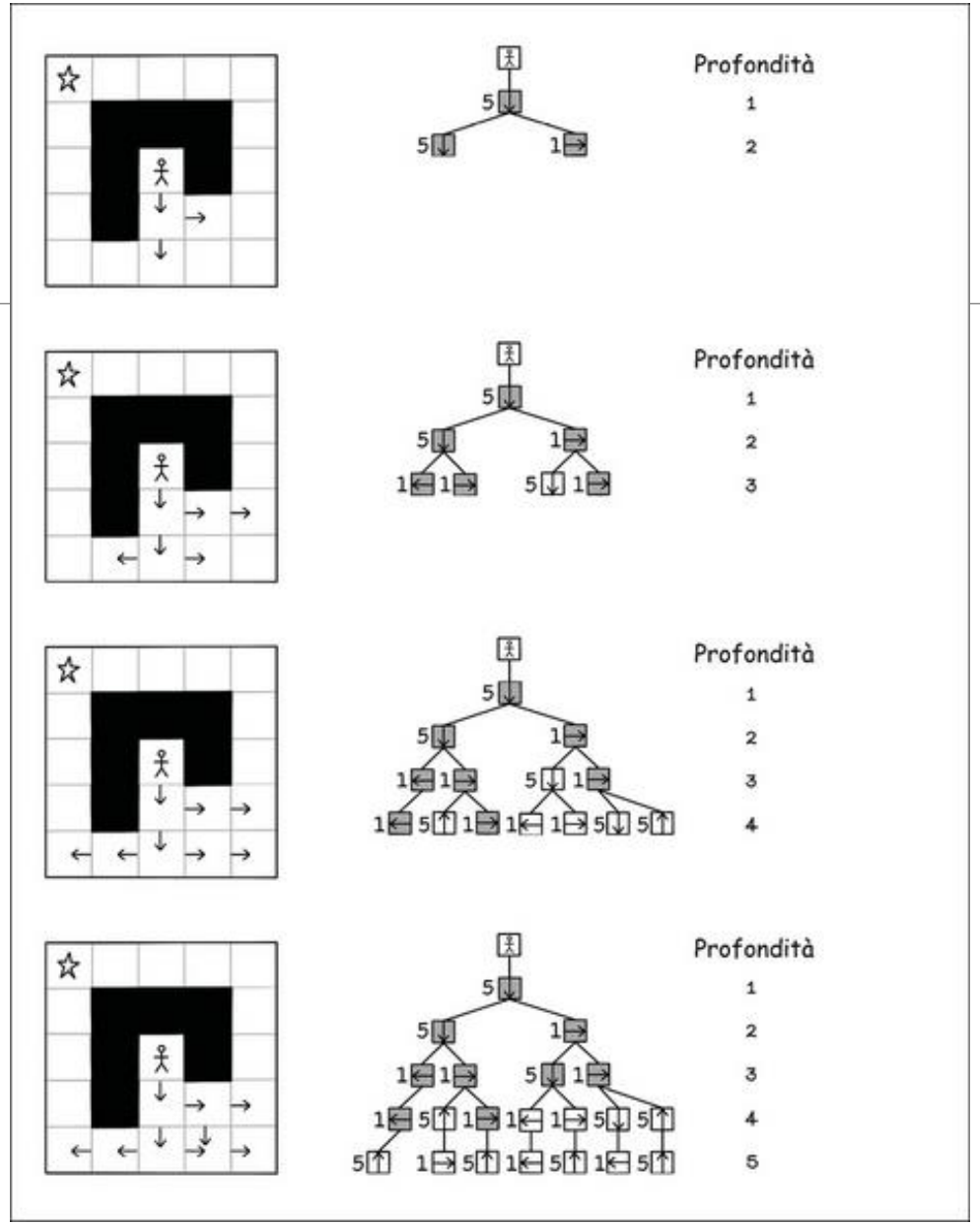
• Imposta il nodo corrente come genitore del fratello. Imposta il nodo di origine come genitore del fratello corrente. Questo passaggio è importante per tracciare il percorso dal fratello corrente al nodo radice. Dal punto di vista della mappa, l'origine è la posizione da cui si è spostato il giocatore e il fratello corrente è la posizione in cui si è spostato il giocatore.

• Calcola il costo per il fratello. La funzione di costo è fondamentale per guidare l'algoritmo A*. Il costo viene calcolato sommando alla distanza dal nodo radice il punteggio euristico della mossa successiva. Un'euristica più intelligente influenzerà direttamente l'algoritmo A*, garantendogli prestazioni migliori.

