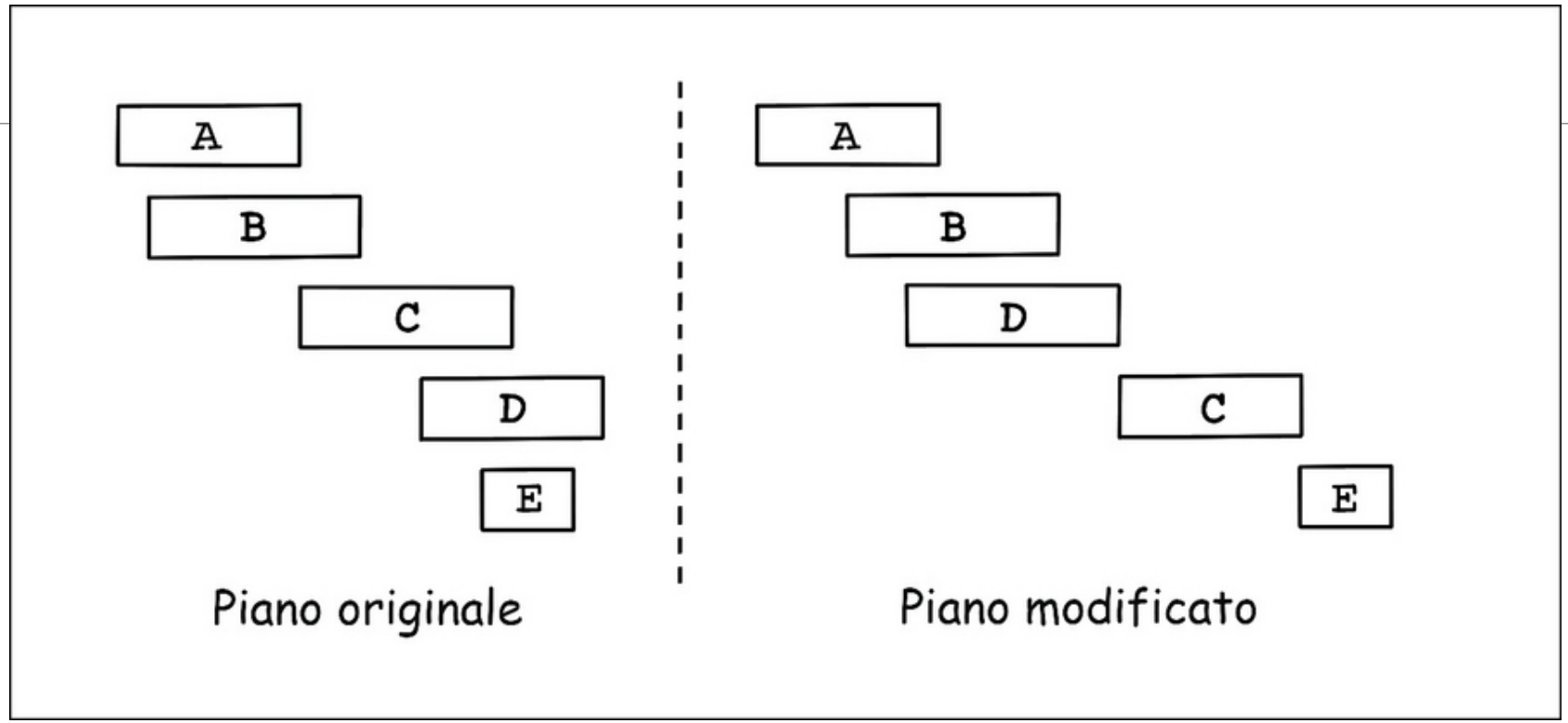
I fondamenti della ricerca

A CURA DI LINO POLO

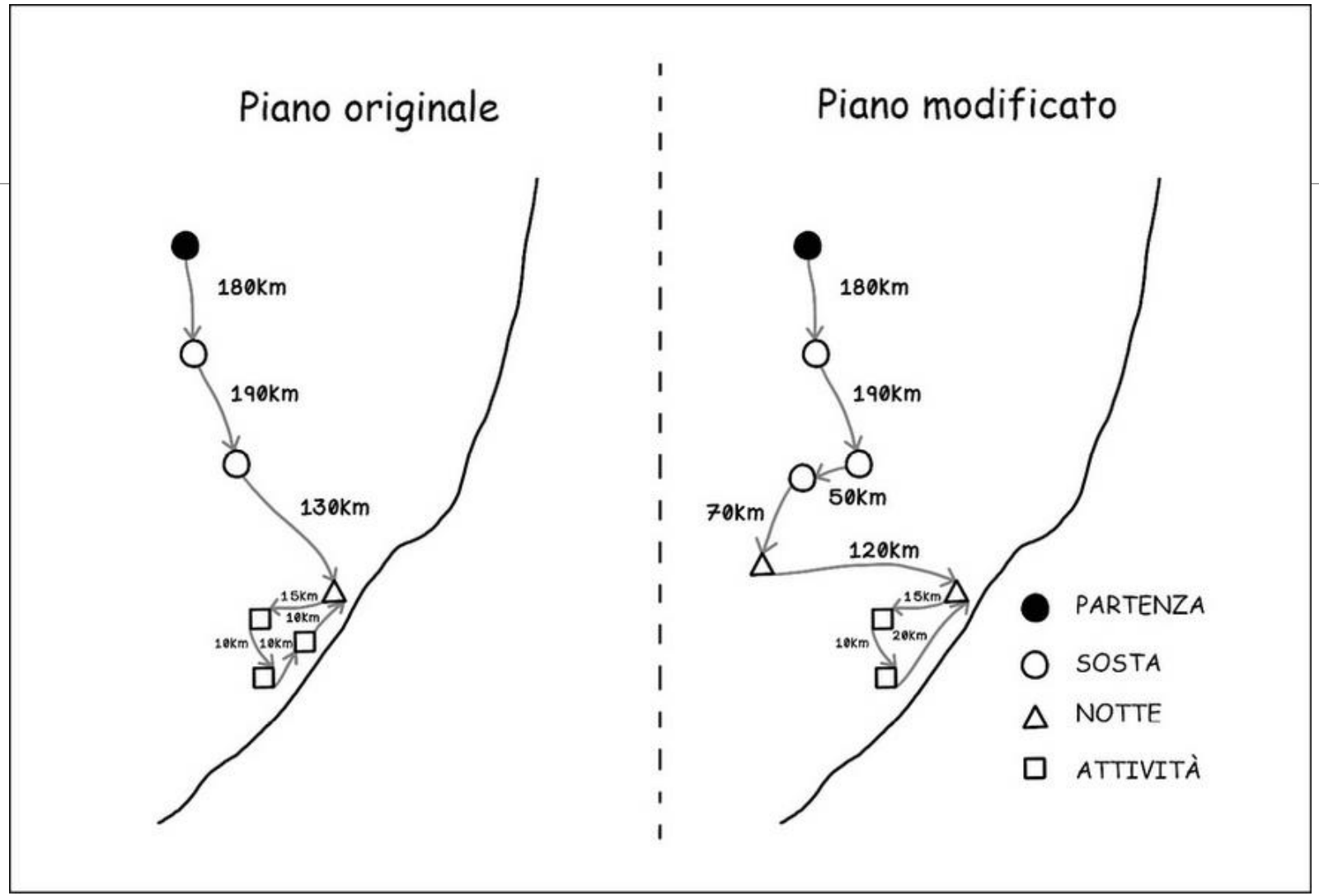
Che cosa sono la pianificazione e la ricerca?

Quando pensiamo a ciò che ci rende intelligenti, consideriamo un attributo importante la capacità di pianificare prima di compiere azioni. Prima di intraprendere un viaggio in un altro Paese, prima di iniziare un nuovo progetto, prima di scrivere il codice di una funzione, svolgiamo una pianificazione. La pianificazione si attua a livelli di dettaglio differenti in contesti differenti per tendere al miglior risultato possibile dovendo svolgere dei compiti per raggiungere determinati obiettivi

I piani raramente funzionano perfettamente nel modo in cui immaginiamo all'inizio di un'impresa. Viviamo in un mondo in cui la realtà cambia costantemente, quindi è impossibile tenere conto di tutte le variabili e le incognite che si presenteranno lungo il percorso. Indipendentemente dalla bontà del piano con cui abbiamo iniziato, deviamo quasi sempre, a causa dei cambiamenti intervenuti nello spazio del problema. Dobbiamo così predisporre un nuovo piano, deviando dal nostro punto attuale e andando avanti, poiché di norma, dopo aver fatto alcuni passi, si verificano eventi imprevisti che richiedono una nuova iterazione della pianificazione per raggiungere gli obiettivi prefissi. Di conseguenza, il piano finale che viene eseguito è solitamente diverso da quello originale.


A
B
C
D
E
Piano originale
A
B
D
C
E
Piano modificato

La ricerca è un modo per guidare la pianificazione creando i passaggi di un piano. Quando pianifichiamo un viaggio, per esempio, cerchiamo strade da percorrere, valutiamo le tappe lungo il percorso e quello che offrono, ricerchiamo alloggi e attività in linea con i nostri gusti e il nostro budget. A seconda dei risultati di queste ricerche, il nostro piano cambia.


Piano originale
180km
190km
130km
15km
10km
10km
10km
Piano modificato
180km
190km
50km
70km
120km
15km
20km
10km
PARTENZA
SOSTA
NOTTE
ATTIVITÀ

Costo del calcolo: lo scopo degli algoritmi intelligenti

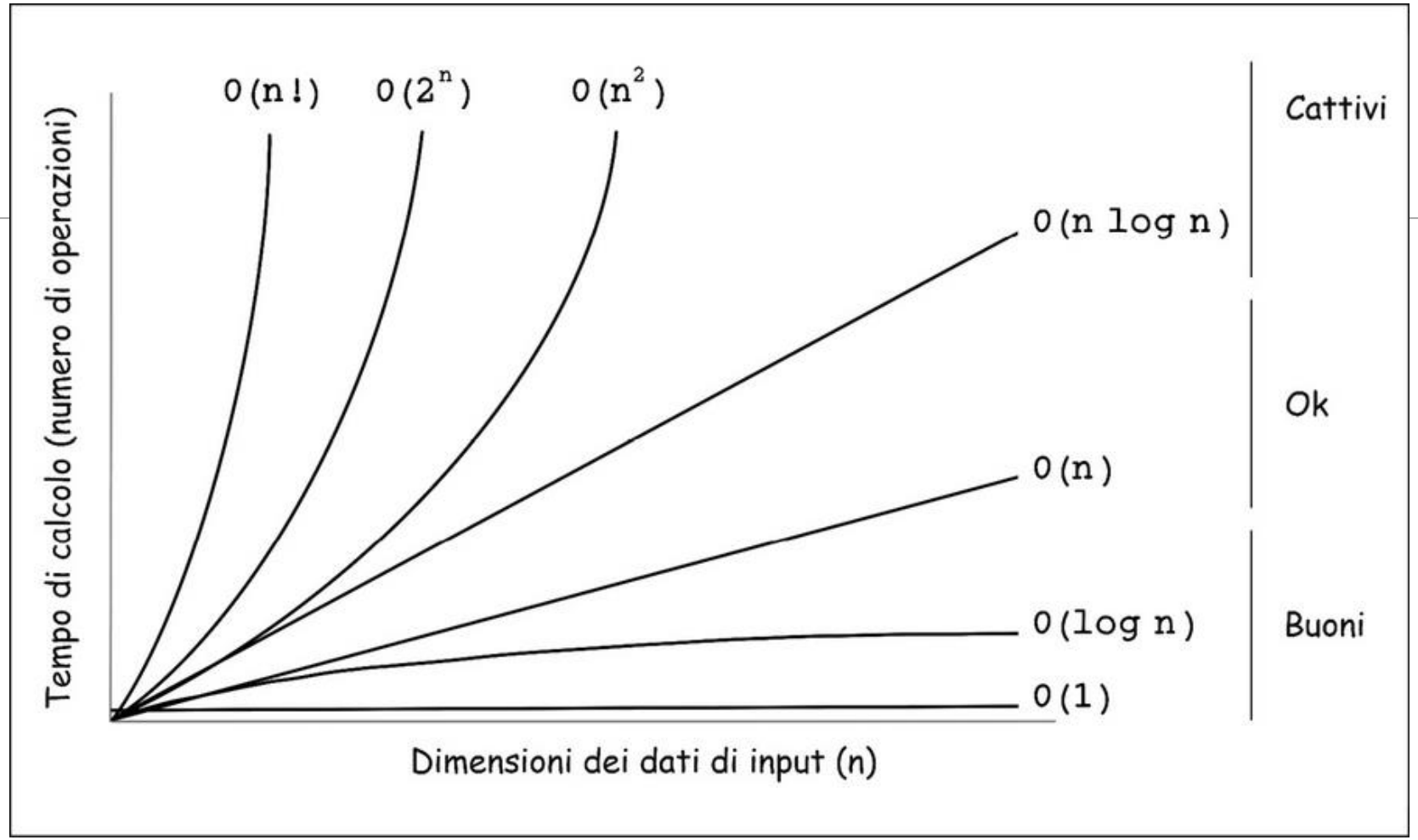
In programmazione, le funzioni sono costituite da operazioni e, dato il modo in cui funzionano i computer, funzioni differenti impiegano tempi di elaborazione differenti. Maggiore è la quantità di calcolo richiesta, più costosa è la funzione. Per descrivere la complessità di una funzione o di un algoritmo viene utilizzata la notazione Big O, che rappresenta il numero di operazioni richieste all'aumentare della dimensione dell'input. Ecco alcuni esempi di problemi e della relativa complessità.

Una singola operazione che stampa Hello World: questa è una singola operazione, quindi il costo del calcolo è O(1).

• Una funzione che esegue un'iterazione su una lista e mostra ciascun elemento: il numero di operazioni dipende dal numero di elementi contenuti nella lista. Il costo è O(n).

• Una funzione che confronta ogni elemento di una lista con ogni elemento di un'altra lista: questa operazione costa $O(n^2)$.

La Figura 2.3 illustra i diversi costi degli algoritmi. Gli algoritmi che richiedono più operazioni all'aumentare della dimensione dell'input sono quelli con le prestazioni peggiori; molto migliori sono gli algoritmi che richiedono un numero il più possibile costante di operazioni all'aumentare del numero di input.

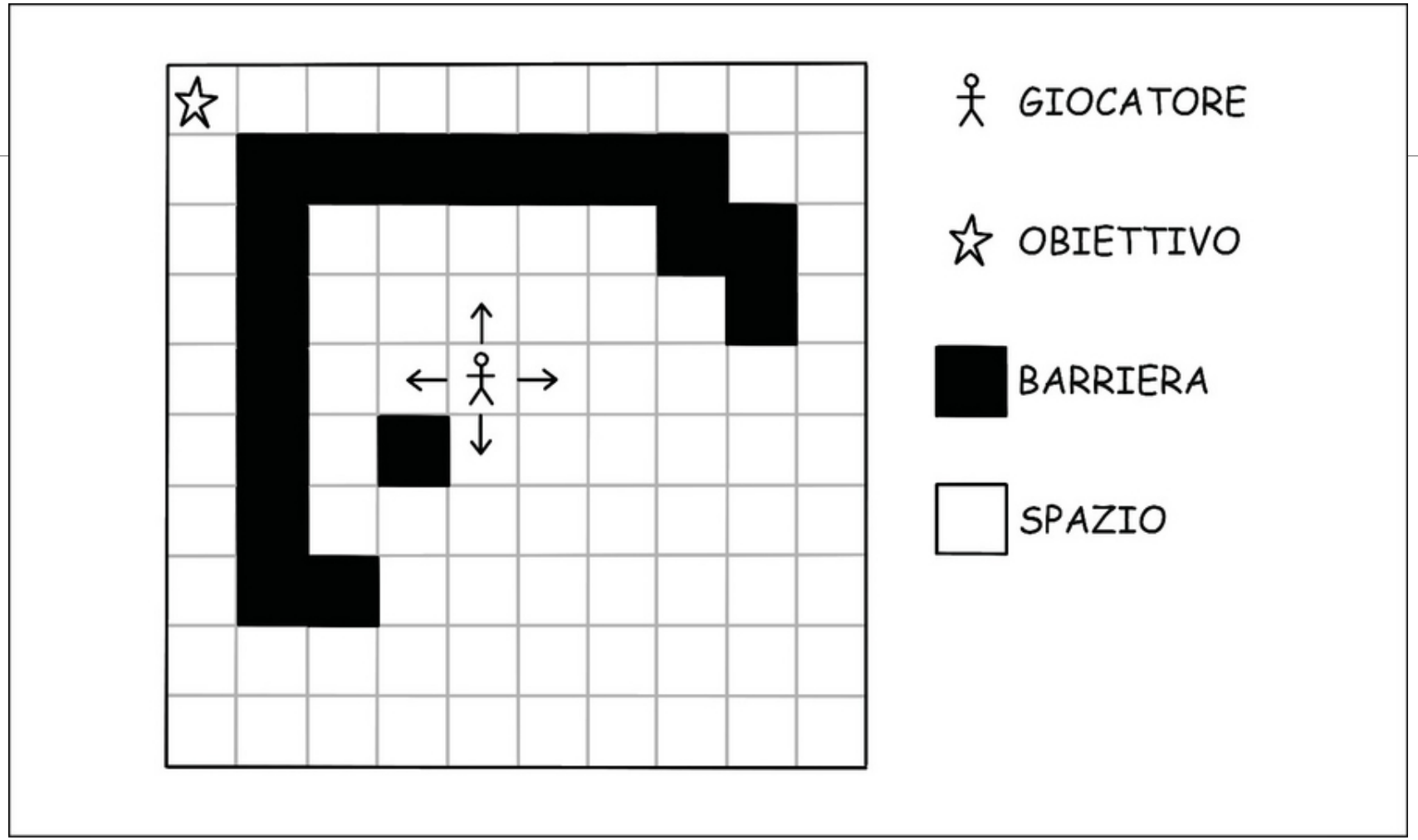

O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n)
O(1)
Cattivi
Ok
Buoni
Tempo di calcolo (numero di operazioni)
Dimensioni dei dati di input (n)

Comprendere il fatto che algoritmi differenti prevedono costi di calcolo differenti è importante, perché esattamente questo è lo scopo di usare algoritmi intelligenti che risolvono i problemi in modo efficace e rapido. Teoricamente, possiamo risolvere quasi tutti i problemi esaminando “a forza bruta” ogni possibile opzione fino a trovare la migliore, ma così facendo il calcolo potrebbe richiedere ore o addirittura anni, il che lo renderebbe inutile per gli scenari del mondo reale.

