

## Risolutori "classici"

Gli agenti *risolutori di problemi "classici"* assumono:

- ambienti completamente osservabili
- ambienti deterministici
- sono nelle condizioni di produrre offline un piano (sequenza di azioni) che può essere eseguito senza imprevisti per raggiungere l'obiettivo

## Verso ambienti più realistici

La ricerca sistematica, o anche euristica, nello spazio degli stati è troppo costosa, sono necessari *metodi di ricerca locale*.

Dobbiamo inoltre *riconsiderare le assunzioni sull'ambiente*:

Le azioni sono non deterministiche e l'ambiente è parzialmente osservabile. Oppure gli ambienti sono sconosciuti e abbiamo quindi problemi di esplorazione.

### Ricerca locale

---

#### Assunzioni

Gli algoritmi visti fin'ora esplorano gli spazi di ricerca cercando un goal e *restituiscono un cammino soluzione*. A volte però *lo stato goal è la soluzione* del problema. Gli algoritmi di ricerca locale sono adatti per problemi in cui:

- *la sequenza di azioni non è importante*, quello che conta è unicamente lo stato goal
- Tutti gli elementi della soluzione sono nello stato ma *alcuni vincoli sono violati*.

#### Example

Nel problema delle regine nella formulazione a stato completo, ci interessa ottenere la soluzione ma non il path

#### Gli algoritmi

Non sono sistematici e tengono traccia solo del nodo corrente spostandosi poi sui nodi adiacenti.

Non tengono traccia dei cammini (non servono in uscita): sono *efficienti in occupazione di memoria* e possono trovare soluzioni ragionevoli anche in spazi molto grandi e infiniti, come nel caso degli spazi continui.

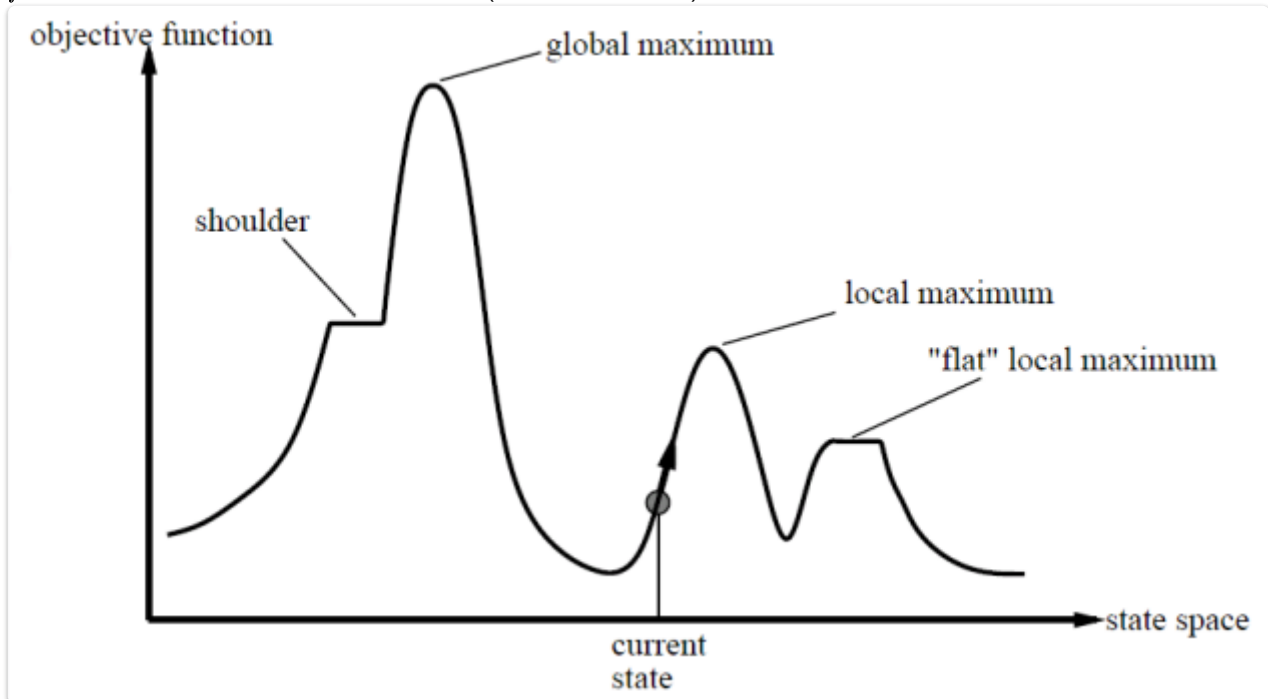
Sono utili per risolvere *problemi di ottimizzazione*: trova lo stato migliore secondo una *funzione obiettivo* e lo stato di *costo minore* (ma non il path).

#### Example

- minimizzare il numero di regine sotto attacco
- training di un modello di Machine Learning

#### Panorama dello spazio degli stati

$f$  è euristica di costo della funzione obiettivo (NON del cammino)



- Uno **stato** ha posizione *sulla superficie* e un'altezza che corrisponde al valore della funzione di valutazione (funzione obiettivo)
- Un **algoritmo** provoca *movimento sulla superficie*
- Utile per trovare l'avvallamento più basso (es. minimo costo) o il picco più alto (es. max di un obiettivo)

### Ricerca in salita - Hill Climbing

#### Steepest ascent/descent

È una *ricerca locale greedy*. Venono generati i successori e valutati, viene scelto un nodo che migliora la valutazione dello stato attuale e non si tiene traccia degli altri, l'albero di ricerca non sta quindi in memoria. I tipi sono:

- si cerca il *migliore*: Hill Climbing a salita rapida/ripida
- se ne cerca *uno a caso* (tra quelli che salgono): Hill Climbing stocastico (anche dipendendo da pendenza)
- si cerca *il primo*: Hill Climbing con prima scelta (il primo generato tra tanti possibili)  
Se non ci sono stati successore migliori, l'algoritmo *termina con fallimento*.