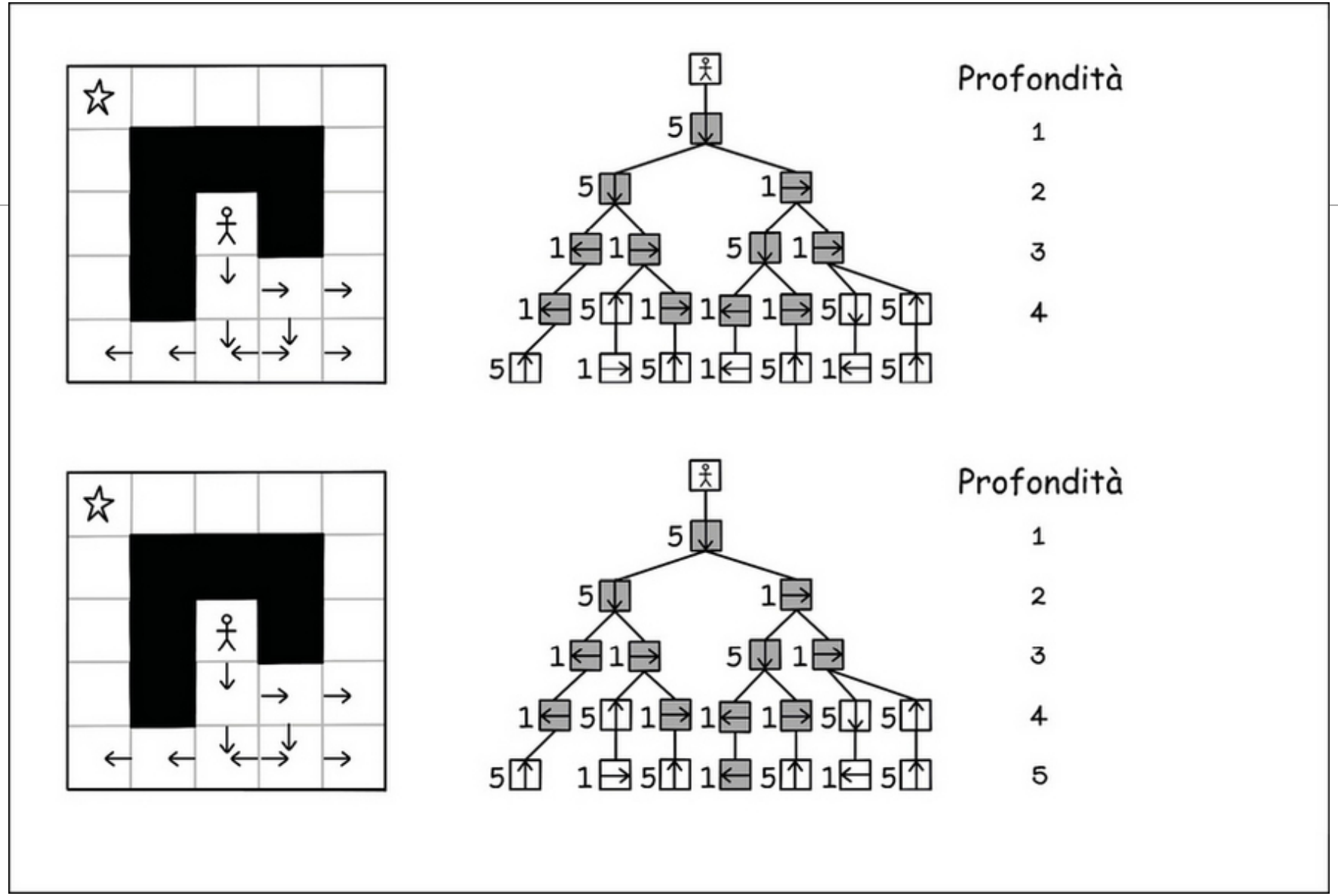

1
Aggiungi il nodo radice allo stack.
2
Lo stack è vuoto?
Sì
3
Restituisci "Nessun percorso verso l'obiettivo".
No
4
Estrai il nodo dallo stack come nodo corrente.
5
Il nodo corrente è già stato visitato?
Sì
No
6
Contrassegna il nodo corrente come visitato.
7
Obiettivo raggiunto?
No
Sì
8
Restituisci il percorso utilizzando il nodo corrente.
9
Il nodo corrente ha un fratello?
Sì
No
10
Ordina lo stack per costo crescente.
No
11
Imposta il nodo corrente come genitore del fratello.
12
Calcola il costo per il fratello.
13
Aggiungi il fratello allo stack.

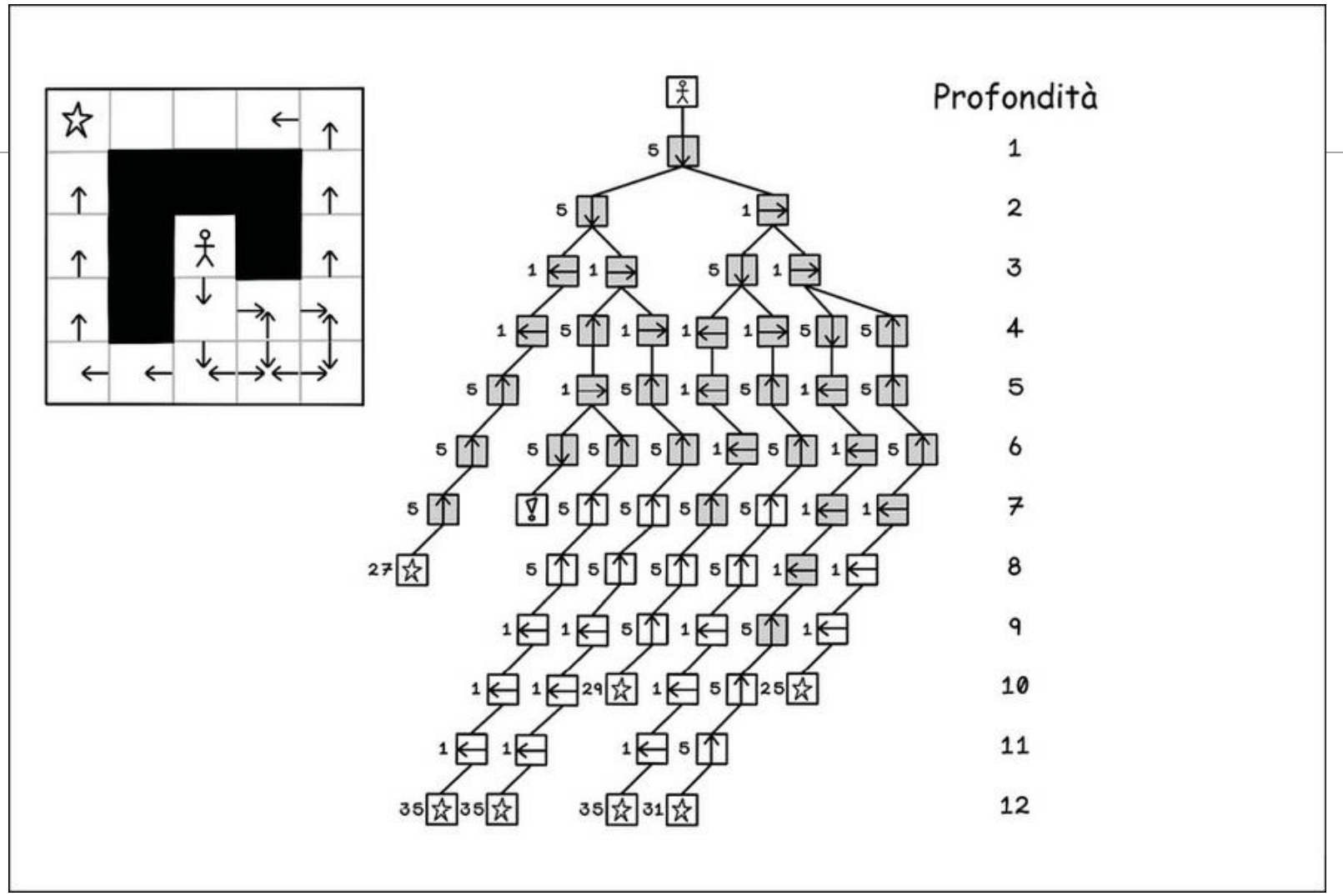
• Aggiungi il fratello allo stack. Il nodo adiacente viene aggiunto allo stack affinché i suoi figli possano essere esplorati. Ancora una volta, questo meccanismo di funzionamento dello stack consente di elaborare i nodi alla massima profondità prima di elaborare i vicini a profondità minori.

Simile alla ricerca in profondità, l'ordine dei nodi figli influenza il percorso selezionato, ma in modo meno marcato. Se due nodi hanno lo stesso costo, viene visitato il primo nodo calcolato


Profondità
1
2
Profondità
1
2
3
Profondità
1
2
3
4
Profondità
1
2
3
4
5

Notate che ci sono diversi percorsi verso l'obiettivo, ma l'algoritmo A* trova un percorso riducendo al minimo il costo per raggiungerlo, con meno mosse e costi di spostamento più economici, dato il fatto che le mosse a nord e a sud sono più costose.


Profondità
1
2
3
4
Profondità
1
2
3
4
5


Profondità
1
2
3
4
5
6
7
8
9
10
11
12

Pseudocodice

L’algoritmo A* utilizza un approccio simile all’algoritmo di ricerca in profondità, ma privilegia i nodi più economici da visitare. Per elaborare i nodi viene utilizzato uno stack, che tuttavia viene ordinato per costo crescente ogni volta che viene eseguito un nuovo calcolo. Questo ordine garantisce che l’oggetto estratto dallo stack sia sempre il più economico, in quanto dopo l’ordinamento è il primo nello stack:

```
run_astar(maze, root_point, visited_points):
  let s equal a new stack
  add root_point to s
  while s is not empty
      pop s and let current_point equal the returned point
      if current_point is not visited:
          mark current_point as visited
```

```
    if value at current_node is the goal:
        return path using current_point
    else:
        add available cells north, east, south, and west to a list neighbors
        for each neighbor in neighbors:
            set neighbor parent as current_point
            set neighbor cost as calculate_cost(current_point, neighbor)
            push neighbor to s
        sort s by cost ascending
return "No path to goal"
```

Le funzioni per il calcolo del costo sono fondamentali per il funzionamento della ricerca A*. La funzione di costo fornisce all'algoritmo le informazioni necessarie per cercare il percorso più economico. Nel nostro esempio di labirinto, un costo più elevato è associato a uno spostamento verso l'alto o verso il basso. Se la funzione di costo presenta un problema, l'algoritmo potrebbe non funzionare.