Problemi risolvibili dagli algoritmi di ricerca

Quasi tutti i problemi che richiedono di prendere una serie di decisioni possono essere risolti con algoritmi di ricerca. A seconda del problema e delle dimensioni dello spazio di ricerca, possono essere impiegati algoritmi differenti per tentare di risolverlo. A seconda dell'algoritmo di ricerca selezionato e della configurazione utilizzata, è possibile trovare la soluzione ottimale o la migliore soluzione disponibile. In altre parole, verrà trovata una buona soluzione, che tuttavia potrebbe non essere la soluzione migliore. Quando parliamo di "buona soluzione" o "soluzione ottimale", ci riferiamo alla capacità della soluzione di affrontare il problema in questione.

Uno scenario in cui gli algoritmi di ricerca sono particolarmente utili è la seguente: siamo in un labirinto e dobbiamo tentare di trovare il percorso più breve per raggiungere un obiettivo. Supponiamo di trovarci in un labirinto quadrato, costituito da un'area di 10 × 10 blocchi. Dobbiamo raggiungere un obiettivo, ma non possiamo attraversare le barriere. L'obiettivo è quello di trovare un percorso verso l'obiettivo, con il minor numero possibile di mosse spostandoci a nord, sud, est o ovest. In questo esempio, il giocatore non può muoversi in diagonale.


GIOCATORE
OBIETTIVO
BARRIERA
SPAZIO

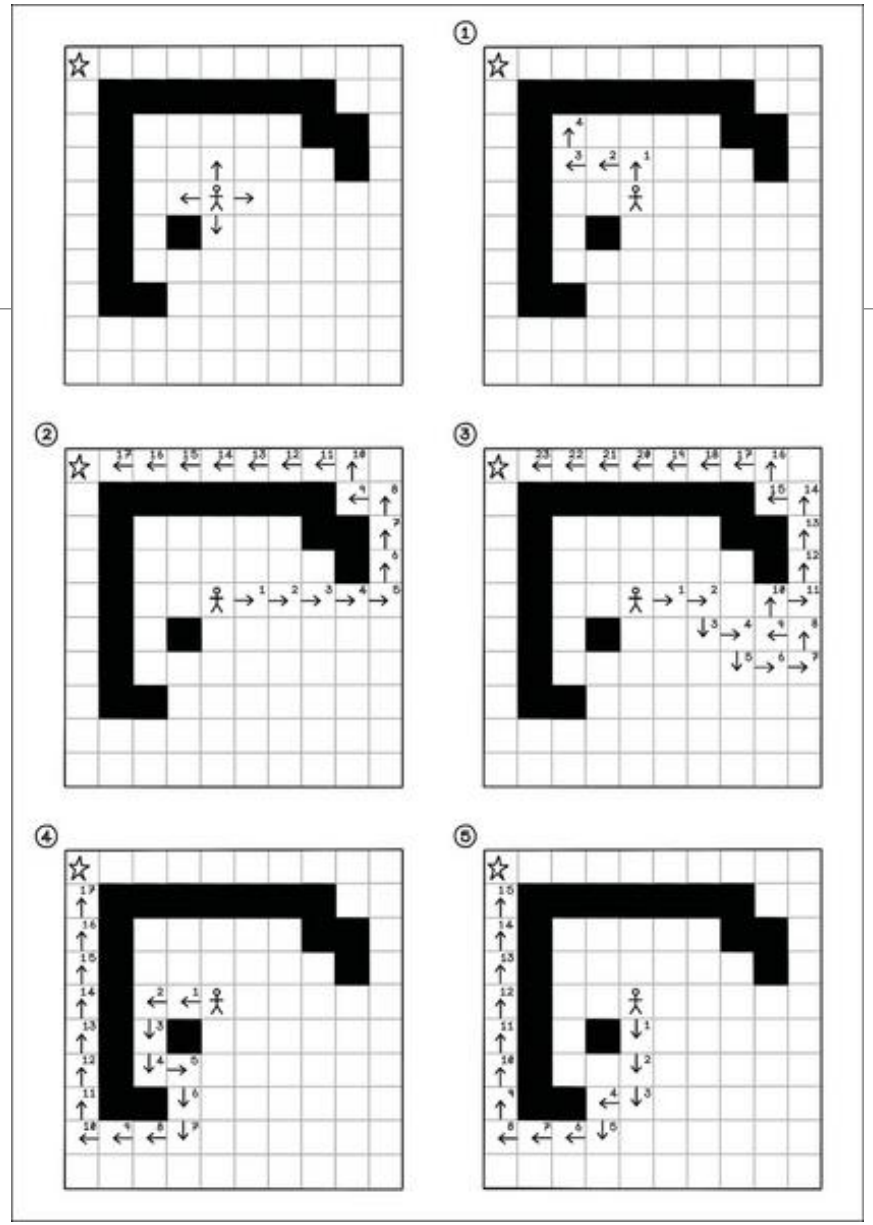
Come possiamo trovare il percorso più breve verso l'obiettivo evitando le barriere? Valutando il problema come esseri umani, possiamo provare ogni possibilità e contare le mosse. Procedendo per tentativi ed errori, possiamo facilmente trovare i percorsi più brevi, dato che questo labirinto è relativamente piccolo.

Usando l'esempio del labirinto, la Figura mostra alcuni possibili percorsi per raggiungere l'obiettivo, anche se notate che non raggiungiamo l'obiettivo nell'opzione 1.

Osservando il labirinto e contando i blocchi in direzioni differenti, possiamo trovare altre soluzioni del problema. Sono stati fatti cinque tentativi per trovare quattro soluzioni di successo su un numero imprecisato di soluzioni. Ci vorrà uno sforzo esaustivo per tentare di calcolare a mano tutte le soluzioni possibili.

- Il tentativo 1 non è una soluzione valida. Impiega 4 azioni e non trova l'obiettivo.
- Il tentativo 2 è una soluzione valida. Impiega 17 azioni per trovare l'obiettivo.
- Il tentativo 3 è una soluzione valida. Impiega 23 azioni per trovare l'obiettivo.
- Il tentativo 4 è una soluzione valida. Impiega 17 azioni per trovare l'obiettivo.
- Il tentativo 5 è la migliore soluzione valida. Impiega 15 azioni per trovare l'obiettivo. Sebbene questo tentativo sia il migliore, è stato trovato per caso.

Se il labirinto fosse molto più grande, come quello rappresentato nella Figura 2.6, ci vorrebbe un’enorme quantità di tempo per calcolare a mano il miglior percorso possibile. Gli algoritmi di ricerca possono aiutarci, in questi casi.

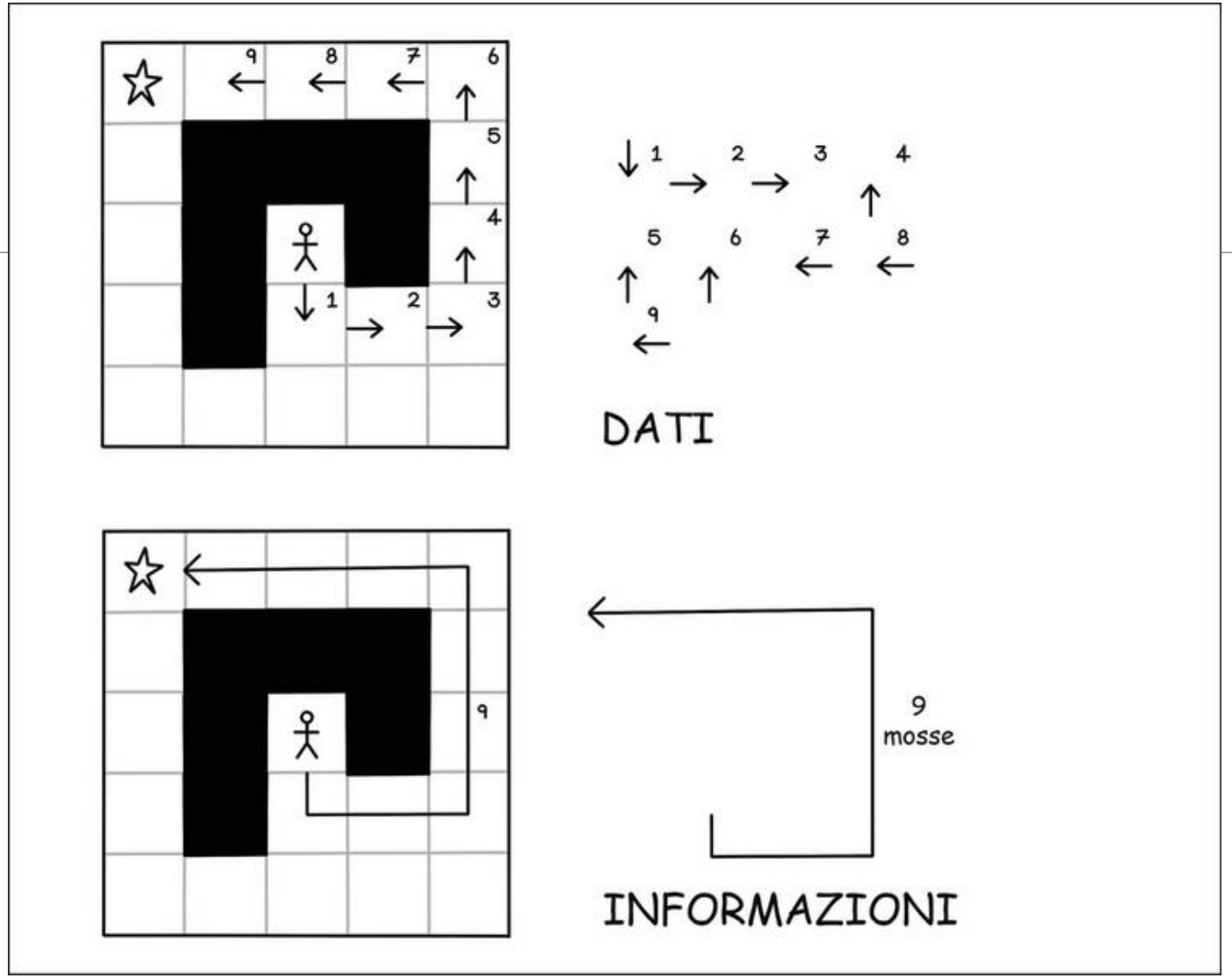


La nostra capacità, come esseri umani, è quella di percepire visivamente il problema, studiarlo e trovare le soluzioni, dati i parametri. Come esseri umani, comprendiamo e interpretiamo i dati e le informazioni in modo astratto. Un computer non è ancora in grado di comprendere le informazioni generalizzate nella loro forma naturale, come facciamo noi. Lo spazio del problema deve essergli rappresentato in una forma che sia applicabile a un calcolo e che possa essere elaborata tramite algoritmi di ricerca.

Rappresentare lo stato: creare una struttura per rappresentare gli spazi dei problemi e le soluzioni

Nel rappresentare i dati e le informazioni in modi comprensibili a un computer, è necessario codificarli in modo che possano essere compresi oggettivamente. Sebbene i dati vengano codificati soggettivamente da chi esegue tale attività, deve esserci un modo conciso e coerente per rappresentarli.

Chiariamo la differenza fra dati e informazioni. I dati sono fatti grezzi riguardanti qualcosa; le informazioni sono interpretazioni di quei fatti che forniscono informazioni sui dati in un determinato dominio. Le informazioni richiedono un contesto e l'elaborazione dei dati per fornire un significato. Per esempio, ogni distanza percorsa nell'esempio del labirinto è un dato, mentre la distanza totale percorsa è un'informazione. A seconda del punto di vista, del livello di dettaglio e del risultato desiderato, classificare qualcosa come un dato o un'informazione può essere una questione soggettiva, dipendente dal contesto e dalla persona/team (Figura 2.7).

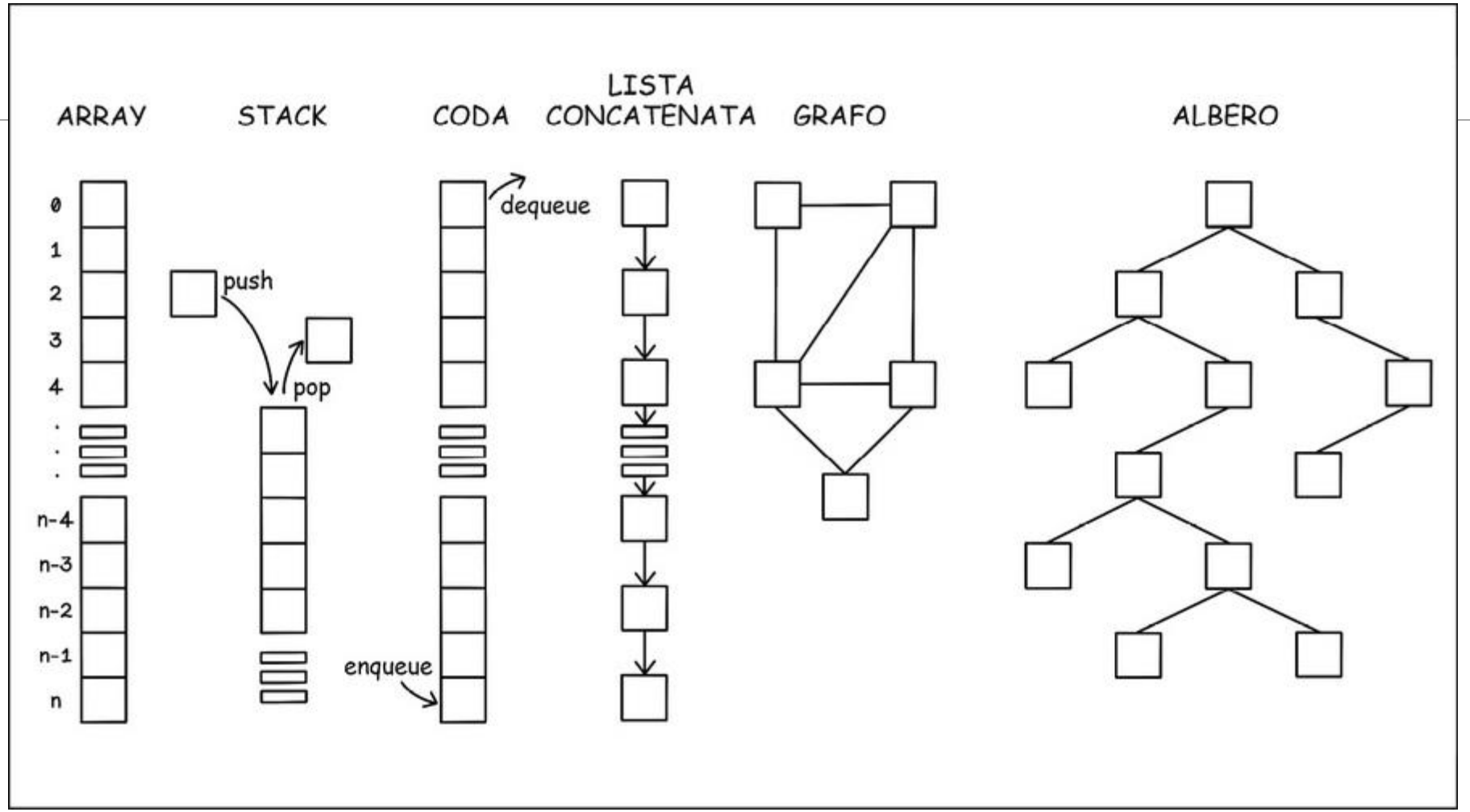

DATI
9
mosse
INFORMAZIONI

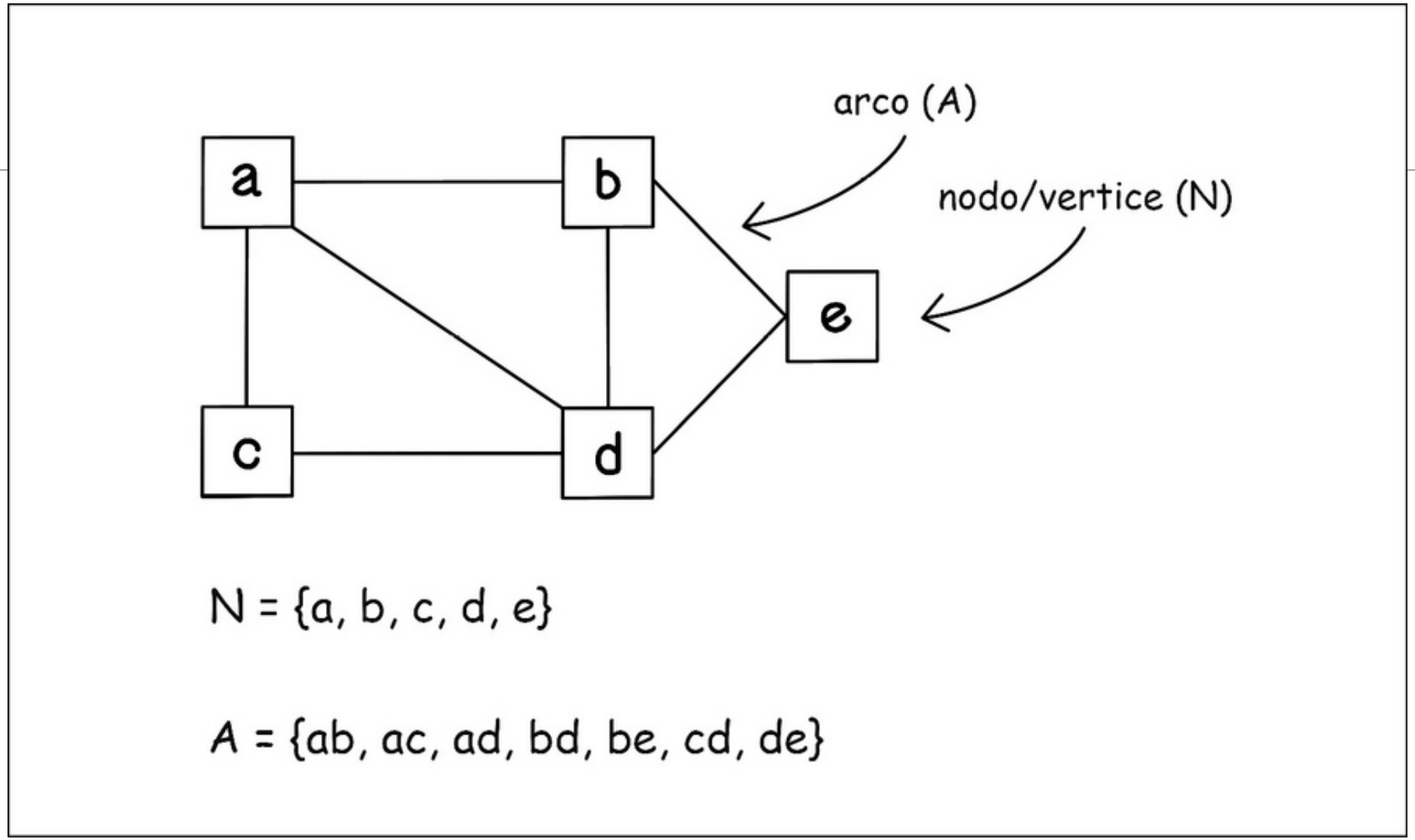
Le strutture di dati sono concetti informatici utilizzati per rappresentare i dati in un modo adatto per consentirne un'elaborazione efficiente da parte degli algoritmi. Una struttura di dati è un tipo di dati astratto, costituito dai dati e dalle operazioni eseguibili su di essi, organizzati in un modo ben preciso. La struttura dei dati che utilizziamo è influenzata dal contesto del problema e dall'obiettivo desiderato.

