Quando l'azione da eseguire non è subito evidente, un agente deve "guardare in avanti" e simulare interiormente tutte le possibili azioni per raggiungere una soluzione. Tali agenti vengono detti **agenti risolutori di problemi**. E il processo che eseguono si chiama **ricerca**.

In ambienti completamente osservabili,discreti,statici,deterministici,episodici e sopratutto noti l'agente può eseguire un processo di risoluzione del problema in 4 fasi:

- **Formulazione dell'obiettivo:** l'agente adotta un obiettivo.
  Gli obiettivi aiutano ad organizzare il comportamento limitando gli scopi e dunque le azioni da considerare
- **Formulazione del problema:** l'agente elabora una descrizione degli stati e delle azioni necessarie per raggiungere un obiettivo.
- **Ricerca:** Prima di effettuare azioni nel mondo reale, l'agente simula nel suo mondo (modello astratto) sequenze di azioni, continuando a cercare finchè trova una sequenza che raggiunge un obiettivo, tale sequenza si chiama **soluzione** (se esiste).
- **Esecuzione:** l'agente esegue le azioni necessarie date dalla soluzione, nel mondo reale.

Un problema di ricerca può essere formulato come segue:

- **Insieme di possibili stati** in cui può trovarsi l'ambiente. (Spazio degli stati)
- **Stato iniziale**, in cui si trova l'agente all'inizio.
- **Stati obiettivo**, descrivono lo stato (o più stati) da raggiungere.
- **Azioni** possibili dell'agente.
- **Modello di transizione** dove viene descritto ciò che fa un agente.
- **Funzione di costo dell'azione** che denota il costo per eseguire un'azione da un certo stato $c(s, a, s')$ denota il costo nel raggiungere s' eseguendo l'azione a partendo da s.

Una sequenza di azioni forma un cammino che è soluzione se porta dallo stato inziale a quello obiettivo. Una soluzione ottima è tra tutte le possibili soluzioni quella che ha il costo minimo.

Un algoritmo di ricerca prende come input un problema di ricerca e restituisce una soluzione o un'indicazione di fallimento.
Ai grafi dello spazio degli stati viene sovrapposto un **albero di ricerca**, formando vari cammini a partire dallo stato inziale e cercano di trovarne uno che raggiunga lo stato goal.

La differenza principale tra spazio degli stati e albero di ricerca, è che il primo descrive l'insieme degli stati del mondo (eventualmente infinito) e le azioni che consentono le transizioni da stato a stato. L'albero di ricerca invece, descrive cammini tra questi stati per il raggiungimento di un certo obiettivo. Notiamo che nell'albero di ricerca potrebbero esservi più cammini che raggiungono qualsiasi stato, per cui nell'albero di ricerca potrebbero esservi più nodi che rappresentano lo stesso stato.

Possiamo **espandere** un nodo considerando tutte le azioni eseguibili a partire da esso, e generare i suoi **successori** (o nodi figli) ovvero nodi raggiungibili dal nodo espanso (detto nodo padre) eseguendo le azioni considerate.

Definiamo inoltre come **frontiera** l'insieme dei nodi generati ma non espansi, e come insieme di nodi **raggiunti** qualsiasi nodo generato (che sia stato espanso o no).

Un primo approccio di strategia di ricerca è la strategia **best-first**, che espande un nodo n tra quelli in frontiera, con valore $f(n)$ minore. $f(n)$ rappresenta una qualsiasi funzione di valutazione. Un nodo viene aggiunto in frontiera se non è mai stato raggiunto prima , o se il cammino appena trovato è il migliore dei precedenti.
Se il nodo scelto è un nodo obiettivo viene restituita la soluzione, altrimenti viene espanso.

Gli algoritmi di ricerca richiedono una struttura dati che tenga traccia dell'albero di ricerca. Ogni nodo è implementato attraverso una struttura dati che mantiene le seguenti informazioni:

- nodo.Stato: ovvero l'informazione di quale stato rappresenti il nodo.
- nodo.Padre: ovvero il nodo padre del nodo corrente che lo ha generato.
- nodo.Azione: ovvero l'azione eseguita nel nodo padre per generare il nodo corrente
- nodo.CostoCammino: ovvero il costo effettivo per raggiungere il nodo corrente a partire dalla radice.

Per mantenere la frontiera viene usata come struttura una coda, tra queste:

- Coda con priorità, viene estratto il nodo con valore $f(n)$ minore.
- Coda LIFO, viene estratto il nodo in cima la coda.
- Stack, viene estratto il nodo in fondo la coda.

Per l'insieme degli stati raggiunti è possibile usare una tabella hash, con chiavi che indicano gli stati e come valori il nodo che lo rappresenta.