Le due funzioni seguenti descrivono come viene calcolato il costo. La distanza dal nodo radice viene aggiunta al costo del movimento successivo. Sulla base del nostro esempio ipotetico, il costo dello spostamento verso nord o verso sud influenza il costo totale della visita di quel nodo:

```
calculate_cost(origin, target):
  let distance_to_root equal length of path from origin to target
  let cost_to:move equal get_ move _cost (origin, target)
  return distance_to_root + cost_to_move
move_cost(origin,target):
  if target is north or south of origin:
      return 5
  else:
      return 1
```

Gli algoritmi di ricerca non informati, come la ricerca in ampiezza e in profondità, esplorano ogni possibilità in modo esaustivo e portano alla soluzione ottimale. La ricerca A* è un buon approccio quando è possibile creare un'euristica ragionevole per guidare la ricerca. Opera in modo più efficiente rispetto agli algoritmi di ricerca non informati, perché ignora i nodi che costano più dei nodi già visitati. Se però l'euristica è errata o non ha senso per il problema e il contesto, A* troverà soluzioni scadenti invece di quelle ottimali.

Casi d’uso per gli algoritmi di ricerca informata

Gli algoritmi di ricerca informata sono versatili e utili per diversi casi d’uso nei quali sia possibile definire un’euristica, come i seguenti.

• Ricerca di percorsi per personaggi di gioco autonomi nei videogiochi: gli sviluppatori di giochi utilizzano spesso questo algoritmo per controllare il movimento delle unità nemiche in quei giochi nei quali l’obiettivo è quello di trovare il giocatore umano all’interno di un ambiente.

• Analisi dei paragrafi nell’elaborazione del linguaggio naturale: il significato di un paragrafo può essere suddiviso in una serie di frasi, che possono essere suddivise in una serie di parole di diverso tipo (nomi e verbi, per esempio), creando una struttura ad albero che può essere valutata. La ricerca informata può essere utile per estrarne il significato.

• Routing in una rete di telecomunicazioni: è possibile utilizzare algoritmi di ricerca informata per trovare i percorsi più brevi per il traffico di rete nelle reti di telecomunicazioni, in modo da migliorarne le prestazioni. Server/nodi di rete e connessioni possono essere rappresentati come grafi ricercabili di nodi e archi.

• Giochi e puzzle a giocatore singolo: gli algoritmi di ricerca informata possono essere utilizzati per risolvere giochi e puzzle a giocatore singolo, come il cubo di Rubik, perché ogni mossa è una decisione in un albero di possibilità, fino a quando non viene trovato lo stato che rappresenta l’obiettivo.

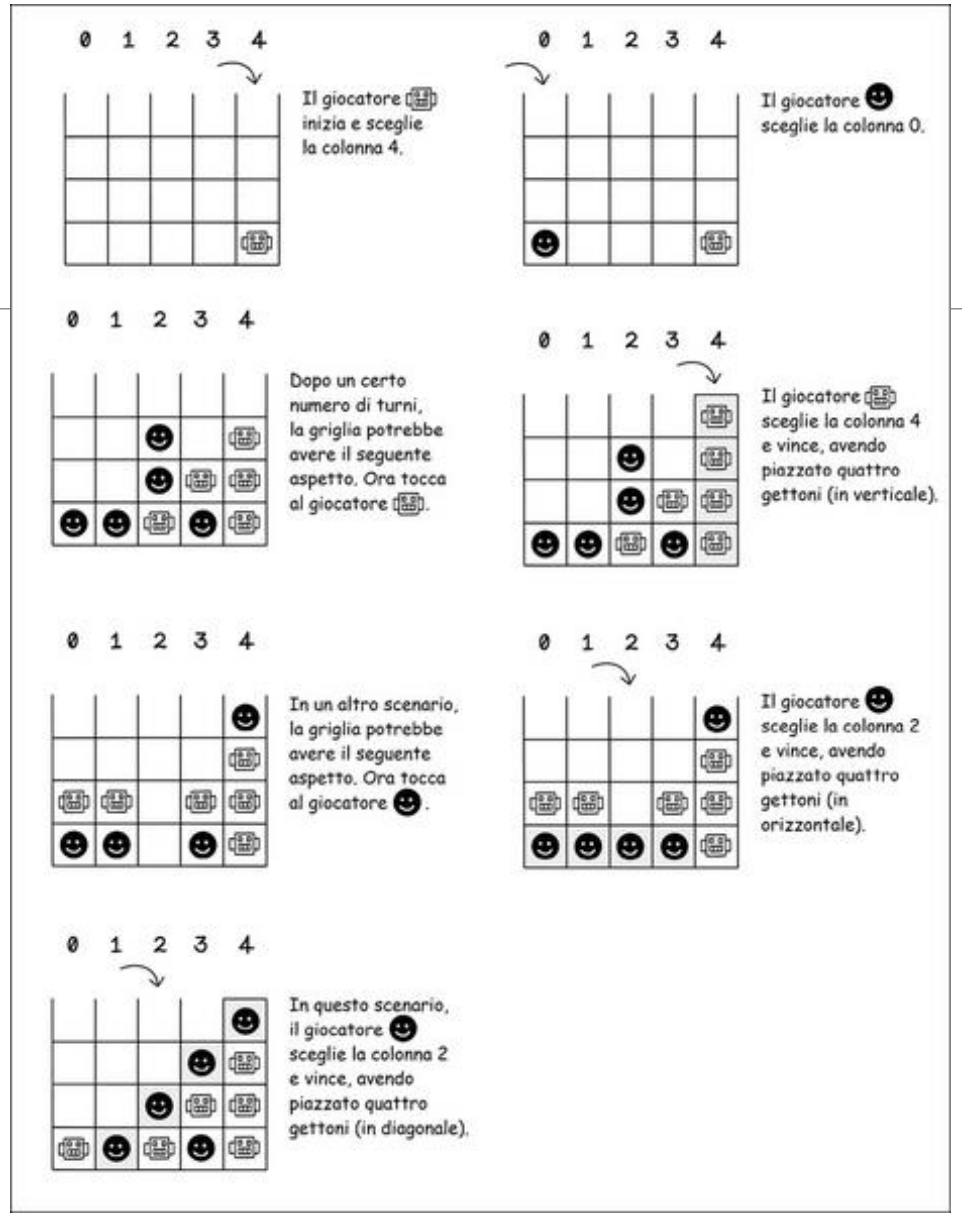
Ricerca avversaria: cercare soluzioni in un ambiente in evoluzione

L'esempio della ricerca nel gioco del labirinto coinvolge un solo attore: il giocatore. L'ambiente è influenzato solo dal giocatore singolo, quindi è solo quel giocatore a generare tutte le possibilità. L'obiettivo finora era quello di massimizzare il vantaggio per il solo giocatore: scegliere percorsi verso un obiettivo con la distanza e il costo minori.