Un esempio di struttura di dati è l’array, che è semplicemente una sequenza di dati. Tipi di array differenti hanno proprietà differenti che li rendono efficienti per scopi differenti. A seconda del linguaggio di programmazione utilizzato, un array può ospitare valori di un tipo differente o richiedere che ogni valore sia dello stesso tipo; in alcuni casi l’array potrebbe non consentire l’esistenza di valori duplicati. Questi diversi tipi di array di solito hanno nomi differenti. Le caratteristiche e i vincoli delle diverse strutture di dati consentono anche di eseguire i

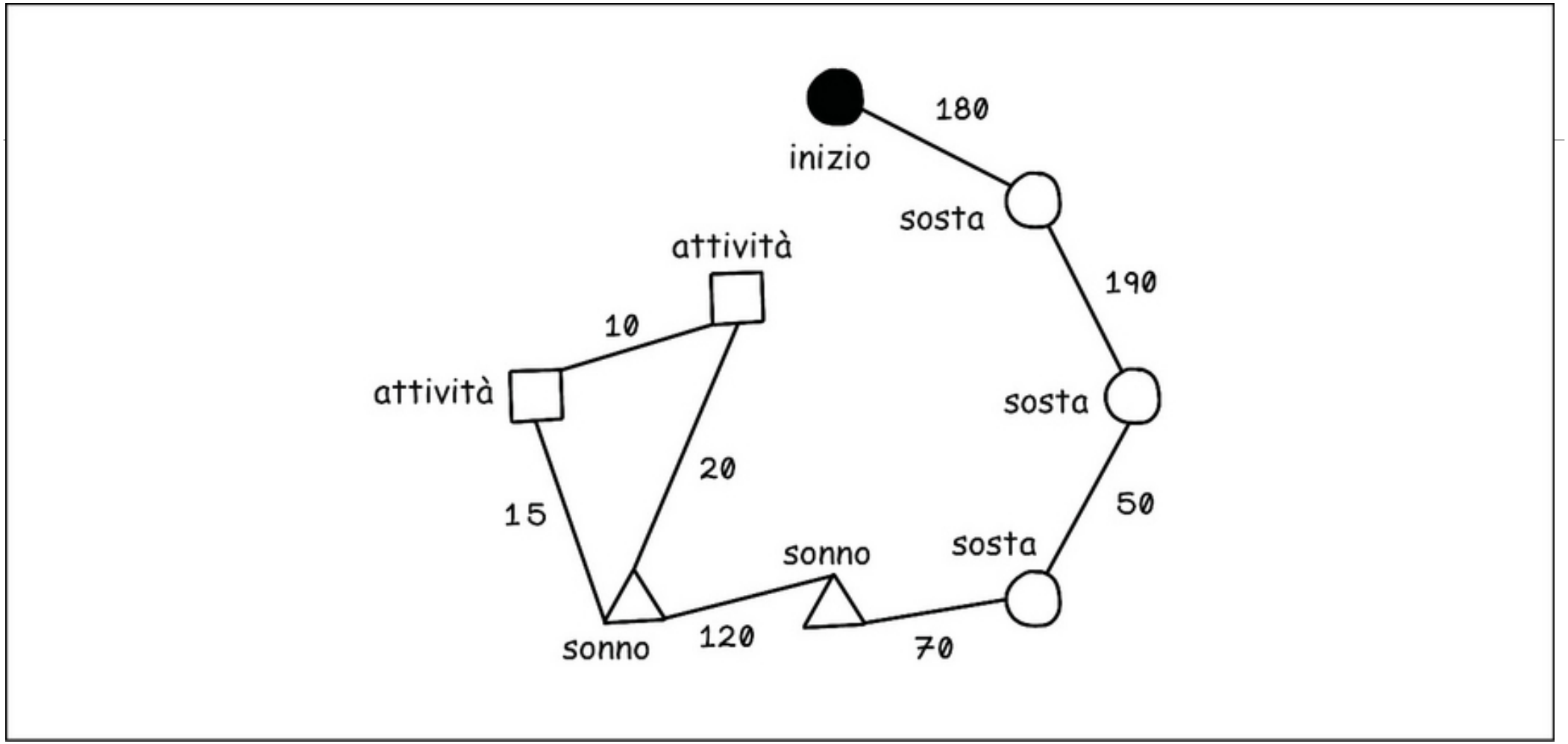
Grafi: rappresentazione dei problemi di ricerca e delle soluzioni

Un grafo è una struttura di dati costituita da vari stati connessi fra loro. Ogni stato di un grafo è chiamato nodo (o talvolta vertice) e una connessione fra due stati è chiamata arco (o lato o spigolo). I grafi derivano dalla teoria dei grafi e vengono utilizzati per modellare le relazioni fra gli oggetti. I grafi sono strutture di dati facili da comprendere, grazie alla possibilità di rappresentarli visivamente e alla loro natura fortemente logica, che li rende ideali per l'elaborazione tramite vari algoritmi


ARRAY
STACK
CODA
LISTA CONCATENATA
GRAFO
ALBERO
0
1
2
3
4
n-4
n-3
n-2
n-1
n
push
pop
dequeue
enqueue

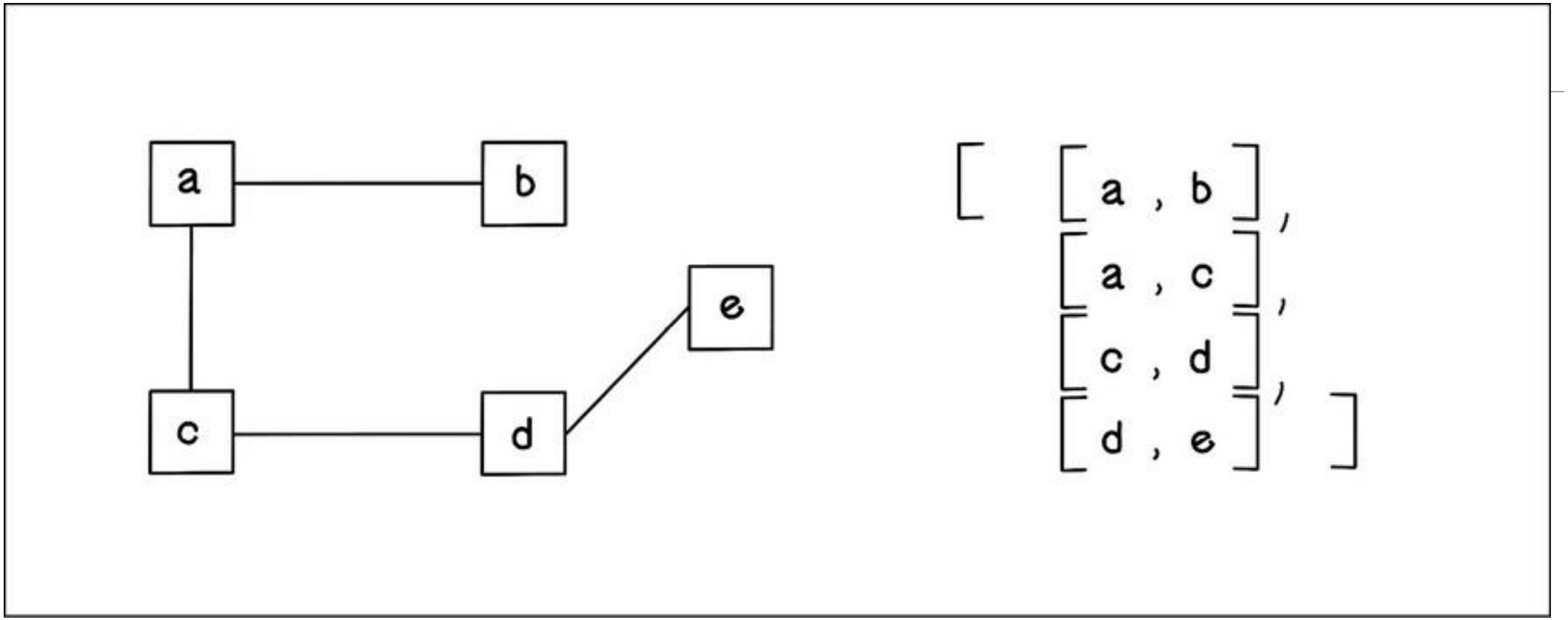

arco (A)
nodo/vertice (N)
a
b
c
d
e
N = {a, b, c, d, e}
A = {ab, ac, ad, bd, be, cd, de}

La Figura mostra un grafo della gita in spiaggia discussa all’inizio di questo capitolo. Ogni sosta è un nodo del grafo; ogni arco fra i nodi rappresenta i punti attraversati; i pesi su ciascun arco indicano la distanza percorsa.


inizio
180
sosta
190
sosta
50
sosta
70
sonno
120
sonno
15
20
10
attività
attività

Rappresentare un grafo come una struttura di dati concreta

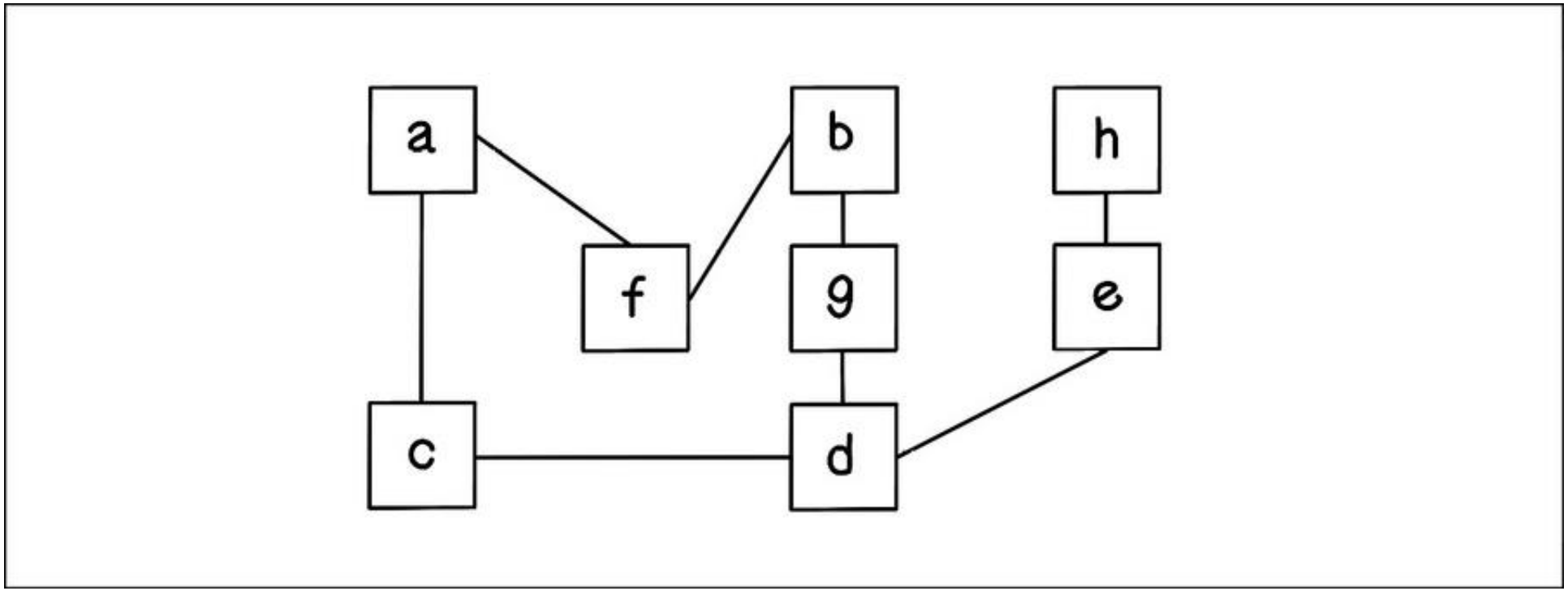
Un grafo può essere rappresentato in diversi modi per rendere possibile un'elaborazione efficiente da parte degli algoritmi. Fondamentalmente, un grafo può essere rappresentato da un array di array, che indica le relazioni fra i nodi. A volte è utile avere un altro array che elenchi semplicemente tutti i nodi del grafo, in modo che i nodi non debbano essere dedotti dalle relazioni.

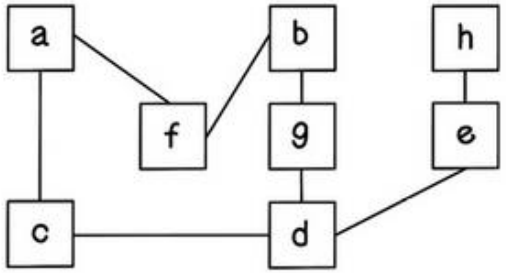

a
b
c
d
e
[[a , b] , [a , c] , [c , d] , [d , e]]

Altre possibili rappresentazioni dei grafi sono la matrice di incidenza, la matrice di adiacenza e la lista di adiacenza. Osservando i nomi di queste rappresentazioni, si vede che l'adiacenza dei nodi di un grafo è importante. Un nodo adiacente è un nodo connesso direttamente a un altro nodo.

Esercizio: rappresentate un grafo come una matrice

Come rappresentereste il seguente grafo usando array di archi?


a
b
h
f
g
e
c
d




```
[ [ a, c ],
  [ a, f ],
  [ b, g ],
  [ b, f ],
  [ c, d ],
  [ d, g ],
  [ d, e ],
  [ e, h ] ]
```

Array di archi

	a	b	c	d	e	f	g	h
a	0	0	1	0	0	1	0	0
b	0	0	0	0	0	1	1	0
c	1	0	0	1	0	0	0	0
d	0	0	1	0	1	0	1	0
e	0	0	0	1	0	0	0	1
f	1	1	0	0	0	0	0	0
g	0	1	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0

Matrice di adiacenza

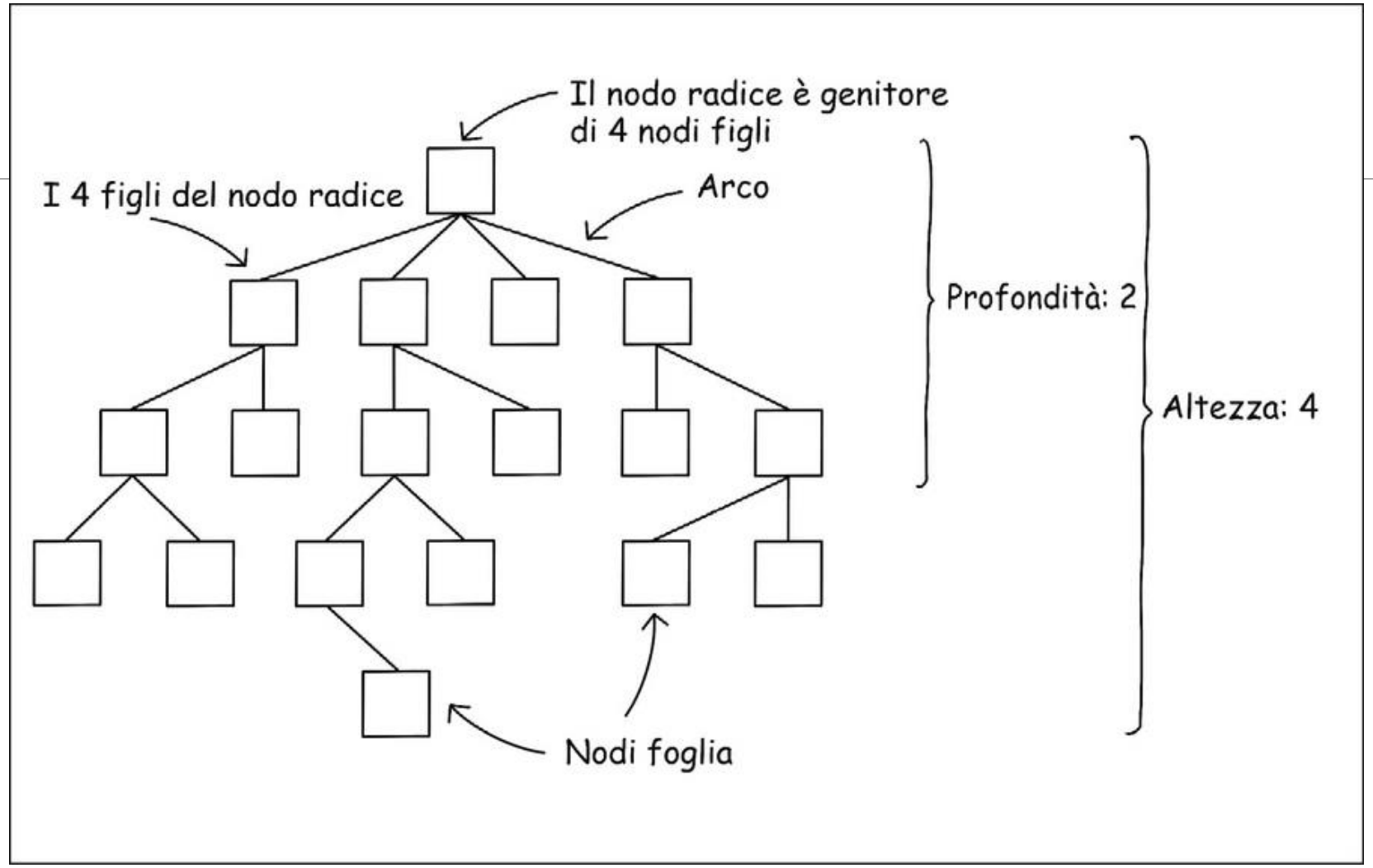
Alberi: le strutture di dati utilizzate per rappresentare le soluzioni di una ricerca

Un albero è una struttura di dati molto utilizzata che simula una gerarchia di valori o oggetti. Una gerarchia è una disposizione di elementi in cui un singolo oggetto è correlato a più oggetti da esso dipendenti. Un albero è un grafo aciclico connesso: ogni nodo ha un arco che punta verso un altro nodo, e non esistono cicli.