In un albero di ricerca può includere cammini ridondanti, quindi più cammini che hanno lo stesso inizio e la stessa destinazione, in più tra questi alcuni non ottimi. Un ciclo è un sotto caso di cammino ridondante.
Per gestire questa situazione sono possibili tre approcci:

- Utilizzare una tabella raggiunti, per tener traccia dei nodi che sono già stati raggiunti e controllare dunque i cammini ridondanti mantendendo soltanto il cammino migliore per ogni stato.
- Non gestire la situazione, in molti problemi è raro trovare cammini ridondanti.
- Risalire la catena dei nodi padre, e verificare se nel cammino vi sono nodi ripetuti. Dunque viene controllata la presenza di cicli ma non di cammini ridondanti.

Adesso vediamo i criteri per valutare un algoritmo di ricerca:

- **Completezza**: l'algoritmo trova la soluzione sempre , e se non esiste restituisce un'indicazione di fallimento.
- **Ottima rispetto al costo:** restuisce il cammino di costo minimo
- **Complessità temporale:** numero di passi che esegue l'algoritmo
- **Complessità spaziale:** quantità di memoria che usa l'algoritmo.

Ricerca non informata

Un algoritmo di ricerca non informata non ha informazioni di quanto uno stato sia vicino allo stato obiettivo

BFS

Quando tutte le azioni hanno stesso costo, una strategia appropriata è la ricerca in ampiezza. Viene espansa prima la radice, poi i suoi successori, e poi i successori di questi e così via.
Viene usata una coda FIFO, che permette l'espansione dei nodi nell'ordine corretto (quelli a profondità maggiore verranno espansi per ultimi e quelli "vecchi",a profondità minore espansi per primi.)
Siccome una volta raggiunto uno stato, quello trovato sarà sicuramente il cammino migliore per raggiungerlo, è possibile effettuare un test obiettivo anticipato una volta generato, inevece che uno posticipato e quindi espandendo il nodo in questione.

La ricerca in ampiezza trova sempre una soluzione con il minimo numero di azioni possibili, perché quando genera i nodi a profondità d, ha già generato tutti i nodi in profondità $d-1$ e se uno di essi fosse stato una soluzione sarebbe stato trovato. L'algortimo è ottimo (se le azioni hanno tutte stesso costo) e completo. Ha una complessità temporale e spaziale di $O(b^d)$

UC

Algoritmo di Dijkstra in informatica teorica. Implementabile attraverso una chiamata Best-First con COSTO-CAMMINO come funzione di valutazione $f(n)$.
Ha complessità temporale e spaziale $O(b^{1+\frac{C^*}{\epsilon}})$, dove C^* è il costo del cammino ottimo e ϵ un limite inferiore posto al costo di ogni azione. Dunque $\frac{C^*}{\epsilon}$ è il numero di mosse nel caso peggiore.
La UC è ottima rispetto al costo e completa, infatti in modo sistematico, esamina tutti i cammini in ordine crescente di costo (se $\epsilon > 0$)

DFS

La ricerca in profondità, espande per prima il nodo a profondità maggiore nella frontiera. Implementata come ricerca ad albero e non ricerca a grafo. Non è ottima, in quanto restituisce sempre la prima soluzione trovata. In spazi degli stati infiniti la ricerca non è sistematica, dunque l'algoritmo non è completo. La complessità temporale è $O(b^d)$.
Il suo vantaggio è che siccome la frontiera è più piccola (tiene in memoria solo i nodi del cammino che sta visitando) e la complessità spaziale è $O(bm+1)$.
Può essere implementata attraverso un algoritmo di backtracking ricorisvo.

RPL

Per evitare che la ricerca in profondità si perda in un cammino infinito, si usa un limite di profondità l,e consideriamo tutti i nodi a questa profondità come se non avessero successori. La complessità temporale è $O(b^l)$ e spaziale $O(bl+1)$. L'algoritmo è completo per problemi in cui si conosce un limite superiore alla profondità della soluzione $(d<l)$. Non è ottimale.

ID

Risolve il problema di trovare un buon valore per il limite l, semplicemente provandoli tutti: $l=0, l=1, ...$ fino a quando non si trova una soluzione. Oppure la RPL restituisce fallimento anziché un valore soglia.
Comp.Spaz: $O(bd+1)$ se esiste soluzione altrimenti $O(bm+1)$
Comp.Temp: $O(b^d)$ se esiste soluzione altrimenti $O(b^m)$
I nodi a profondità d vengono generati 1 volta sola, quelli a livello soprastante due, fino al nodo radice d volte, che nel caso peggiore è:
$d(b)+(d-1)(b^2)+...+b^d=O(b^d)$.