La ricerca avversaria è caratterizzata dall'esistenza di un'opposizione o un conflitto. I problemi ad avversari ci richiedono di anticipare, comprendere e contrastare le azioni dell'avversario nel perseguimento di un obiettivo. Esempi di problemi ad avversari includono giochi a turni per due giocatori come Tris (Tic-Tac-Toe) e Forza quattro (Connect Four). I giocatori, a turno, hanno l'opportunità di cambiare lo stato dell'ambiente di gioco a loro favore. Un insieme di regole determina come l'ambiente può essere modificato e quali sono gli stati vincenti e finali.

Un semplice problema ad avversari

Questo paragrafo utilizza il gioco Forza quattro per esplorare i problemi ad avversari. Forza quattro (Figura 3.11) è un gioco costituito da una griglia in cui i giocatori, a turno, rilasciano gettoni in una colonna. I gettoni si accumulano e vince il giocatore che riesca a creare sequenze di quattro gettoni adiacenti (in verticale, orizzontale o diagonale) del suo colore. Se la griglia è piena, senza vincitori, la partita termina in parità.


0 1 2 3 4
Il giocatore inizia e sceglie la colonna 4.
0 1 2 3 4
Il giocatore sceglie la colonna 0.
0 1 2 3 4
Dopo un certo numero di turni, la griglia potrebbe avere il seguente aspetto. Ora tocca al giocatore.
0 1 2 3 4
Il giocatore sceglie la colonna 4 e vince, avendo piazzato quattro gettoni (in verticale).
0 1 2 3 4
In un altro scenario, la griglia potrebbe avere il seguente aspetto. Ora tocca al giocatore.
0 1 2 3 4
Il giocatore sceglie la colonna 2 e vince, avendo piazzato quattro gettoni (in orizzontale).
0 1 2 3 4
In questo scenario, il giocatore sceglie la colonna 2 e vince, avendo piazzato quattro gettoni (in diagonale).

Ricerca min-max: simula le azioni e sceglie il futuro migliore

La ricerca min-max ha lo scopo di costruire un albero dei risultati possibili in base alle mosse che ogni giocatore potrebbe fare, e favorire percorsi vantaggiosi per un giocatore evitando i percorsi favorevoli all'avversario. Per fare ciò, questo tipo di ricerca simula le possibili mosse e assegna un punteggio a ogni stato in base a un'euristica dopo aver effettuato la rispettiva mossa. La ricerca min-max tenta di scoprire quanti più stati possibili in futuro; ma a causa dei limiti di memoria e calcolo, scoprire l'intero albero del gioco potrebbe non essere possibile, quindi limita la ricerca a una determinata profondità. La ricerca min-max simula i turni di ciascun giocatore, quindi la profondità specificata è direttamente legata al numero di turni fra i due giocatori. Una profondità di 4, per esempio, prevede due turni per ogni giocatore. Il giocatore A fa una mossa, il giocatore B fa una mossa, il giocatore A fa un'altra mossa e il giocatore B fa un'altra mossa.

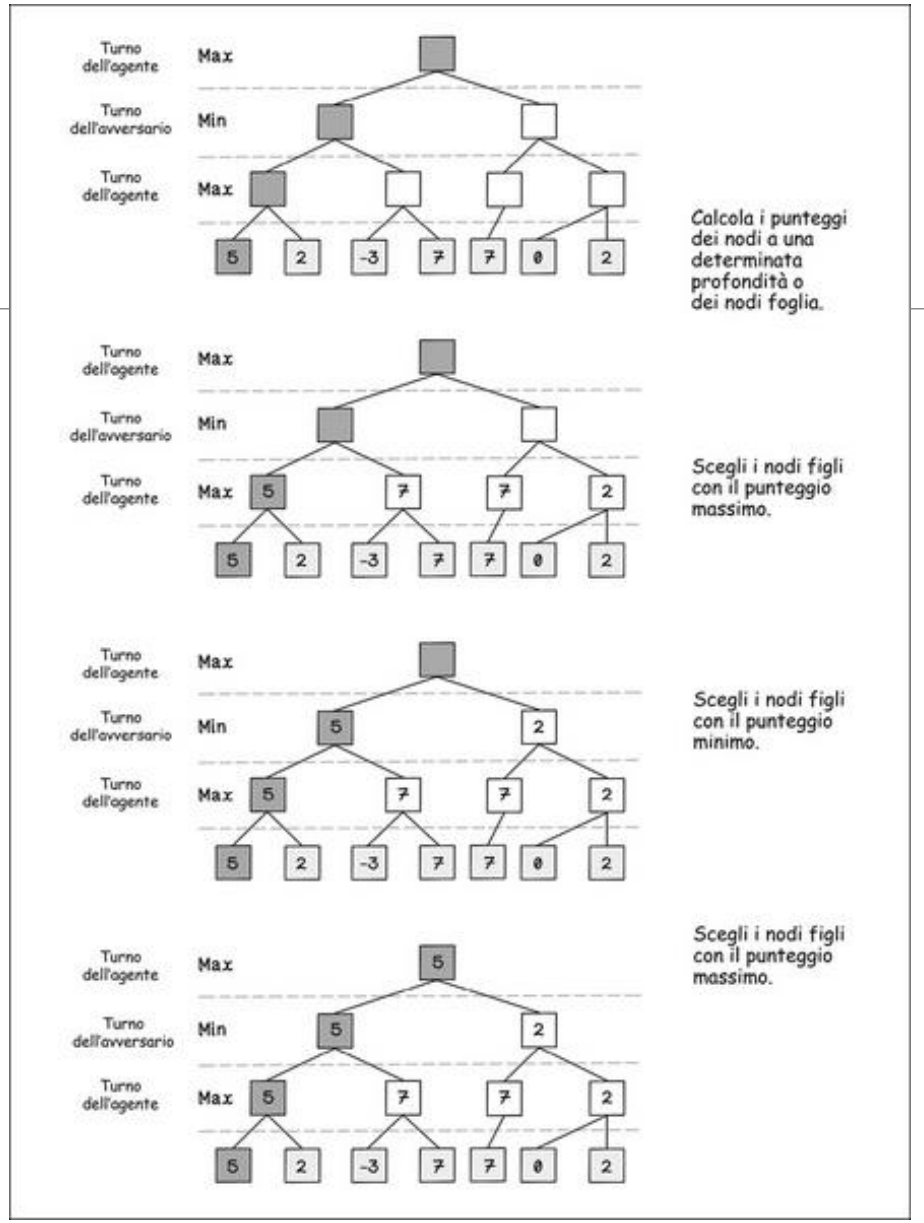
Euristica

L'algoritmo min-max utilizza un punteggio euristico per prendere le sue decisioni. Questo punteggio è definito da una determinata euristica e non è appreso dall'algoritmo. Dato un determinato stato di gioco, ogni possibile risultato valido di una mossa a partire da quello stato sarà un nodo figlio dell'albero del gioco.