In un albero, il valore o l'oggetto rappresentato in un determinato punto è chiamato nodo. Gli alberi, in genere, hanno un singolo nodo radice con zero o più nodi figli che a loro volta possono contenere sottoalberi. Fate un bel respiro ed entriamo nel dettaglio con un po' di terminologia. Quando a un nodo sono connessi altri nodi, il nodo radice è chiamato genitore. Potete applicare questo ragionamento in modo ricorsivo. Un nodo figlio può avere i propri nodi figli, i quali possono a loro volta contenere ulteriori sottoalberi. Ogni nodo figlio ha un singolo nodo genitore. Un nodo senza figli è detto nodo foglia.

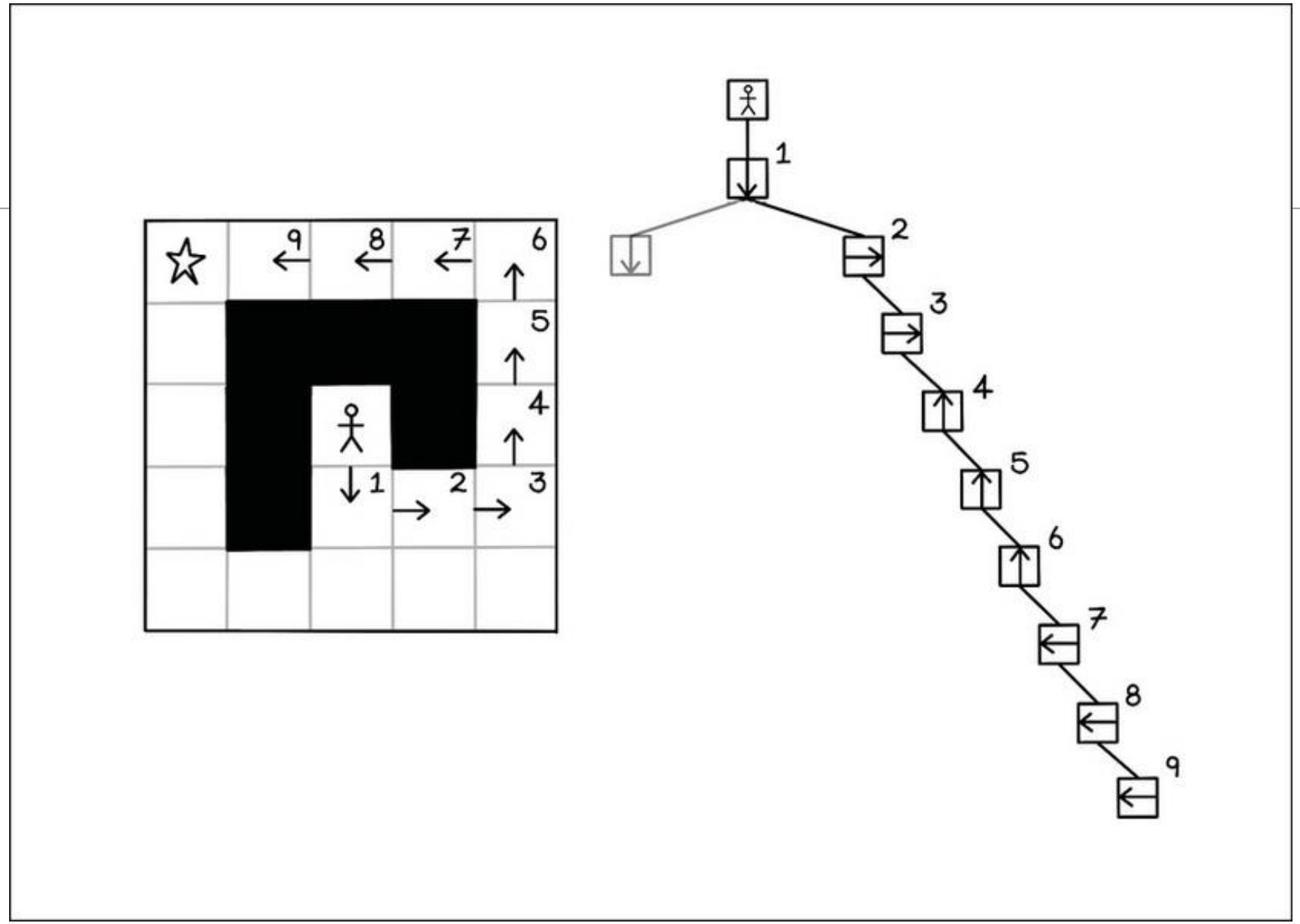
Gli alberi hanno anche un'altezza totale. Nella discesa di livelli di nodi, si parla di profondità

Quando si lavora con gli alberi, spesso si usa una terminologia legata alle relazioni fra i membri di una famiglia. Tenete a mente questa analogia, poiché vi aiuterà a memorizzare i concetti tipici della struttura di dati albero. Notate che nella Figura l'altezza e la profondità sono indicizzate a partire da 0: il nodo radice.


Il nodo radice è genitore
di 4 nodi figli
I 4 figli del nodo radice
Arco
Profondità: 2
Altezza: 4
Nodi foglia

Il nodo più in alto di un albero è chiamato **nodo radice**. Un nodo dal quale dipendono uno o più altri nodi è chiamato nodo padre o nodo genitore, e i nodi connessi che discendono da un nodo genitore sono chiamati nodi figli. I nodi connessi allo stesso nodo genitore sono chiamati nodi fratelli. Una connessione fra due nodi è chiamata arco.

Un percorso è una sequenza di nodi e archi che collegano due nodi non direttamente connessi. Un nodo connesso a un altro nodo seguendo un percorso che parte dal nodo radice è chiamato discendente; un nodo connesso a un altro nodo seguendo un percorso che si dirige verso il nodo radice è chiamato antenato. Un nodo senza figli è chiamato nodo foglia. Per descrivere il numero di figli di un nodo si usa il termine grado; pertanto, un nodo foglia ha grado zero.


9
8
7
6
5
4
1
2
3
1
2
3
4
5
6
7
8
9

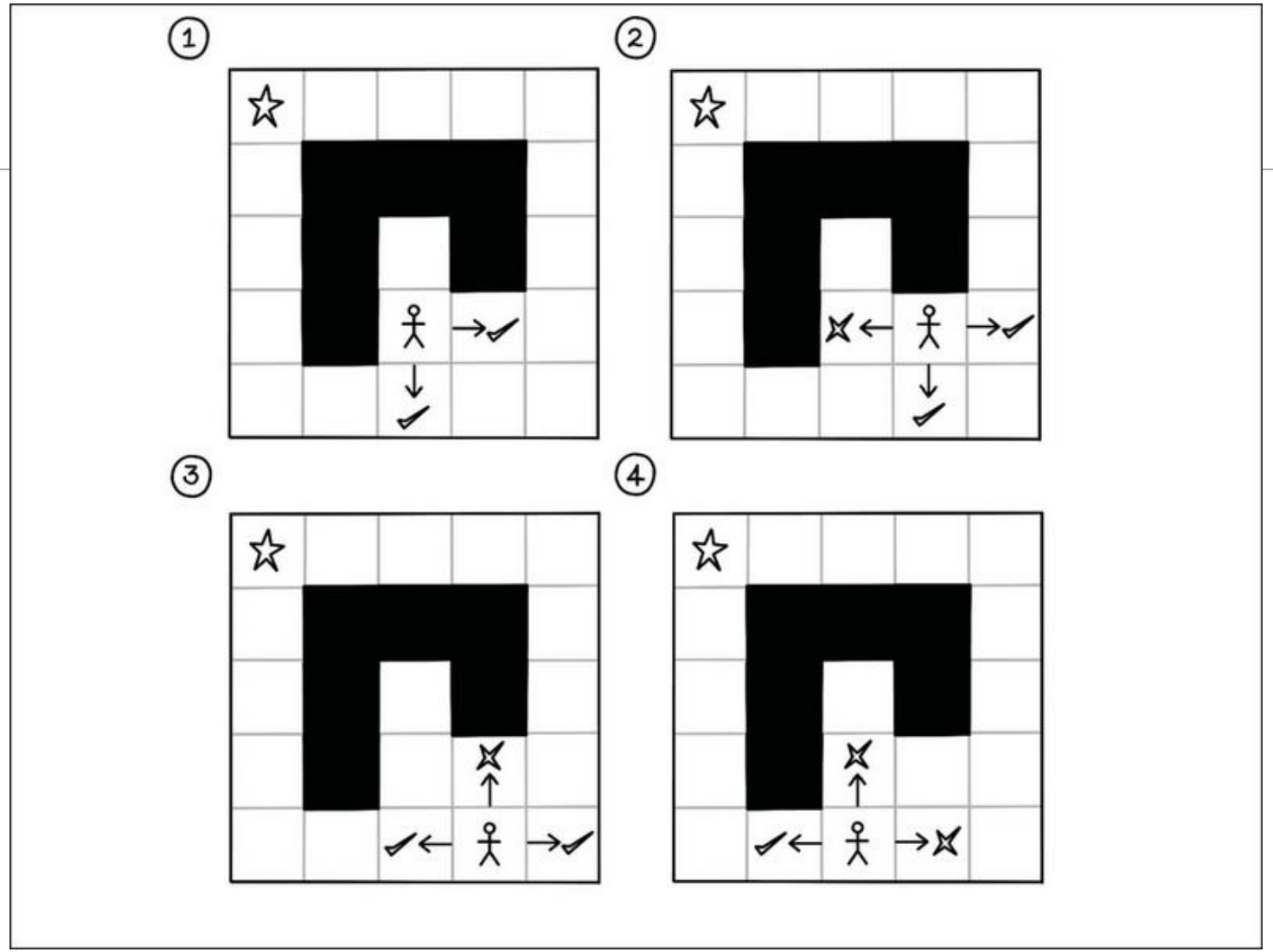
Gli alberi sono la struttura di dati fondamentale per gli algoritmi di ricerca, che approfondiremo in seguito. Gli algoritmi di ordinamento sono utili anche per risolvere in modo più efficiente determinati problemi di calcolo.

Ricerca non informata: ricerca cieca delle soluzioni

La ricerca non informata è detta anche ricerca non guidata, ricerca cieca o ricerca a forza bruta. Gli algoritmi di ricerca non informati non dispongono di informazioni sul dominio del problema, a parte la rappresentazione del problema, che di solito è un albero.

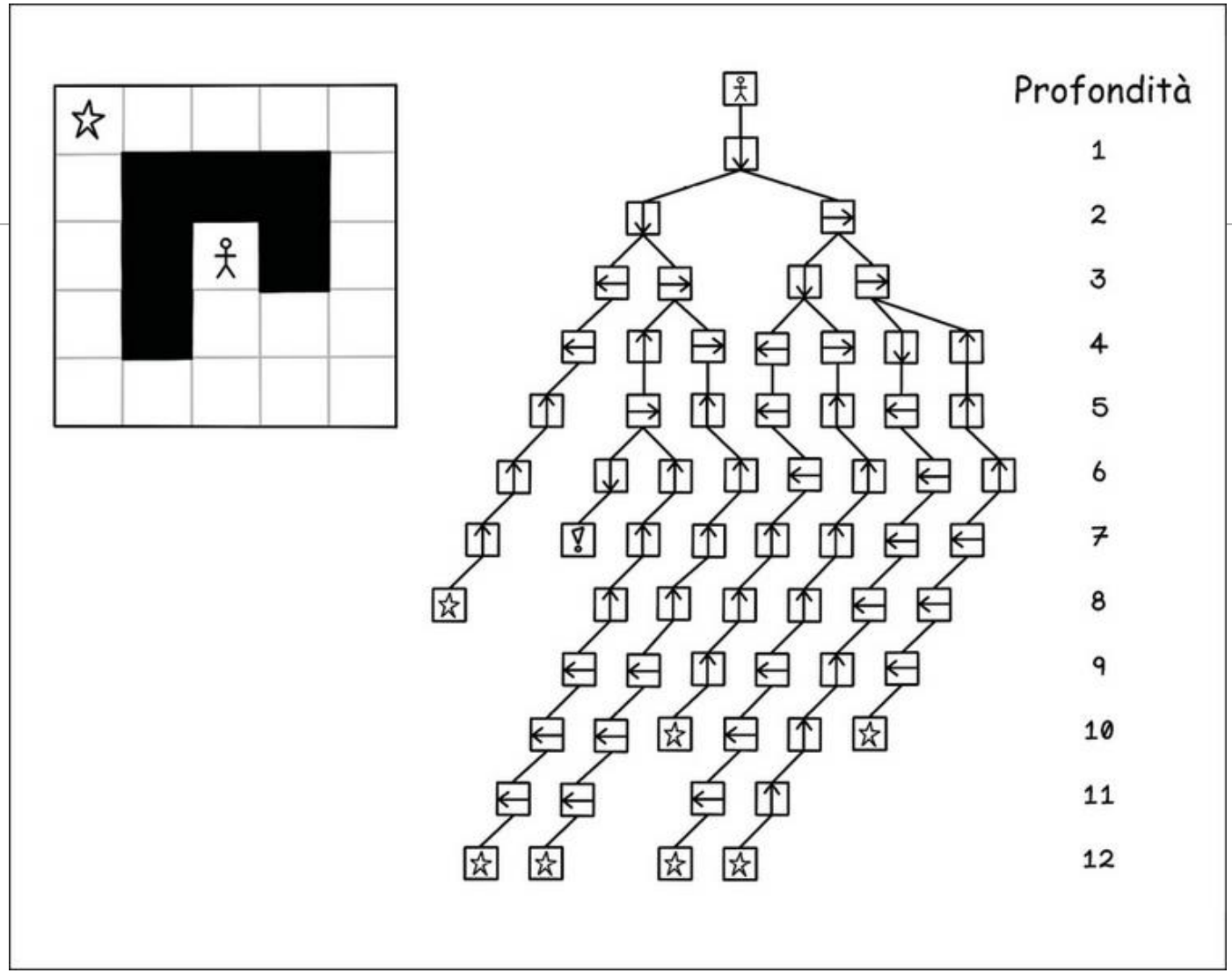
Pensate a come si esplorano le cose da imparare. Alcuni potrebbero esaminare in ampiezza tutti gli argomenti e apprendere le basi di ciascuno di essi; altri potrebbero scegliere un determinato argomento ed esplorare tutti i suoi sottoargomenti, in profondità. Si parla quindi di ricerca in ampiezza e ricerca in profondità. Una ricerca in profondità (Depth-First Search, DFS) esplora un determinato percorso dall'inizio finché non trova un obiettivo alla massima profondità dell'albero. Una ricerca in ampiezza (Breadth-First Search, BFS) esplora prima tutte le opzioni a un livello di profondità, prima di passare a un livello più in profondità nell'albero.

Considerate lo scenario del labirinto. Nel tentativo di trovare un percorso ottimale verso l'obiettivo, applicate il seguente semplice vincolo per evitare di rimanere bloccati in un ciclo infinito, ovvero per impedire la formazione di cicli nel nostro albero: il giocatore non può portarsi in un blocco che ha precedentemente occupato. Poiché gli algoritmi non informati tentano ogni possibile opzione di ogni nodo, la creazione di un ciclo provocherebbe il fallimento catastrofico dell'algoritmo.


1
2
3
4

Questo vincolo impedisce la formazione di cicli nel percorso verso l'obiettivo. Ma questo vincolo introdurrà problemi se, in un labirinto diverso con vincoli o regole differenti, sarà necessario portarsi più di una volta su un blocco per trovare la soluzione

sono rappresentati tutti i possibili percorsi nell'albero, per evidenziare le diverse opzioni disponibili. Questo albero contiene sette percorsi che portano all'obiettivo e un percorso che produce una soluzione non valida, dato il vincolo di non portarsi su blocchi precedentemente occupati. È importante capire che in questo piccolo labirinto, non è difficile rappresentare tutte le possibilità. Ma lo scopo degli algoritmi di ricerca è quello di cercare o generare questi alberi in modo iterativo, nei casi in cui generare l'intero albero delle possibilità si inefficiente a causa del puro costo computazionale.