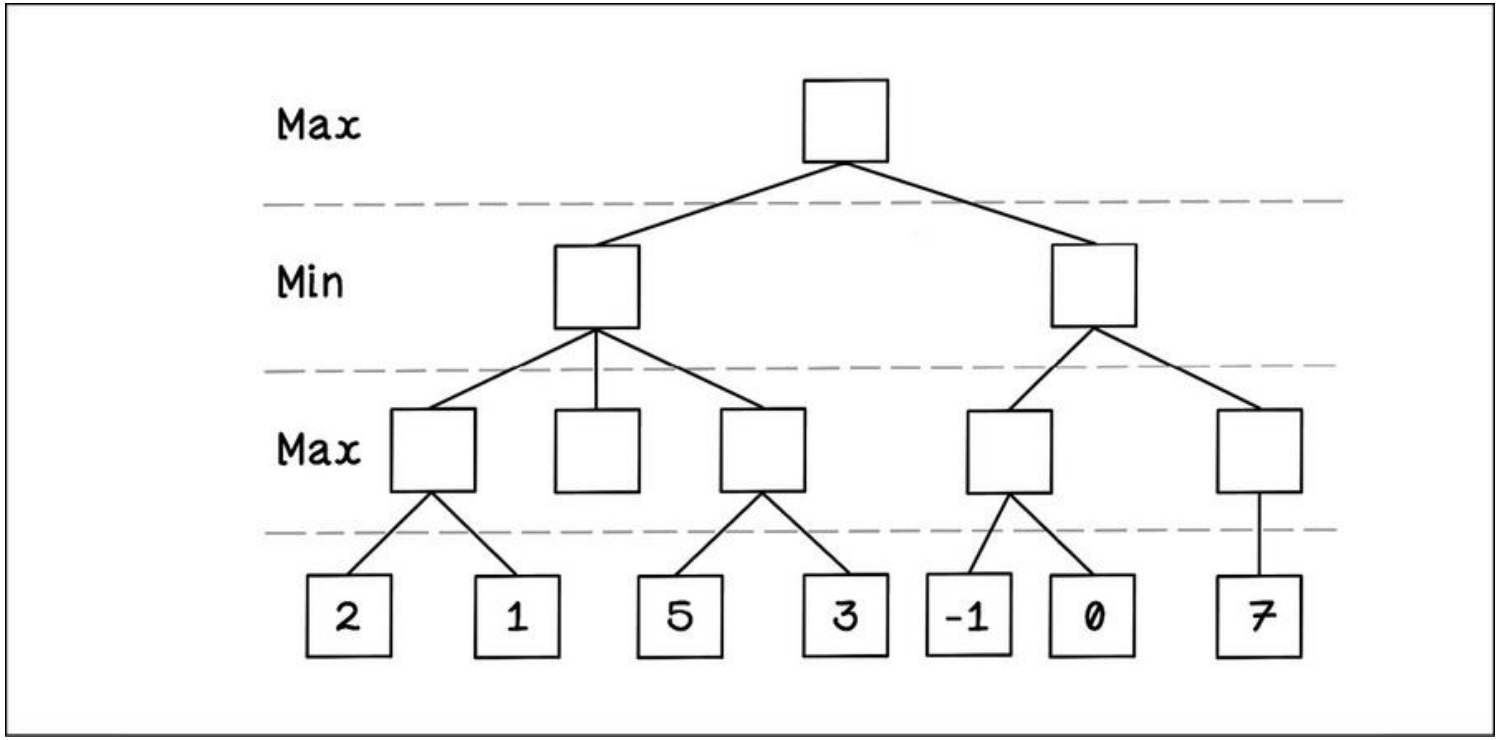
Supponiamo di avere un'euristica che fornisce un punteggio in cui i numeri positivi sono migliori dei numeri negativi. Simulando ogni possibile mossa valida, l'algoritmo di ricerca min-max cerca di minimizzare le mosse in cui l'avversario avrà un vantaggio o uno stato vincente e di massimizzare le mosse che danno all'agente un vantaggio o uno stato vincente.

La Figura 3.12 illustra un albero di ricerca min-max. In questa figura, i nodi foglia sono gli unici nodi nei quali viene calcolato il punteggio euristico, poiché questi stati indicano un vincitore o un pareggio. Gli altri nodi dell'albero indicano gli stati in corso. A partire dalla profondità in cui viene calcolata l'euristica e procedendo verso l'alto, viene scelto il nodo figlio che offre il punteggio minimo o il nodo figlio con il punteggio massimo, a seconda di chi muoverà al prossimo turno. Partendo dall'alto, l'algoritmo tenta di massimizzare il punteggio dell'agente; a turni alterni l'intenzione cambia, perché l'obiettivo è quello di massimizzare il punteggio per l'agente e minimizzare il punteggio per l'avversario.

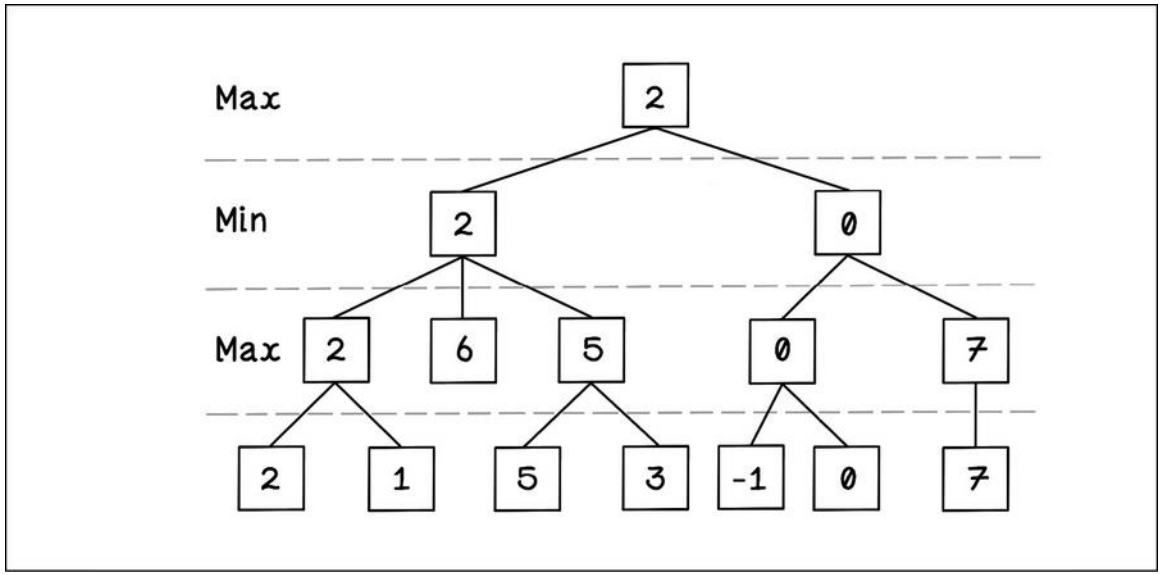
Poiché l'algoritmo di ricerca min-max simula i risultati possibili, nei giochi che offrono più scelte, l'albero del gioco esplode, e così diventa rapidamente troppo costoso, dal punto di vista computazionale, esplorare l'intero albero. Nel semplice esempio di Forza quattro giocato su un tabellone 5 × 4, il numero di possibilità rende già inefficiente l'esplorazione di ogni turno dell'intero albero di gioco


Turno dell'agente
Max
Turno dell'avversario
Min
Turno dell'agente
Max
5 2 -3 7 7 0 2
Calcola i punteggi dei nodi a una determinata profondità o dei nodi foglia.
Turno dell'agente
Max
Turno dell'avversario
Min
Turno dell'agente
Max 5 7 7 2
5 2 -3 7 7 0 2
Scegli i nodi figli con il punteggio massimo.
Turno dell'agente
Max
Turno dell'avversario
Min 5 2
Turno dell'agente
Max 5 7 7 2
5 2 -3 7 7 0 2
Scegli i nodi figli con il punteggio minimo.
Turno dell'agente
Max 5
Turno dell'avversario
Min 5 2
Turno dell'agente
Max 5 7 7 2
5 2 -3 7 7 0 2
Scegli i nodi figli con il punteggio massimo.