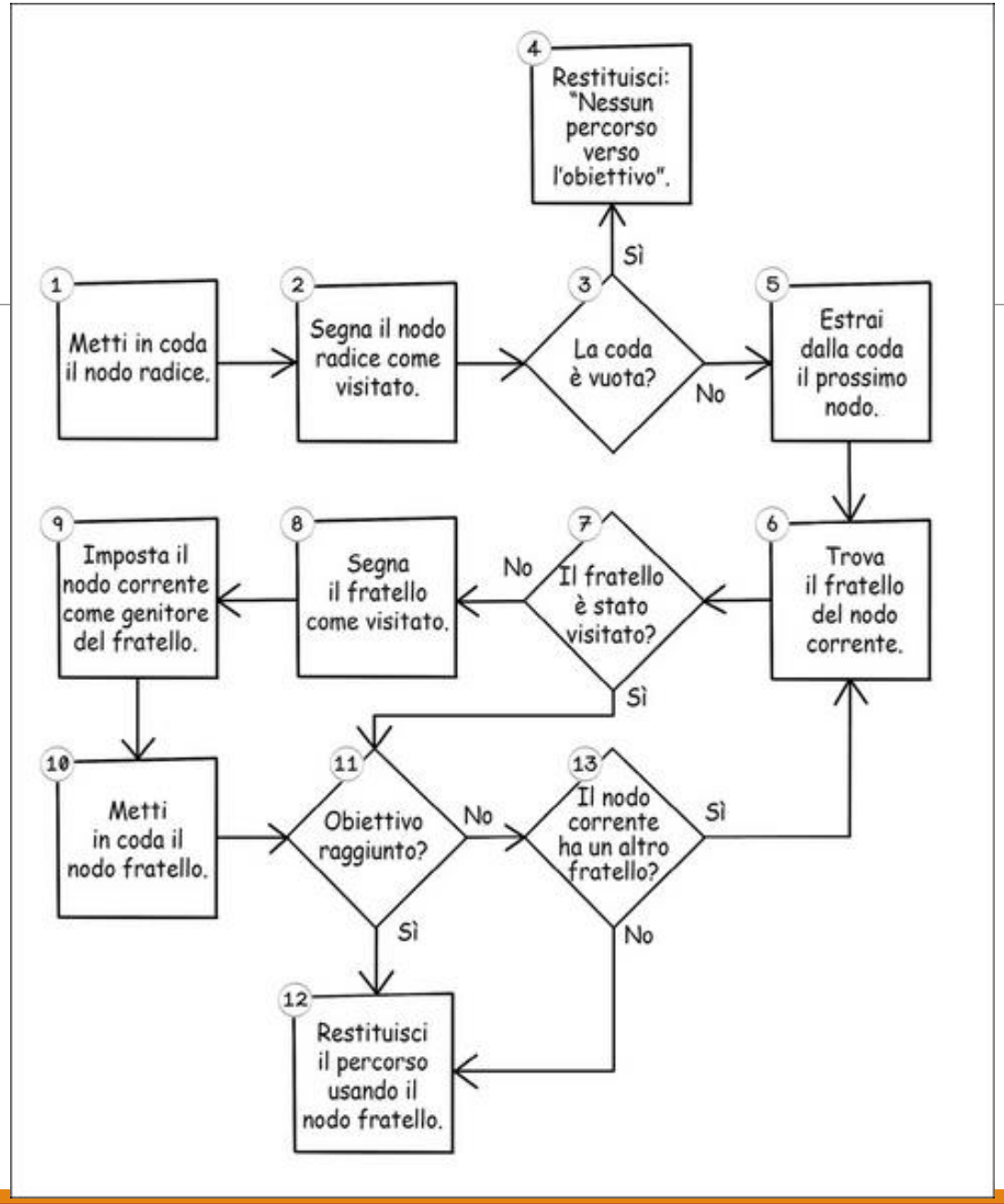

Profondità
1
2
3
4
5
6
7
8
9
10
11
12

Ricerca in ampiezza: prima in ampiezza e poi in profondità

La ricerca in ampiezza è un algoritmo utilizzato per attraversare o generare un albero. Questo algoritmo parte da un nodo, la radice o root, ed esplora ogni singolo nodo a quel livello di profondità, prima di passare ai nodi del livello successivo, e questo finché non trova un nodo foglia obiettivo.

L'algoritmo di ricerca in ampiezza viene implementato al meglio utilizzando una coda first-in, first-out in cui vengono elaborati i nodi a una certa profondità e i loro figli vengono messi in coda per essere elaborati successivamente. Questo ordine di elaborazione è esattamente ciò di cui abbiamo bisogno quando implementiamo questo algoritmo.

La Figura è un diagramma di flusso che descrive la sequenza di passaggi coinvolti nell'algoritmo di ricerca in ampiezza.


1
Metti in coda il nodo radice.
2
Segna il nodo radice come visitato.
3
La coda è vuota?
Sì
No
4
Restituisci: "Nessun percorso verso l'obiettivo".
5
Estrai dalla coda il prossimo nodo.
6
Trova il fratello del nodo corrente.
7
Il fratello è stato visitato?
No
Sì
8
Segna il fratello come visitato.
9
Imposta il nodo corrente come genitore del fratello.
10
Metti in coda il nodo fratello.
11
Obiettivo raggiunto?
No
Sì
12
Restituisci il percorso usando il nodo fratello.
13
Il nodo corrente ha un altro fratello?
Sì
No

Di seguito sono riportate alcune note e osservazioni aggiuntive per ciascuna fase del processo.

1.Metti in coda il nodo radice. L'algoritmo di ricerca in ampiezza viene implementato al meglio con una coda. Gli oggetti vengono elaborati nella sequenza in cui vengono aggiunti alla coda. Questo processo è noto anche come coda FIFO (First In, First Out). Il primo passaggio consiste nell'aggiungere il nodo radice alla coda. Questo nodo rappresenta la posizione di partenza del giocatore sulla mappa.

2.Segna il nodo radice come visitato. Ora che il nodo radice è stato aggiunto alla coda per l'elaborazione, viene contrassegnato come visitato, per evitare che venga visitato di nuovo senza motivo.

3.La coda è vuota? Se la coda è vuota (tutti i nodi sono stati elaborati dopo le iterazioni) e non è stato restituito alcun percorso nel Passaggio 12 dell'algoritmo, non esiste alcun percorso verso l'obiettivo. Se ci sono ancora nodi nella coda, l'algoritmo può continuare la sua ricerca per trovare l'obiettivo.

4.Restituisci: "Nessun percorso verso l'obiettivo". Questo messaggio è l'unica possibile via d'uscita dall'algoritmo se non esiste alcun percorso verso l'obiettivo.

5.Estrai dalla coda il prossimo nodo. Estraendo l'oggetto successivo dalla coda e impostandolo come nodo corrente, possiamo esplorarne le caratteristiche. All'avvio dell'algoritmo, il nodo corrente sarà il nodo radice.

6.Trova il fratello del nodo corrente. Questo passaggio ottiene la prossima mossa possibile dalla posizione corrente, facendo riferimento al labirinto e determinando se è possibile un movimento a nord, sud, est o ovest.

7.Il fratello è stato visitato? Se il fratello corrente non è ancora stato visitato, non è stato ancora esplorato e può essere elaborato ora.

8.Segna il fratello come visitato. Questo passaggio indica che questo nodo fratello è stato visitato.

9.Imposta il nodo corrente come genitore del fratello. Imposta il nodo di origine come genitore del fratello. 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.

10.Metti in coda il nodo fratello. Il nodo adiacente viene messo in coda affinché possano essere esplorati i suoi figli. Questo meccanismo consente ai nodi di ogni singola profondità di essere elaborati in ordine.

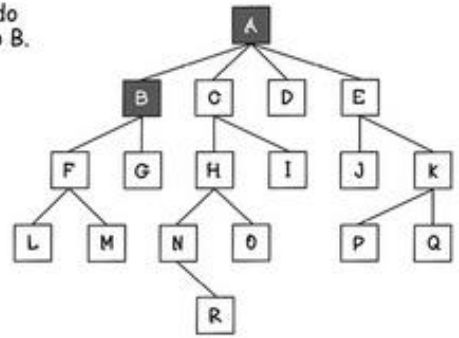
11.Obiettivo raggiunto? Questo passaggio determina se il fratello corrente contiene l'obiettivo che l'algoritmo sta cercando.

12.Restituisci il percorso usando il nodo fratello. Facendo riferimento al genitore del nodo fratello, quindi al genitore di quel nodo e così via, si traccia il percorso dall'obiettivo alla radice. Il nodo radice sarà un nodo senza genitore.

13.Il nodo corrente ha un altro fratello? Se il nodo corrente ha altre mosse possibili nel labirinto, andate al Passaggio 6 per fare una mossa.

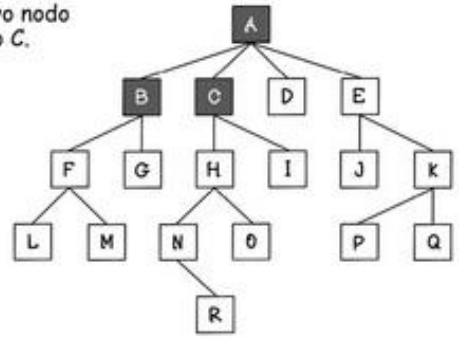
Esaminiamo come funzionerebbe la cosa in un semplice albero. Notate che, dato il modo in cui l’albero viene esplorato e i nodi vengono aggiunti alla coda FIFO, i nodi vengono elaborati nell’ordine desiderato sfruttando la coda.

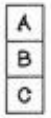
Visita il primo nodo figlio di A, ovvero B.



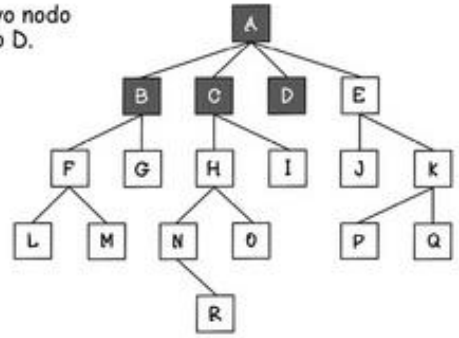
Sequenza di elaborazione della coda.

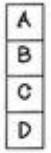
Visita il successivo nodo figlio di A, ovvero C.



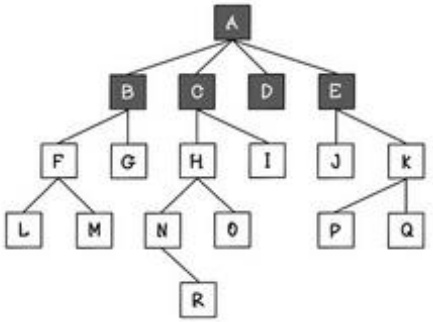


Visita il successivo nodo figlio di A, ovvero D.

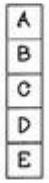




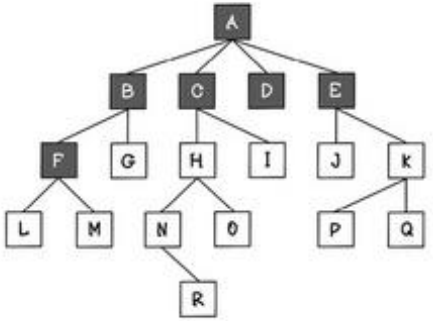
Visita l'ultimo nodo figlio di A, ovvero E.

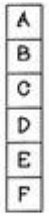


Sequenza di elaborazione della coda.

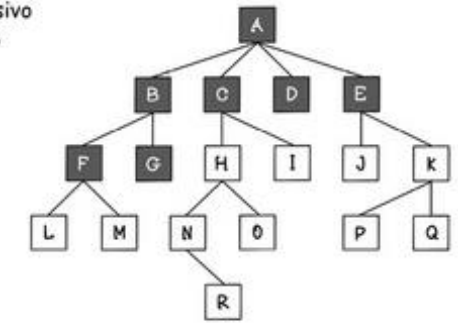


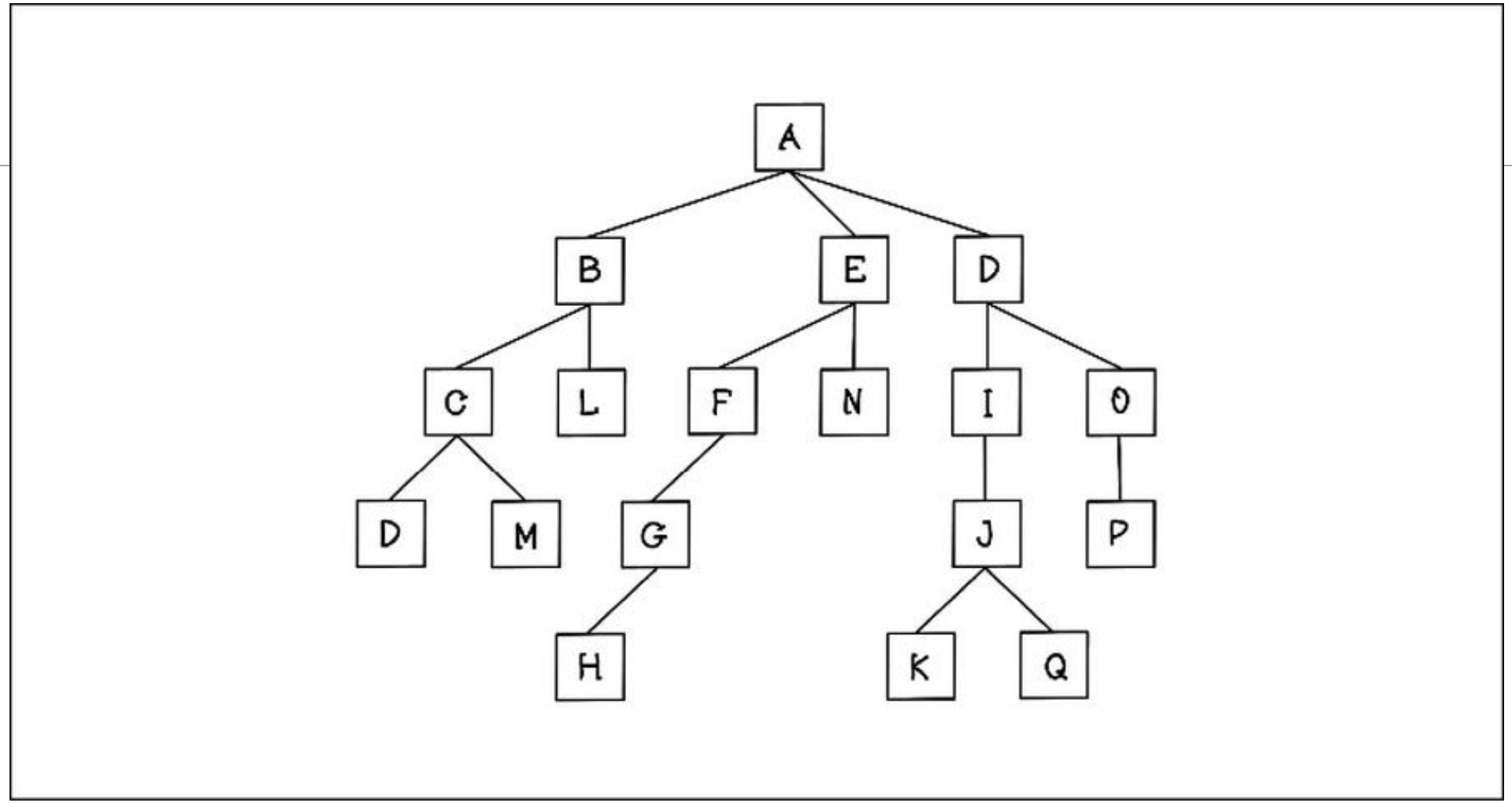
Visita il primo figlio del primo figlio di A.
Si tratta del primo figlio di B, ovvero F.

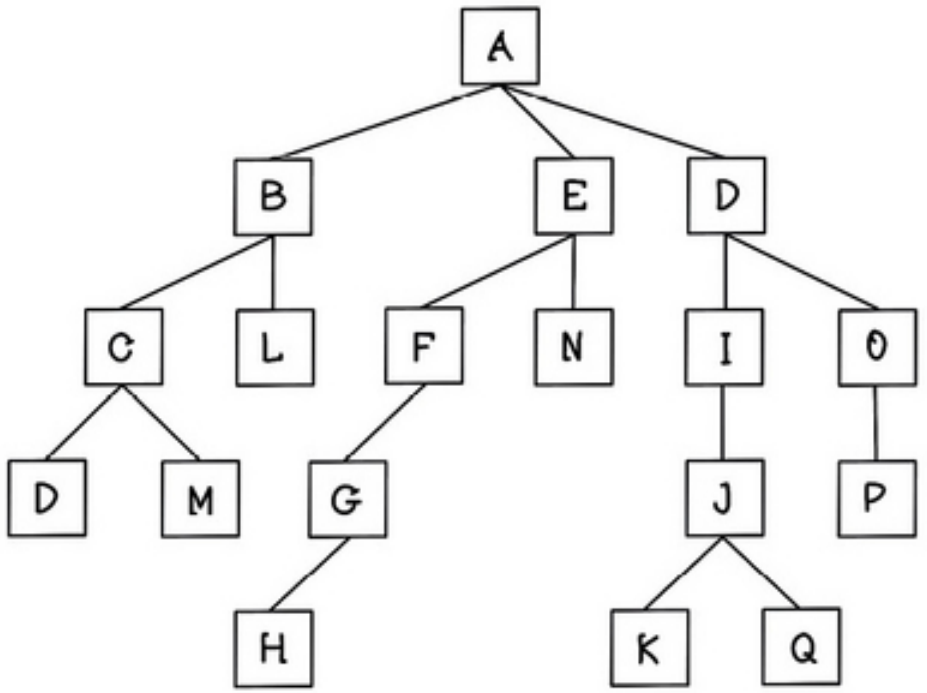




Visita il successivo nodo figlio di B, ovvero G.




A
B
E
D
C
L
F
N
I
0
D
M
G
J
P
H
K
Q



Ordine di ricerca dell'algoritmo di ricerca in ampiezza
A, B, E, D, C, L, F, N, I, O, D, M, G, J, P, H, K, Q

Nell’esempio del labirinto, l’algoritmo parte dalla posizione attuale del giocatore nel labirinto, valuta tutte le possibili scelte delle mosse e ripete quella logica per ogni movimento effettuato, fino al raggiungimento dell’obiettivo. In questo modo, l’algoritmo genera un albero con un unico percorso verso l’obiettivo.

È importante capire che per generare il percorso vengono utilizzati i processi di visita dei nodi dell’albero. Stiamo trovando i nodi correlati utilizzando un meccanismo automatico.

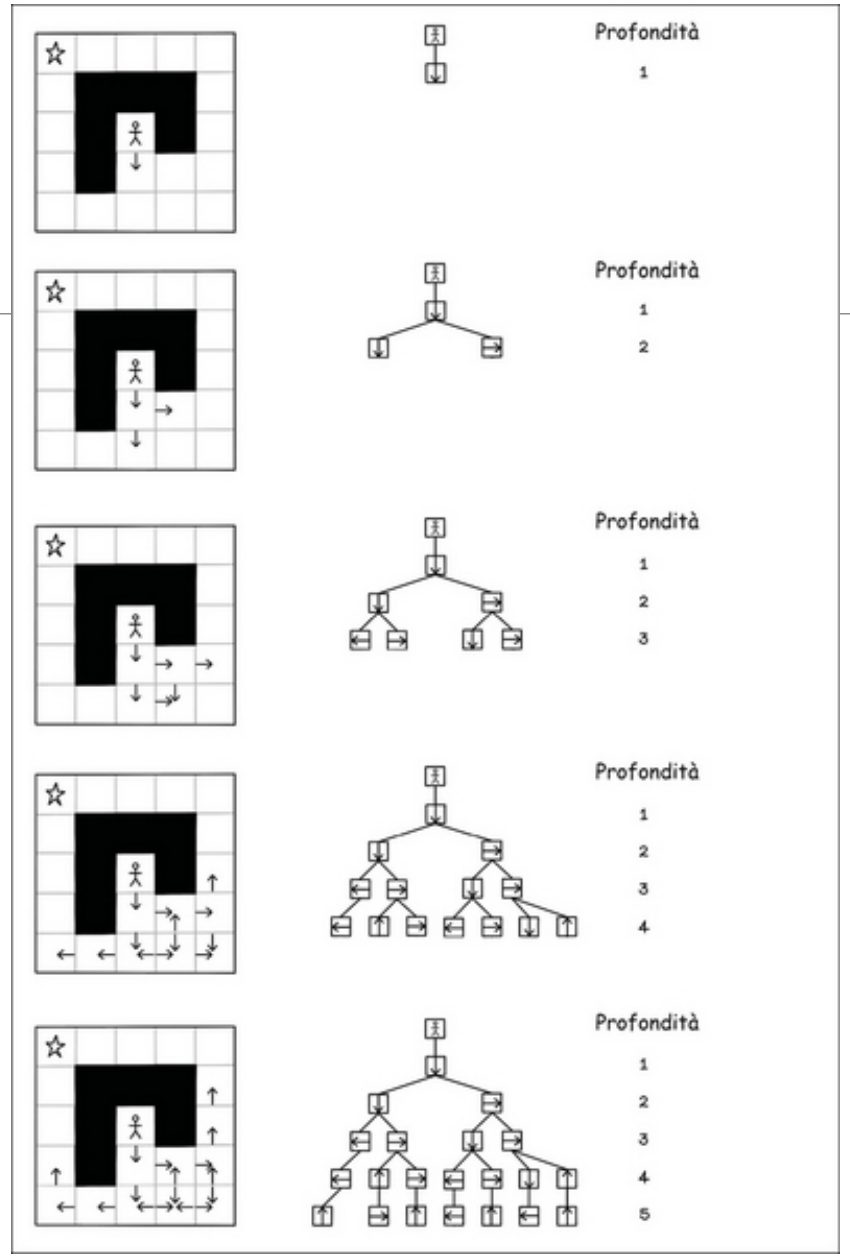
Ogni percorso verso l'obiettivo consiste in una serie di mosse per raggiungere l'obiettivo. Il numero di mosse del percorso è la distanza per raggiungere l'obiettivo per quel percorso, un valore che chiameremo costo. Il numero di spostamenti è pari anche al numero di nodi visitati nel percorso, dal nodo radice al nodo foglia che rappresenta l'obiettivo. L'algoritmo si sposta lungo l'albero scendendo ogni volta a un nuovo livello di profondità, finché non trova un obiettivo; poi restituisce come soluzione il primo percorso che l'ha portato a tale obiettivo. Potrebbe esserci un percorso migliore verso l'obiettivo, ma poiché la ricerca in ampiezza procede alla cieca, non è garantito che lo trovi.

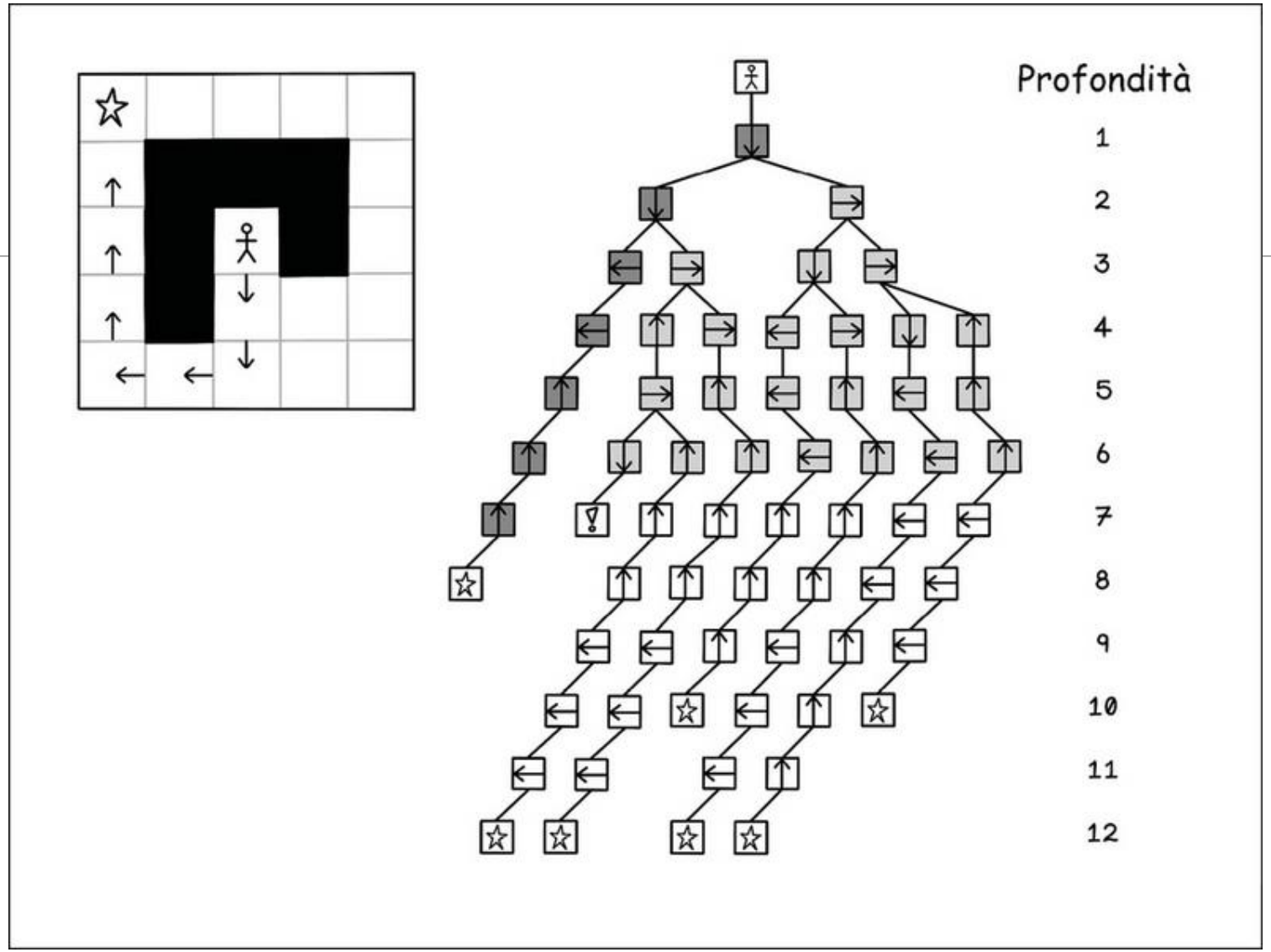
La Figura 2.19 mostra la generazione di un albero utilizzando i movimenti nel labirinto. Poiché l’esplorazione segue una ricerca in ampiezza, prima vengono esplorati i nodi a un dato livello di profondità e solo dopo viene iniziata l’esplorazione della profondità successiva.

Pseudocodice

Come ho accennato, l’algoritmo di ricerca in ampiezza utilizza una coda per generare un albero, un livello di profondità alla volta. Avere una struttura per archiviare i nodi visitati è fondamentale per evitare di rimanere bloccati in circuiti ciclici; e l’impostazione del genitore di ciascun nodo è importante per determinare un percorso dal punto di partenza nel labirinto all’obiettivo:

```
run_bfs(maze, current_point, visited_points):
 let q equal a new queue
  push current_point to q
  mark current_point as visited
  while q is not empty:
      pop q and let current_point equal the returned point
      add available cells north, east, south, and west to a list neighbors
      for each neighbor in neighbors:
          if neighbor is not visited:
              set neighbor parent as current_point
              mark neighbor as visited
              push neighbor to q
              if value at neighbor is the goal:
                  return path using neighbor
  return "No path to goal"
```


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


Profondità
1
2
3
4
5
6
7
8
9
10
11
12

Ricerca in profondità: prima in profondità e poi in ampiezza

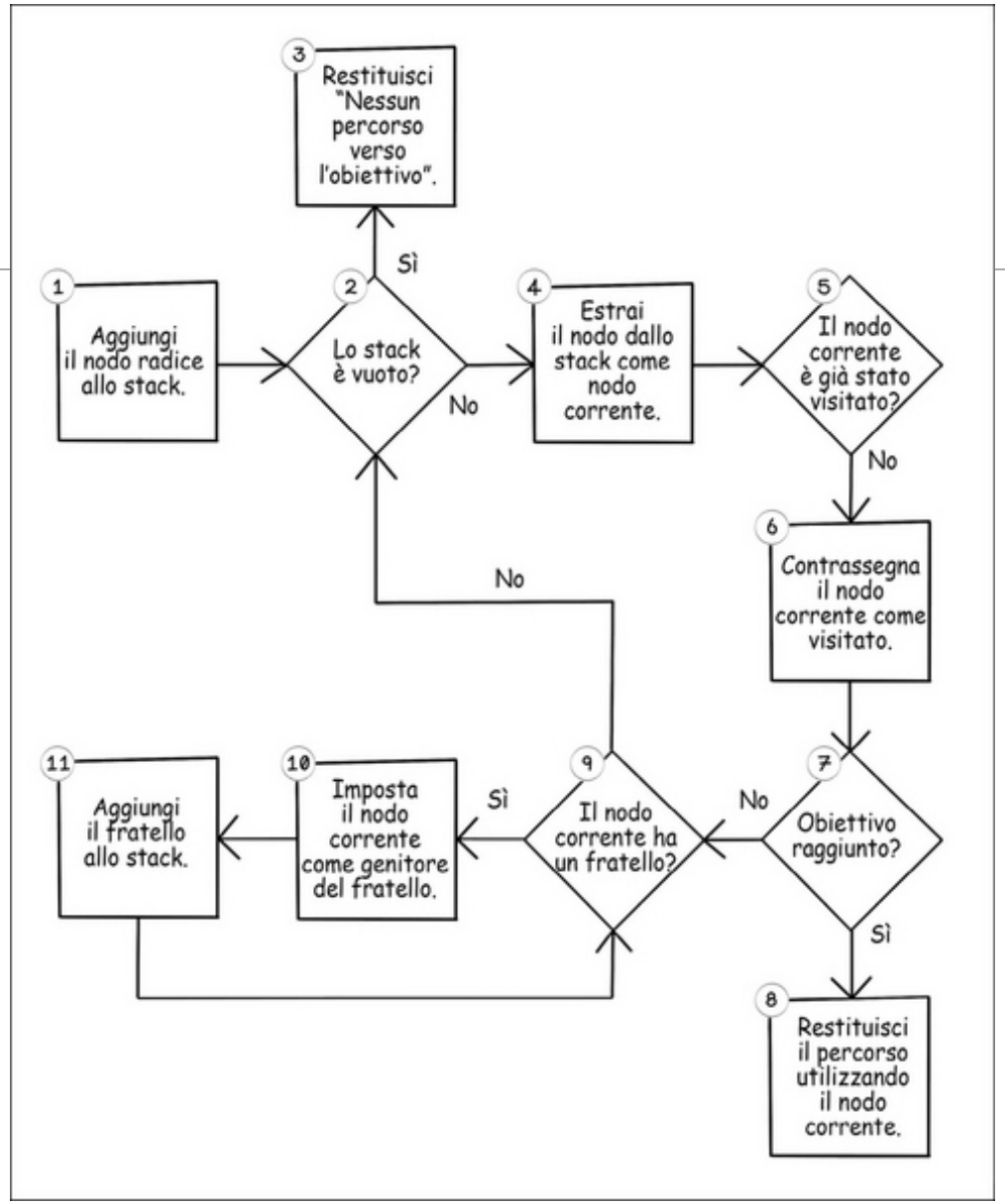
La ricerca in profondità è un altro algoritmo utilizzato per attraversare un albero o per generare nodi e percorsi in un albero. Questo algoritmo parte da un determinato nodo ed esplora in profondità i percorsi relativi ai nodi connessi al primo figlio, procedendo in modo ricorsivo fino a raggiungere il nodo foglia più lontano, per poi tornare indietro ed esplorare altri percorsi verso nodi foglia ma da altri nodi figli visitati. La Figura illustra il flusso generale dell'algoritmo di ricerca in profondità.

Esaminiamo il flusso dell'algoritmo di ricerca in profondità.

• Aggiungi il nodo radice allo stack. L'algoritmo di ricerca in profondità può essere implementato utilizzando uno stack, in cui viene elaborato per primo l'ultimo oggetto inserito. Questo processo è noto come coda LIFO (Last In, First Out). 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 invece ci sono ancora nodi nello stack, l'algoritmo può continuare la sua ricerca per trovare l'obiettivo.

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


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?
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
10
Imposta il nodo corrente come genitore del fratello.
11
Aggiungi il fratello allo stack.
No

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

• 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 che venga elaborato una seconda volta.

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

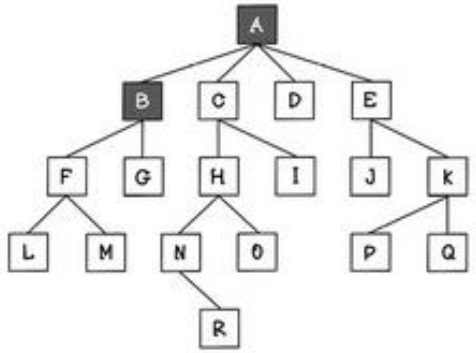
• Restituisci il percorso utilizzando il nodo corrente. Facendo riferimento al genitore del nodo corrente, quindi al genitore di quel nodo e così via, viene creato il percorso dall'obiettivo alla radice. Il nodo radice sarà un nodo senza genitore.

• Il nodo corrente ha un fratello? Se il nodo corrente ha altre mosse possibili, tale mossa può essere aggiunta allo stack per essere elaborata. In caso contrario, l'algoritmo può tornare al Passaggio 2, in cui può essere elaborato l'oggetto successivo dello stack, se non è vuoto. La natura dello stack LIFO consente all'algoritmo di elaborare tutti i nodi fino a raggiungere la profondità di un nodo foglia, prima di tornare indietro per visitare altri figli del nodo radice.

• 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.

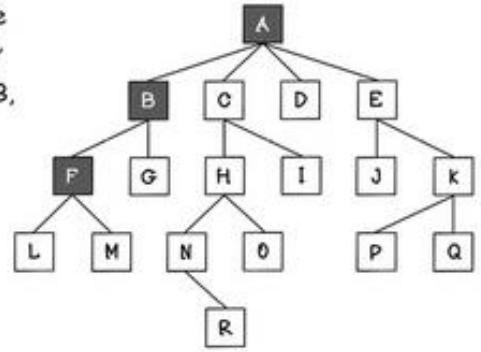
• Aggiungi il fratello allo stack. Il nodo adiacente viene aggiunto allo stack affinché possano essere esplorati i suoi figli. Ancora una volta, questo meccanismo a stack consente di elaborare i nodi alla massima profondità dell'albero prima di elaborare i fratelli a minori profondità.

Analogamente alla ricerca in ampiezza, si visita il primo figlio del nodo A, ovvero B.

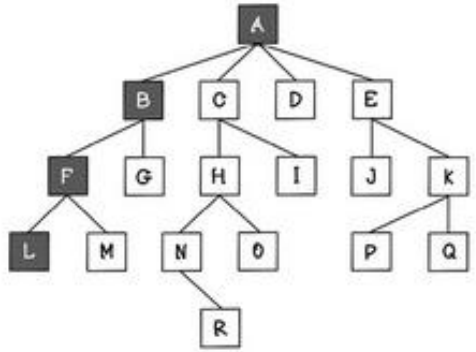


Sequenza di elaborazione dello stack.

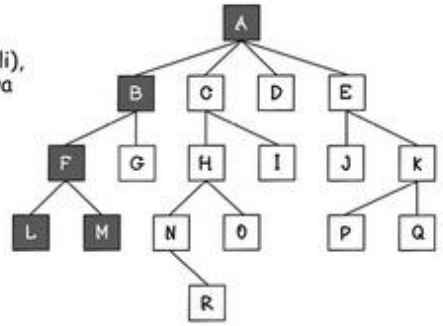
Invece di visitare gli altri figli di A, si procede con il primo figlio di B, ovvero F.



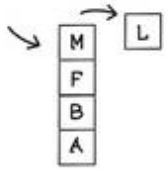
Di nuovo, viene visitato il primo figlio di F, ovvero L.



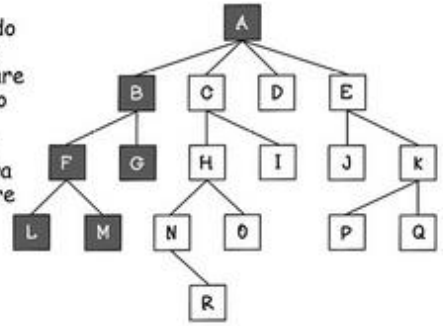
Poiché L è un nodo foglia (ovvero senza figli), l'algoritmo arretra per visitare il successivo figlio di F, ovvero M.