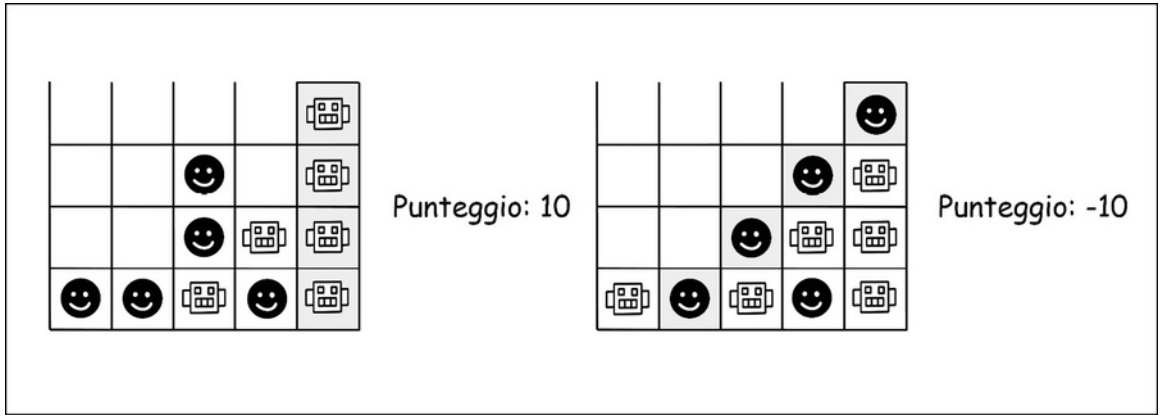
Esercizio: quali valori si propagherebbero nel caso del seguente albero min-max?



soluzione



Per utilizzare la ricerca min-max nell'esempio Forza quattro, l'algoritmo valuta essenzialmente tutte le mosse possibili da uno stato di gioco corrente; quindi determina tutte le possibili mosse da ciascuno di questi stati, finché non trova il percorso più favorevole. Gli stati del gioco che si traducono in una vittoria per l'agente restituiscono un punteggio 10 e gli stati che si traducono in una vittoria per l'avversario restituiscono un punteggio -10. La ricerca min-max cerca di massimizzare il punteggio positivo per l'agente



Sebbene il diagramma di flusso per l'algoritmo di ricerca min-max sembri complesso a causa delle sue dimensioni, in realtà non lo è. Quello che "gonfia" in grafico è il numero di condizioni che controllano se lo stato corrente deve essere massimizzato o minimizzato.

Esaminiamo il flusso dell'algoritmo di ricerca min-max.

• Dato uno stato di gioco, indipendentemente dal fatto che la modalità corrente sia di minimizzazione o di massimizzazione e data una profondità corrente, l'algoritmo può iniziare. È importante comprendere gli input dell'algoritmo, poiché l'algoritmo di ricerca min-max è ricorsivo. Un algoritmo ricorsivo richiama se stesso in uno o più passaggi. È importante che un algoritmo ricorsivo abbia una condizione di uscita per impedirgli di richiamare se stesso indefinitamente.

• Lo stato attuale è finale o ha profondità 0? Questa condizione determina se lo stato attuale del gioco è uno stato terminale o se è stata raggiunta la profondità desiderata. Uno stato è terminale se uno dei giocatori ha vinto o la partita è patta. Il punteggio 10 rappresenta una vittoria per l'agente, il punteggio -10 rappresenta una vittoria per l'avversario e il punteggio 0 indica un pareggio. Viene specificata una profondità, perché l'attraversamento dell'intero albero delle possibilità a coprire tutti gli stati finali è computazionalmente infattibile e richiederà troppo tempo su un normale computer. Specificando una profondità, l'algoritmo ha la possibilità di guardare al futuro per determinare se esiste uno stato terminale.

• Restituisci il punteggio corrente e l'ultima mossa. Il punteggio per lo stato corrente viene restituito se lo stato corrente è uno stato di gioco terminale o se è stata raggiunta la profondità specificata.

• La modalità corrente è MAX? Se l'iterazione corrente dell'algoritmo è nello stato di massimizzazione, tenta di massimizzare il punteggio per l'agente.

• Imposta il miglior punteggio noto a +∞. Se la modalità corrente è quella di minimizzare il punteggio, il punteggio migliore è impostato a infinito positivo, perché sappiamo che i punteggi restituiti dagli stati del gioco saranno comunque inferiori. Nell'implementazione effettiva, viene utilizzato un numero molto elevato al posto dell'infinito.

• Imposta il miglior punteggio noto a -∞. Se la modalità corrente è quella di massimizzare il punteggio, il punteggio migliore è impostato a infinito negativo, perché sappiamo che i punteggi restituiti dagli stati del gioco saranno sempre maggiori. Nell'implementazione effettiva, viene utilizzato un numero negativo molto elevato al posto dell'infinito.

Ottieni tutte le mosse possibili, dato lo stato attuale del gioco. Questo passaggio specifica un elenco di possibili mosse che possono essere effettuate, dato lo stato attuale del gioco. A mano a mano che il gioco procede, non tutte le mosse disponibili all'inizio potrebbero essere ancora disponibili. Nell'esempio Forza quattro, una colonna può essere già tutta piena; pertanto, una mossa che selezionasse quella colonna non sarebbe valida.

• Vi sono altre mosse valide? Se eventuali mosse possibili non sono ancora state simulate e non ci sono altre mosse valide da eseguire, l'algoritmo restituisce la mossa migliore in quell'istanza della chiamata a funzione.

• Copia lo stato corrente del gioco come game_n. È necessaria una copia dello stato di gioco corrente per eseguire simulazioni delle possibili mosse future.

• Simula applicando la mossa allo stato del gioco game_n. Questo passaggio applica l'attuale mossa allo stato di gioco copiato.

Imposta best_n come risultato dell'esecuzione ricorsiva di questo algoritmo. Ecco dove entra in gioco la ricorsione. best_n è una variabile utilizzata per memorizzare la prossima mossa migliore: stiamo facendo in modo che l'algoritmo esplori le possibilità future da questa mossa.

• Se la modalità corrente è MAX? Quando la chiamata ricorsiva restituisce un miglior candidato, questa condizione determina se la modalità corrente è quella di massimizzare il punteggio.

• best_n è minore del miglior stato noto? Questo passaggio determina se l'algoritmo ha trovato un punteggio migliore di quello trovato in precedenza, se la modalità è quella di massimizzare il punteggio.

• best_n è maggiore di best_n conosciuto? Questo passaggio determina se l'algoritmo ha trovato un punteggio migliore di quello trovato in precedenza, se la modalità è quella di minimizzare il punteggio.

• Imposta il migliore stato noto come best_n. Se viene trovato il nuovo miglior punteggio, lo imposta come punteggio migliore noto.