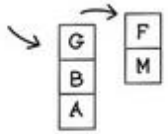


Sequenza di elaborazione dello stack.

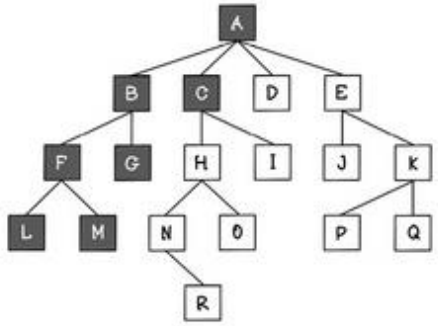


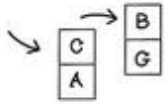
Poiché M è un nodo foglia, l'algoritmo arretra per visitare il successivo figlio di F. Ma i figli sonofiniti, quindi l'algoritmo arretra ancora per visitare il successivo figlio di B, ovvero G.





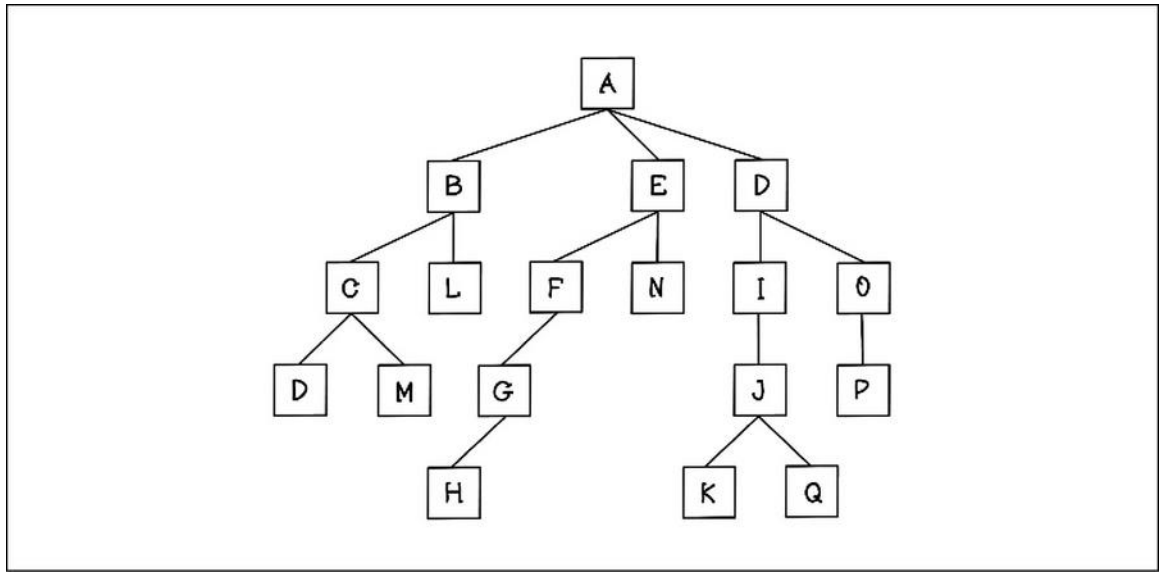
Infine, poiché i figli di B sono finiti, l'algoritmo arretra ancora per visitare il successivo figlio di A, ovvero C.

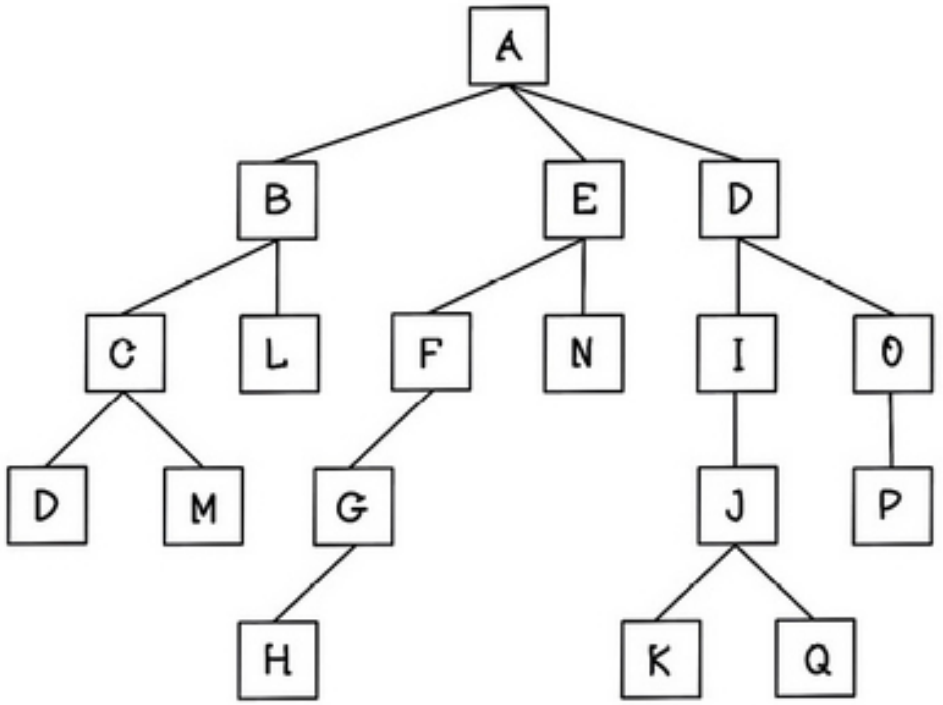




Esercizio: determinare il percorso verso la soluzione

Quale sarebbe l'ordine delle visite nella ricerca in profondità del seguente albero?

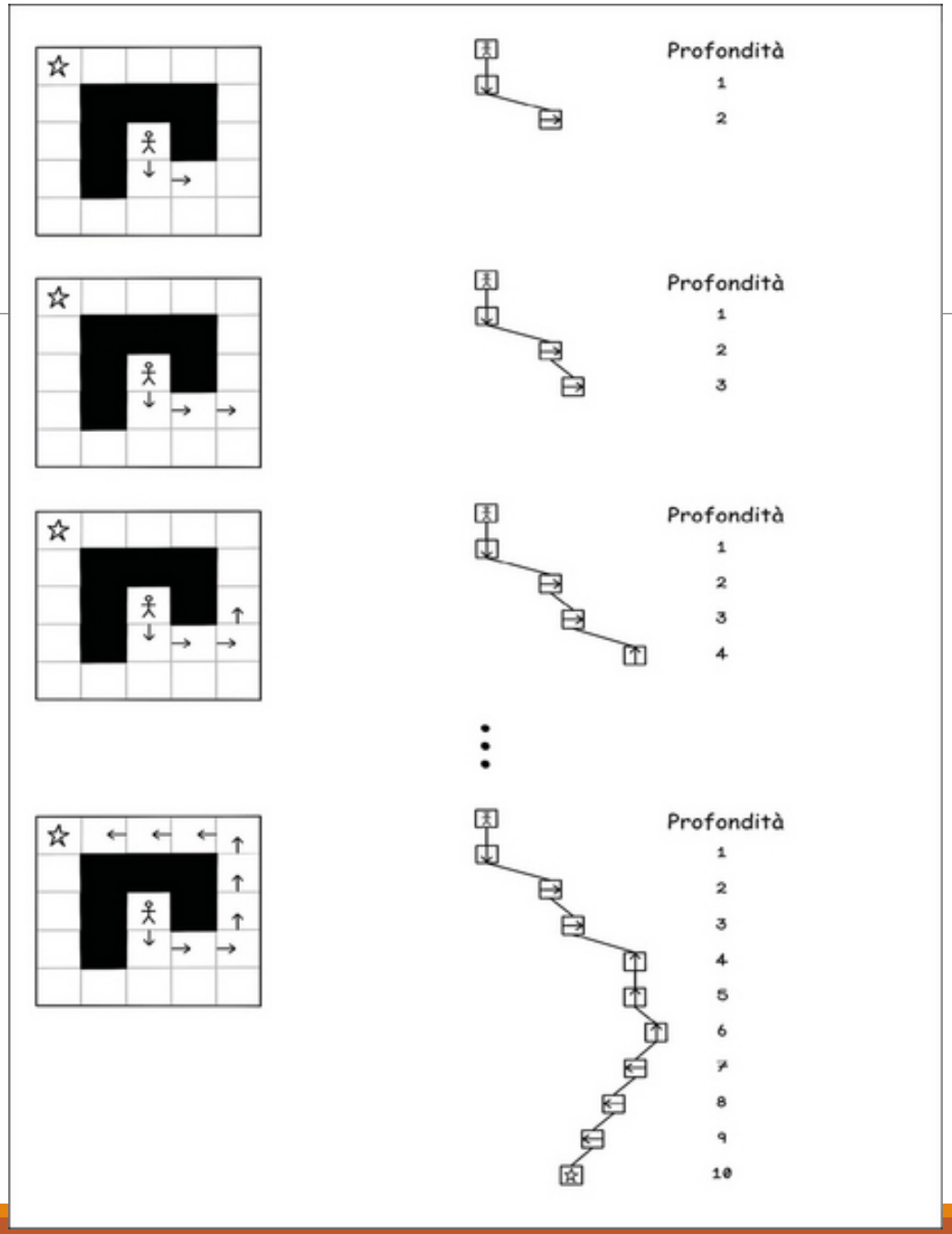


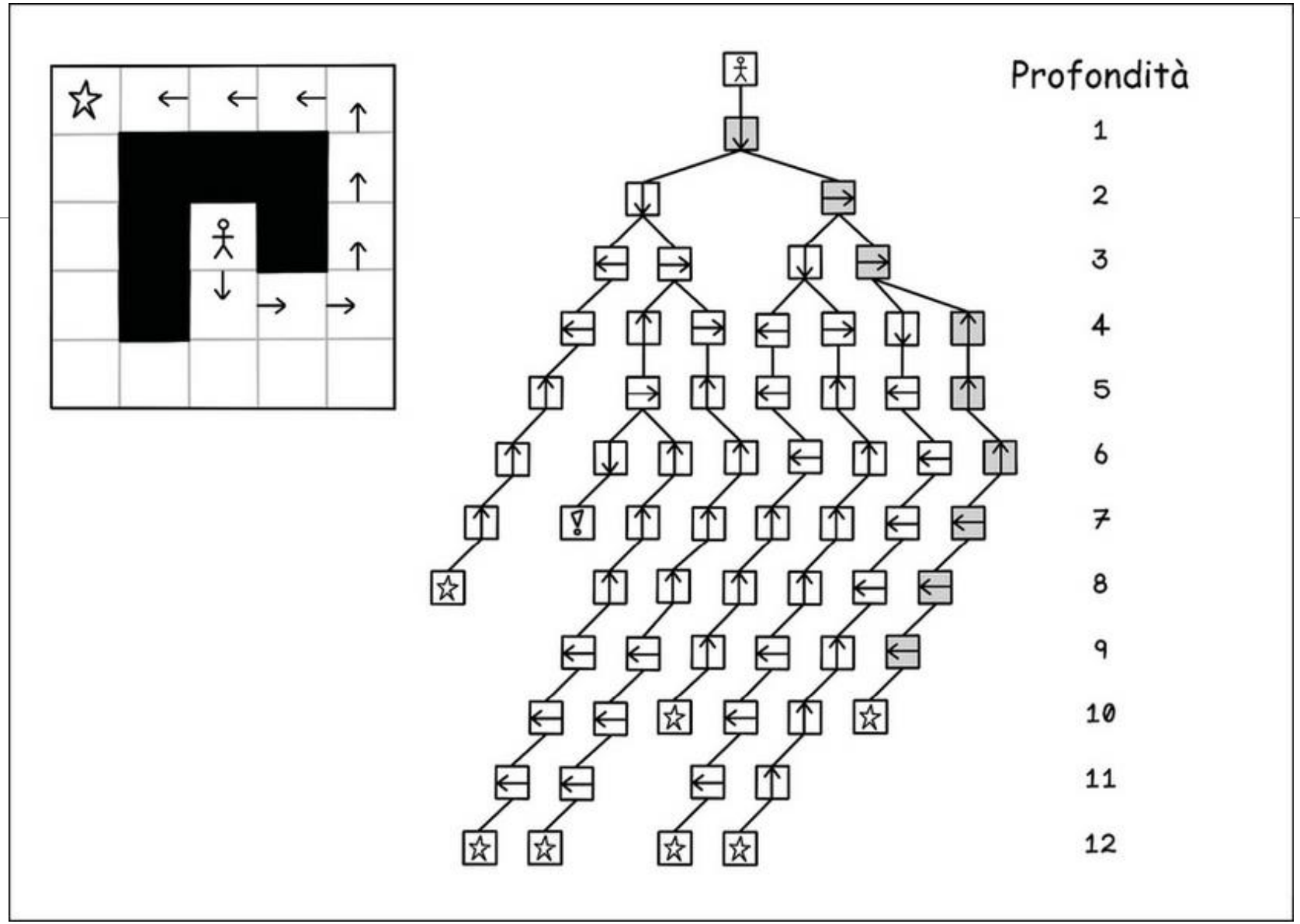


Ordine di ricerca dell'algoritmo di ricerca in profondità
A, B, C, D, M, L, E, F, G, H, N, D, I, J, K, Q, O, P

```
run_dfs(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
              push neighbor to s
  return "No path to goal"
```


Profondità
1
2
Profondità
1
2
3
Profondità
1
2
3
4
Profondità
1
2
3
4
5
6
7
8
9
10


Profondità
1
2
3
4
5
6
7
8
9
10
11
12

Casi d’uso per gli algoritmi di ricerca non informati

Gli algoritmi di ricerca non informati sono versatili e utili in diversi casi d’uso del mondo reale. Ecco alcuni esempi.

• Ricerca di percorsi fra i nodi di una rete: quando due computer devono comunicare su una rete, la connessione attraversa molti computer e dispositivi. Gli algoritmi di ricerca possono essere utilizzati per trovare in quella rete un percorso fra i due dispositivi.

• Scansione di pagine web: le ricerche web ci consentono di trovare informazioni in uno sterminato insieme di pagine web. Per indicizzare queste pagine web, i crawler in genere leggono le informazioni presenti in ciascuna pagina, oltre a seguire in modo ricorsivo ogni link presente in quella pagina. Gli algoritmi di ricerca sono utili per creare crawler, strutture di metadati e relazioni fra i contenuti.

• Trovare connessioni nei social network: le applicazioni social ospitano molte persone e le loro relazioni. Bob può essere amico di Alice, per esempio, la quale è amica di John; quindi Bob e John sono connessi solo indirettamente tramite Alice. Un’applicazione di social media può suggerire che Bob e John diventino amici, perché potrebbero conoscersi grazie alla reciproca amicizia con Alice.

Facoltativo: ulteriori informazioni sulle categorie di grafi

I grafi sono utili per molti problemi informatici e matematici e, a causa della natura dei diversi tipi di grafi, a determinate categorie di grafi possono essere applicati principi e algoritmi differenti. Un grafo può essere classificato in base alla sua struttura, al numero di nodi, al numero di archi e all'interconnettività fra i nodi.

È utile conoscere queste categorie di grafi, in quanto spesso sono citate nelle ricerche e in altri algoritmi di intelligenza artificiale.

• Grafo non orientato: il grafo non ha archi orientati. Le relazioni fra due nodi sono reciproche. Come le strade fra città, prevedono entrambe le direzioni di movimento.

• Grafo orientato: gli archi indicano la direzione. Le relazioni fra due nodi sono esplicite. Come in un grafo che rappresenta un figlio e un genitore, il nodo figlio non può essere "genitore del suo genitore".

• Grafo disconnesso: uno o più nodi non sono collegati da alcun arco. Come in un grafo che rappresenti i punti di contatto fra continenti, alcuni nodi non sono collegati. Come nel caso dei continenti, alcuni sono collegati dalla terra e altri sono separati dagli oceani.

• Grafo aciclico: un grafo che non contiene cicli. Come nel caso del tempo, il grafo non può tornare indietro a nessun punto del passato.

• Grafo completo: ogni nodo è connesso a ogni altro nodo da un arco. Come nelle linee di comunicazione di un piccolo team, tutti parlano con tutti gli altri.

• Grafo bipartito completo: una partizione di archi è un raggruppamento di archi. Data una partizione di archi, ogni nodo di una partizione è connesso a ogni nodo dell'altra partizione tramite archi. Come in una degustazione di formaggi, tipicamente, ogni singola persona assaggia ogni singolo tipo di formaggio.

• Grafo pesato: un grafo in cui gli archi fra i nodi hanno un peso. Come nella distanza fra le città, alcune città sono più distanti fra loro di altre. Le loro connessioni, quindi, "pesano" di più.

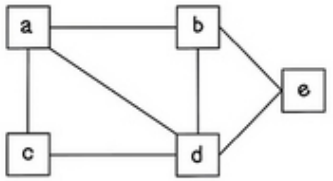
È utile comprendere i diversi tipi di grafi per descrivere al meglio il problema in questione e utilizzare l'algoritmo più efficiente per l'elaborazione.

Facoltativo: altri modi per rappresentare i grafi

A seconda del contesto, altre codifiche di grafi possono essere più efficienti per l'elaborazione o più facili da utilizzare, a seconda del linguaggio di programmazione e degli strumenti che state utilizzando.

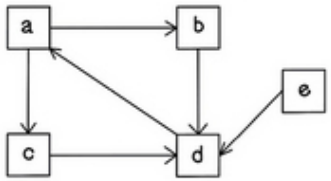
GRAFO NON ORIENTATO

Gli archi non hanno direzione. Le relazioni fra due nodi sono reciproche.



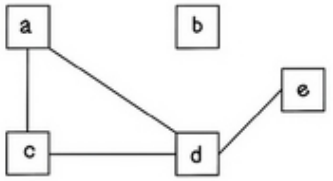
GRAFO ORIENTATO

Gli archi indicano la direzione. Le relazioni fra due nodi sono esplicite.



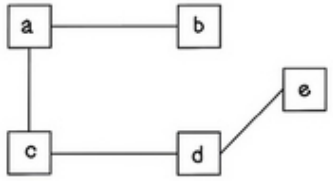
GRAFO DISCONNESSO

Uno o più nodi non sono collegati da alcun arco.