Pseudocodice

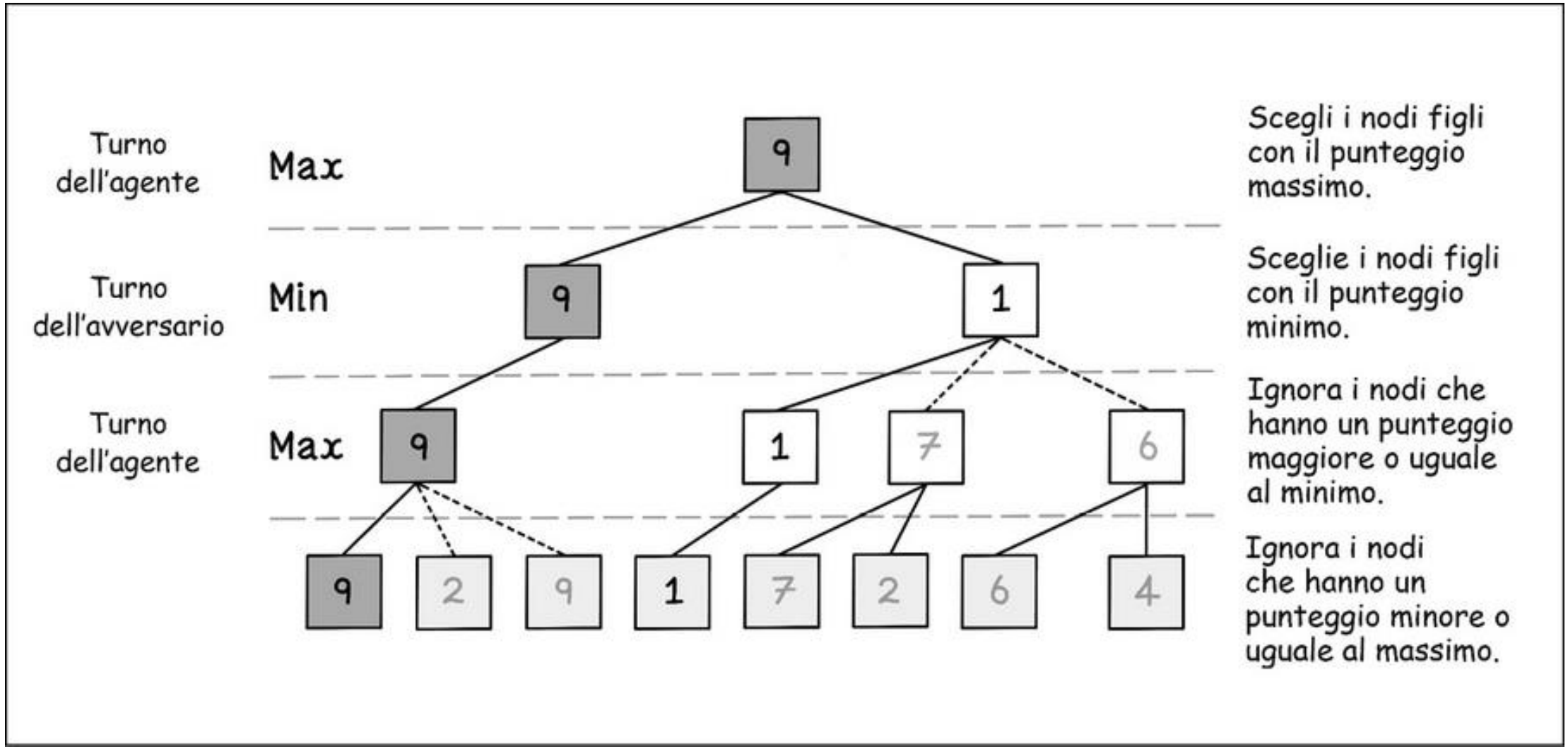
L’algoritmo di ricerca min-max è implementato come una funzione ricorsiva. Alla funzione viene fornito lo stato corrente, la profondità desiderata per la ricerca, la modalità di minimizzazione o massimizzazione e l’ultima mossa. L’algoritmo termina restituendo la mossa e il punteggio migliori per ogni nodo figlio a ogni profondità dell’albero. Confrontando il codice con il diagramma di flusso rappresentato nella Figura 3.15, notiamo che le noiose condizioni di controllo se la modalità corrente è di massimizzazione o minimizzazione non sono così evidenti. Nello pseudocodice, 1 o -1 rappresentano rispettivamente l’intenzione di massimizzare o minimizzare. Utilizzando una logica intelligente, il punteggio, le condizioni e gli stati di commutazione migliori possono essere ottenuti tramite il principio della moltiplicazione negativa: un numero negativo moltiplicato per un altro numero negativo dà un risultato positivo. Quindi se -1 indica il turno dell’avversario, moltiplicandolo per -1 si ottiene 1, che indica il turno dell’agente. Per il turno successivo, 1 moltiplicato per -1 dà come risultato -1 per indicare nuovamente il turno dell’avversario:

```
minmax(state, depth, last_move):
  let current_score equal state.get_score
  if current_score is not equal to 0 or state.is_full or depth is equal to 0:

    return new Move(last_move, current_score)
 let best_score equal to min_or_max multiplied by −∞
 let best_move −1
 for each possible choice(0 to 4 in a 5x4 board) as move:
    let neighbor equal to a copy of state
    execute current move on neighbor
    let best_neighbor equal minmax(neighbor, depth −1, min_or_max * -1, move)
    if(best_neighbor.score is greater than best_score and is MAX)
        or (best_neighbor.score is less than best_score and min_or_max is MIN):
        let best_score = best_neighbor.score
        let best_move = best_neighbor.move
 return new Move(best_move, best_score)
```

Potatura alfa-beta: ottimizza esplorando solo i percorsi sensati

La potatura alfa-beta è una tecnica utilizzata con l'algoritmo di ricerca min-max per evitare di esplorare aree dell'albero del gioco che producono soluzioni notoriamente scadenti. Questa tecnica ottimizza l'algoritmo di ricerca min-max per risparmiare calcoli, poiché i percorsi non sensati vengono ignorati. Poiché sappiamo che l'albero di gioco dell'esempio con Forza quattro esplode, è evidente che il fatto di ignorare selettivamente un maggior numero di percorsi migliorerà significativamente le prestazioni


Turno dell'agente
Max
Turno dell'avversario
Min
Turno dell'agente
Max
9
9
1
9
1
7
6
9
2
9
1
7
2
6
4
Scegli i nodi figli con il punteggio massimo.
Sceglie i nodi figli con il punteggio minimo.
Ignora i nodi che hanno un punteggio maggiore o uguale al minimo.
Ignora i nodi che hanno un punteggio minore o uguale al massimo.

L’algoritmo di potatura alfa-beta funziona memorizzando il miglior punteggio per il giocatore che massimizza come alfa e il miglior punteggio per il giocatore che minimizza come beta. Inizialmente, alfa è impostato a -∞ e beta è impostato a ∞, il peggior punteggio per ogni giocatore. Se il miglior punteggio del giocatore che minimizza è minore del miglior punteggio del giocatore che massimizza, è logico che altri percorsi figli dei nodi già visitati non influenzeranno il miglior punteggio.

I blocchi evidenziati sono i passaggi aggiuntivi nel flusso dell'algoritmo di ricerca min-max.