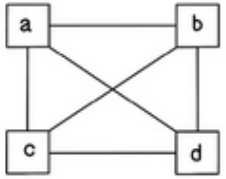
GRAFO ACICLICO

Un grafo che non contiene cicli.



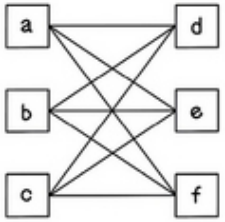
GRAFO COMPLETO

Ogni nodo è connesso a ogni altro nodo da un arco.



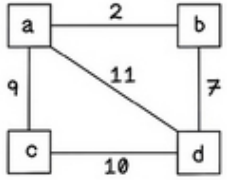
GRAFO BIPARTITO COMPLETO

Ogni nodo di una partizione è connesso a ogni nodo dell'altra partizione tramite archi.



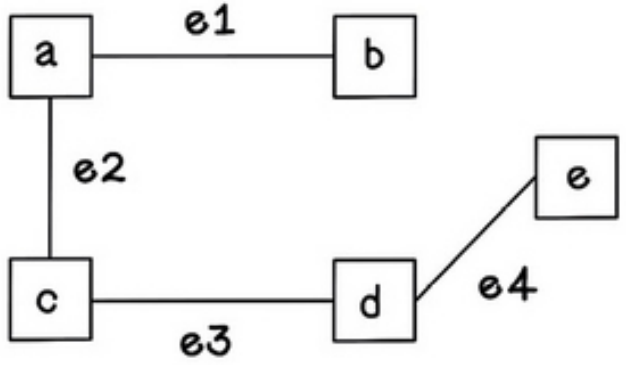
GRAFO PESATO

Un grafo in cui gli archi fra i nodi hanno un peso.



Matrice di incidenza

Una matrice di incidenza utilizza una matrice in cui l'altezza equivale al numero di nodi del grafo e la larghezza è il numero di archi. Ogni riga rappresenta le relazioni di un nodo. Se un nodo non è connesso da un arco, nella cella viene memorizzato il valore 0. Se un nodo riceve una connessione da un altro nodo, nel caso di un grafo orientato, viene memorizzato il valore -1. Se un nodo è connesso in uscita a un altro nodo (o è connesso nel caso di un grafo non orientato), viene memorizzato il valore 1. Una matrice di incidenza può essere utilizzata

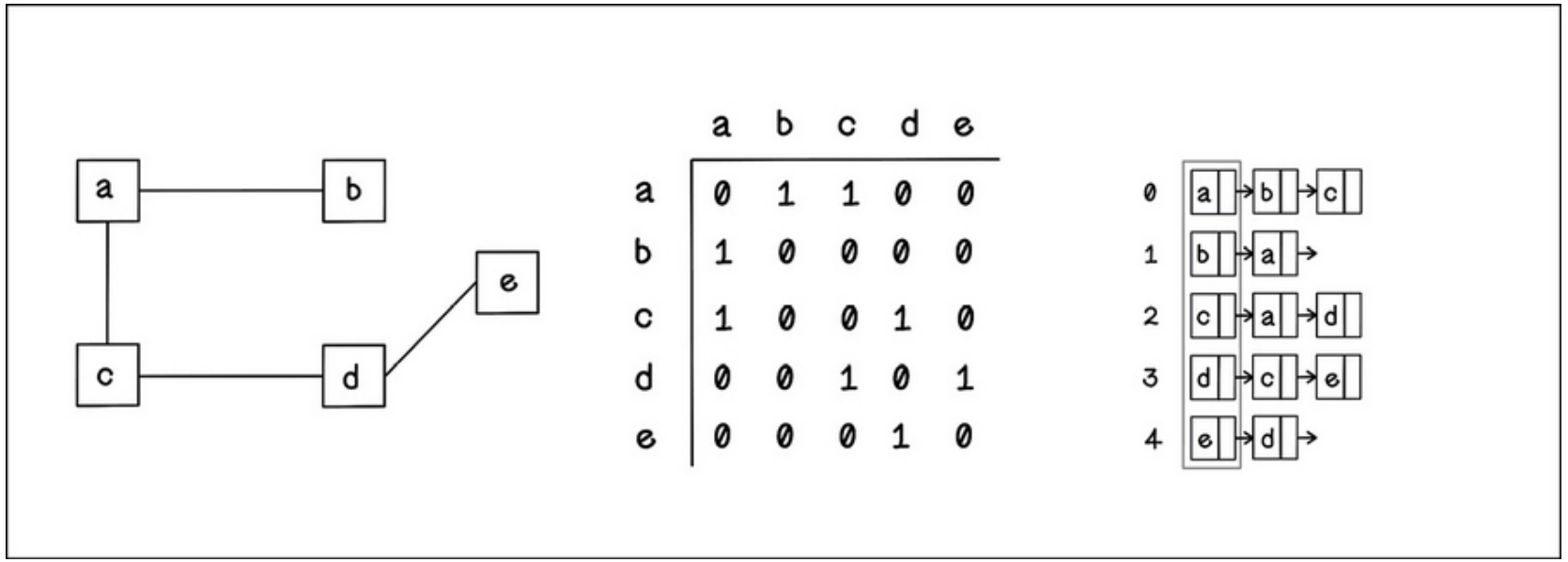


	e1	e2	e3	e4
a	1	1	0	0
b	1	0	0	0
c	0	1	1	0
d	0	0	1	1
e	0	0	0	1

```
[ [ 1, 1, 0, 0 ],
  [ 1, 0, 0, 0 ],
  [ 0, 1, 1, 0 ],
  [ 0, 0, 1, 1 ],
  [ 0, 0, 0, 1 ] ]
```

Lista di adiacenza

Una lista di adiacenza utilizza liste concatenate in cui la dimensione della lista iniziale è il numero di nodi presenti nel grafo e ogni valore rappresenta i nodi connessi a un determinato nodo. Una lista di adiacenza può essere utilizzata per rappresentare grafi sia orientati sia non orientati

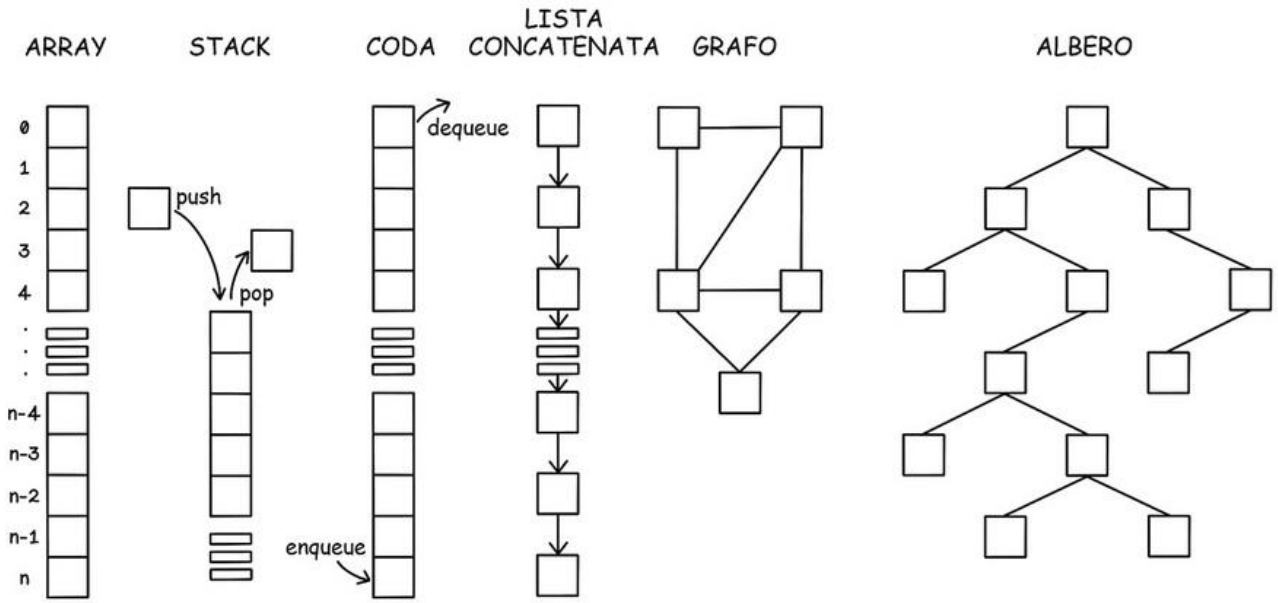


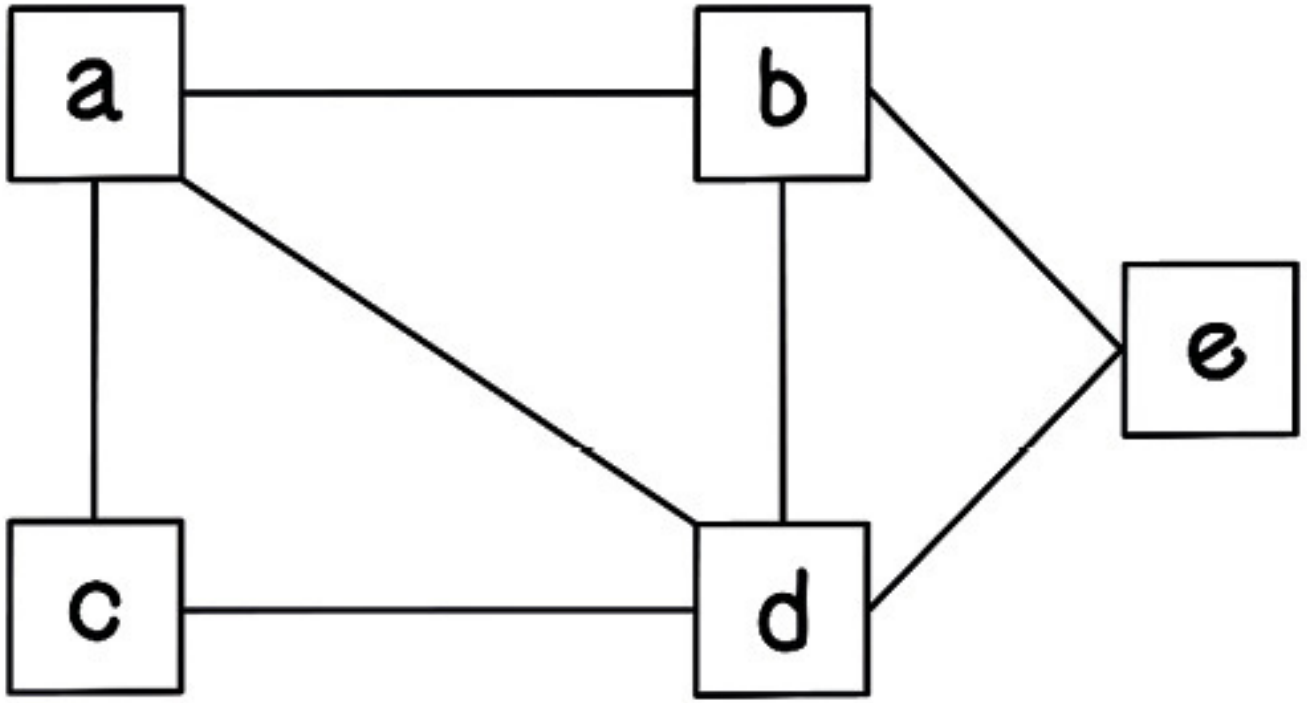
I grafi sono strutture di dati interessanti e utili, perché possono essere facilmente rappresentati come equazioni matematiche, offrendo quindi un supporto a tutti gli algoritmi che utilizziamo. Troverete maggiori informazioni su questo argomento un po' in tutto il libro.

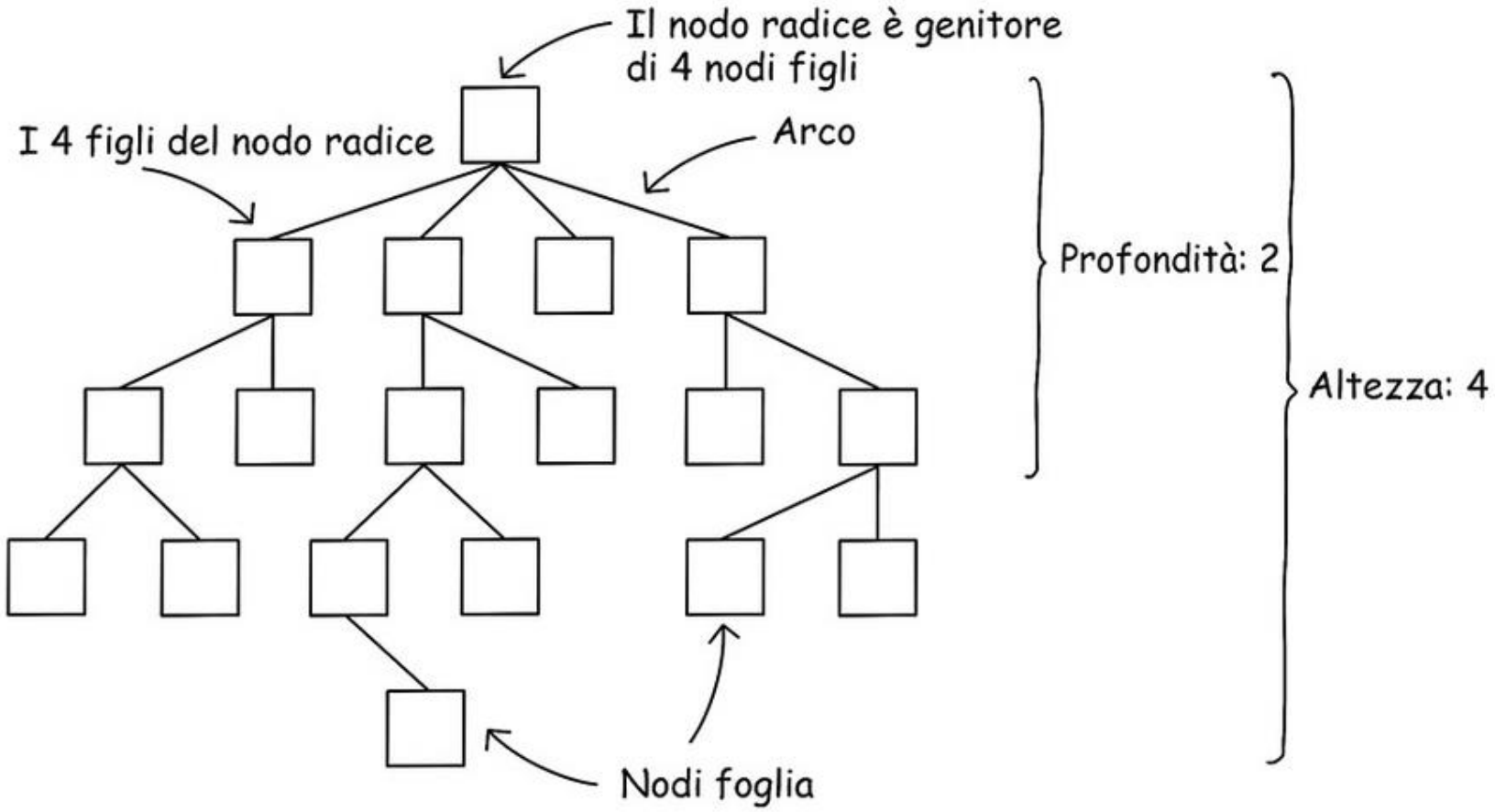
Riepilogo

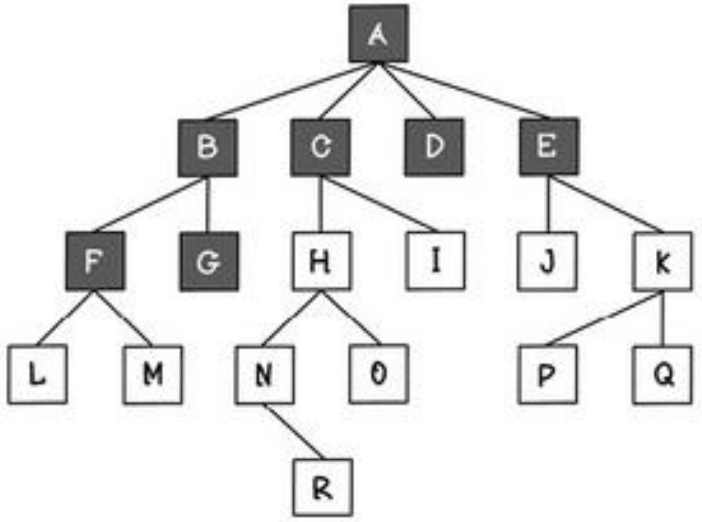
Le strutture di dati sono importanti per risolvere i problemi.

• Gli algoritmi di ricerca sono utili nella pianificazione e ricerca di soluzioni in alcuni ambienti soggetti a cambiamenti.

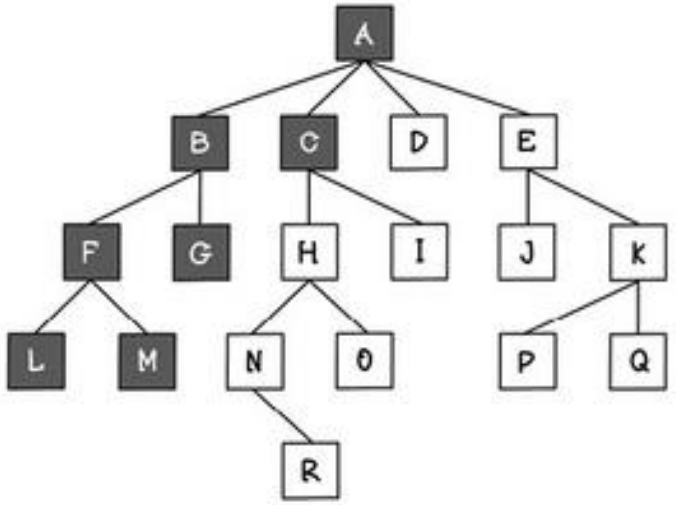



a
b
e
c
d


Il nodo radice è genitore
di 4 nodi figli
I 4 figli del nodo radice
Arco
Profondità: 2
Altezza: 4
Nodi foglia



Ricerca in ampiezza



Ricerca in profondità