• La modalità corrente è MAX? Di nuovo, determina se l'algoritmo sta attualmente tentando di massimizzare o minimizzare il punteggio.

• best_n è maggiore o uguale ad alfa? Se la modalità corrente è quella di massimizzare il punteggio e il miglior punteggio corrente è maggiore o uguale ad alfa, nei nodi figli di quel nodo non si troveranno punteggi migliori, quindi l'algoritmo può ignorare quel nodo.

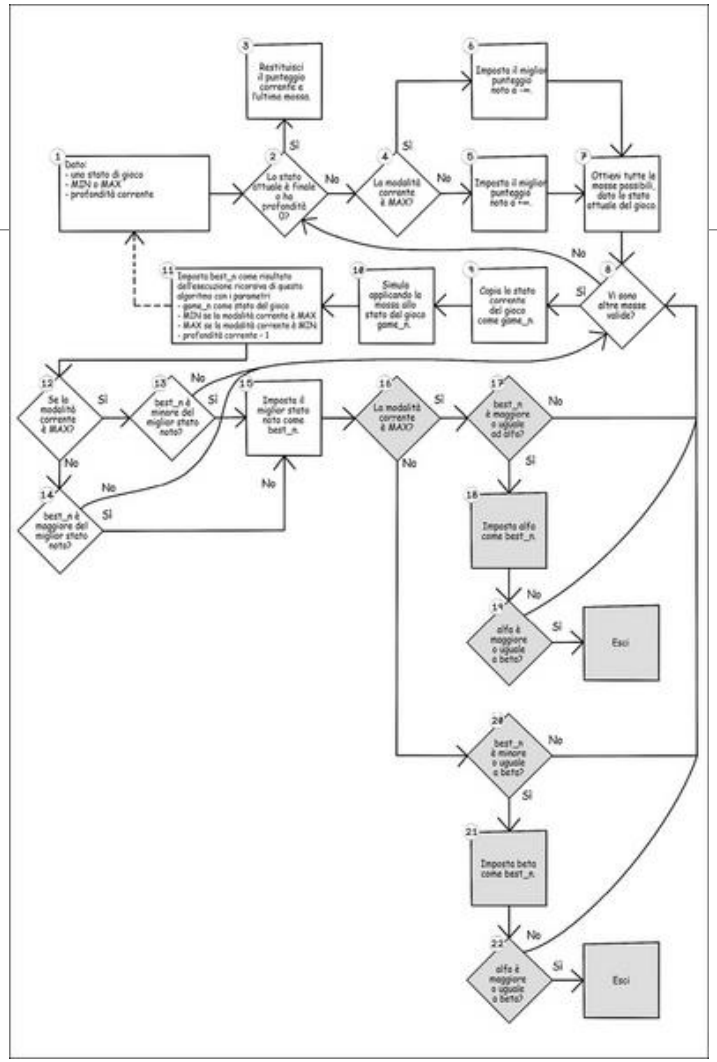
• Imposta alfa come best_n. Imposta la variabile alfa come best_n.

• alfa è maggiore o uguale a beta? Il punteggio non è migliore degli altri punteggi trovati, e il resto dell'esplorazione di quel nodo può essere "potato".

• best_n è minore o uguale a beta? Se la modalità corrente è quella di minimizzare il punteggio e il miglior punteggio corrente è minore o uguale a beta, nei nodi figli di quel nodo non si troveranno punteggi migliori, quindi l'algoritmo può ignorare quel nodo.

• Imposta beta come best_n. Imposta la variabile beta come best_n.

• alfa è maggiore o uguale a beta? Il punteggio non è migliore degli altri punteggi trovati, e il resto dell'esplorazione di quel nodo può essere "potato".


1
Dato:
- uno stato di gioco
- MIN o MAX
- profondità corrente
2
Lo stato attuale è finale o ha profondità 0?
Sì
No
3
Restituisci il punteggio corrente e l'ultima mossa
4
La modalità corrente è MAX?
Sì
No
5
Imposta il miglior punteggio noto a +∞
6
Imposta il miglior punteggio noto a -∞
7
Ottieni tutte le mosse possibili, dato lo stato attuale del gioco
8
Vi sono altre mosse valide?
Sì
No
9
Copia lo stato corrente del gioco come game_n
10
Simula applicando la mossa allo stato del gioco game_n
11
Imposta best_n come risultato dell'esecuzione ricorsiva di questo algoritmo con i parametri
- game_n come stato del gioco
- MIN se la modalità corrente è MAX
- MAX se la modalità corrente è MIN
- profondità corrente - 1
12
Se la modalità corrente è MAX?
Sì
No
13
best_n è minore del miglior stato noto?
Sì
No
14
best_n è maggiore del miglior stato noto?
Sì
No
15
Imposta il miglior stato noto come best_n
16
La modalità corrente è MAX?
Sì
No
17
best_n è maggiore o uguale ad alfa?
Sì
No
18
Imposta alfa come best_n
19
alfa è maggiore o uguale a beta?
Sì
No
Esci
20
best_n è minore o uguale a beta?
Sì
No
21
Imposta beta come best_n
22
alfa è maggiore o uguale a beta?
Sì
No
Esci

Pseudocodice

Lo pseudocodice per ottenere la potatura alfa-beta è in gran parte lo stesso del codice per la ricerca min-max, con l'aggiunta della registrazione e gestione dei valori alfa e beta mentre viene attraversato l'albero. Notate che quando viene selezionato il minimo (MIN) la variabile min_or_max è -1, e quando viene selezionato il massimo (MAX), la variabile min_or_max è 1:

```
minmax_ab_pruning(state, depth, min_or_max, last_move, alpha, beta):
  let current_score equal state.get_score
  if current_score is not equal to 0 or state.is_full or depth is equal to 0:
      return new Move(last_move, current_score)

let best_score equal to min_or_max multiplied by −∞
let best_move = −1
for each possible choice(0 to 4 in a 5x4 board) as move:
    let neighbor equal to a copy of state
    execute current move on neighbor
    let best_neighbor equal minmax(neighbor, depth −1, min_or_max * −1, move, alpha, beta)
    if (best_neighbor.score is greater than best_score and min_or_max is MAX)
      or (best_neighbor.score is less than best_score and min_or_max is MIN):
        let best_score = best_neighbor.score
        let best_move = best_neighbor.move
        if best_score >= alpha:
            alpha = best_score
        if best_score <= beta:
            beta = best_score
        if alpha >= beta:
            break
return new Move(best_move, best_score)
```

Casi d’uso per gli algoritmi di ricerca avversaria

Gli algoritmi di ricerca informata sono versatili e utili in casi d’uso come i seguenti.

• Creazione di agenti per giochi a turni con informazioni note: in alcuni giochi, due o più giocatori operano nello stesso ambiente. Vi sono implementazioni di successo di scacchi, dama e altri giochi classici. I giochi con informazioni note non hanno informazioni nascoste o interventi del caso.

• Creazione di agenti per giochi a turni con informazioni imperfette: in questi giochi esistono opzioni future sconosciute, come nel poker e in Scarabeo.

• Ricerca avversaria e ottimizzazione a colonia di formiche (ACO) per l’ottimizzazione dei percorsi: la ricerca avversaria viene utilizzata in combinazione con l’algoritmo ACO (vedi il Capitolo 6) per ottimizzare i percorsi di consegna dei pacchi.