È come il DF ma ancora più economico, non si mantengono in memoria neanche i nodi fratelli, a parte in casi eccezionali.

#### STEEPEST ASCENT

```
function Hill-climbing (problema)
    returns uno stato che è massimo locale
    nodo-corrente = CreaNodo(problema.Stato-iniziale)
    loop do
        vicino = il successore di nodo-corrente di valore più alto
        if vicino.Valore <= nodo-corrente.Valore then
            return nodo-corrente.Stato // interrompe la ricerca
        else nodo-corrente = vicino
        // altrimenti, se vicino è migliore continua
```

#### ⚠ Nota

Si prosegue solo se il vicino (più alto) è migliore dello stato corrente, se tutti i vicini sono peggiori o uguali si ferma

Non c'è frontiera a cui ritornare, *si tiene un solo stato*.

**Tempo:** il numero di cicli è variabile in base al punto di partenza

## PROBLEMA DELLE 8 REGINE

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♙	13	16	13	16
♙	14	17	15	♙	14	16	16
17	♙	16	18	15	♙	15	♙
18	14	♙	15	15	14	♙	16
14	14	13	17	12	14	12	18

Vogliamo minimizzare le coppie di regine che si attaccano, nella figura ci sono già i valori dei successori (di  $h$  se ci si spostasse in quel punto). Tra tutti i possibili successori migliori (in questo caso 12), se ne sceglie *uno a caso*. Non è detto che vada sempre verso lo zero, se si finisce in un *minimo locale, non si può migliorare*. Con questo algoritmo, il problema si blocca l'86% delle volte, in media però in 4 passi si trova la soluzione o in 3 passi si accorge di essersi bloccato.

- *Costo  $h$*  (stima euristica del costo  $f$ ): numero di coppie di regine che si attaccano a vicenda (17 nell'esempio)
- Si cerca il minimo
- I numeri sono i valori dei successori ( $7 \times 8$ , 7 posizioni per ogni regina, su ogni colonna).
- Tra i migliori di pari valore (12) si sceglie a caso
- *Minimo globale* = 0

### MIGLIORAMENTI

1. Consentire (un numero limitato di) *mosse laterali*:  
ci si ferma per  $<$  nell'algoritmo invece che per  $\leq$ , quindi continua anche a parità di  $h$ .
2. *Hill Climbing stocastico*: si sceglie a caso tra le mosse in salita (magari tenendo conto della pendenza) converge più lentamente ma a volte trova soluzioni migliori. Si allontana più agevolmente dalle situazioni di massimo locale.
3. *Hill Climbing con prima scelta*  
Può generare le mosse a caso, uno alla volta, fino a trovarne una migliore dello stato corrente (si prende solo il primo che migliora).  
È come la stocastica ma è utile quando i successori sono molti, evitando una scelta tra tutti. Migliora l'efficienza.
4. *Hill Climbing con riavvio casuale*  
Ripartire da un punto a caso se ci si ferma in un massimo locale: se la probabilità di successo è  $p$ , saranno necessarie in media  $\frac{1}{p}$  ripartenze per trovare la soluzione (es. nelle 8 regine  $p = 0.14 \rightarrow 7$  ripartenze).  
Hill Climbing con random-restart è tendenzialmente completo.  
La riuscita dipende dalla forma del panorama degli stati: molti minimi locali abbassano  $p$  e trovare il buon punto di ripartenza è difficile.

## Tempra Simulata

Questo algoritmo *combina Hill-Climbing con una scelta stocastica* (non del tutto casuale perché sarebbe poco efficiente). Viene fatta un'analogia con il processo di tempra dei metalli in metallurgia: I metalli vengono portati a temperature molto elevate (alta energia/stocasticità iniziale) e raffreddati gradualmente consentendo di cristallizzare in uno stato a (più) bassa energia. All'inizio quindi ammettiamo molte mosse stocastiche e, andando avanti, sempre meno.

#### TEMPRA SIMULATA PER ASCESA

Ad ogni passo *si sceglie un successore  $n'$  a caso*:

- se migliora, lo stato corrente viene espanso
- se peggiora/rimane uguale ( $\Delta E = f(n') - f(n) \leq 0$ ), quel nodo viene scelto con probabilità  $p = e^{\frac{\Delta E}{T}}$ . Si genera un numero casuale tra 0 e 1: se questo è  $< p$  il successore viene scelto, altrimenti no
- Quindi  $p$  è *inversamente proporzionale al peggioramento*, infatti se la mossa peggiora molto,  $\Delta E$  diventa un negativo alto e la  $p$  si abbassa
- **Fattore Temperatura ( $T$ )**: decresce con il progredire dell'algoritmo secondo un piano definito. Al progredire dell'algoritmo, quindi quando  $T$  diventa bassa ( $e^{-\infty} \approx 0$ ), si ferma la probabilità di prendere mosse peggiorative. Si può tornare indietro ma con una probabilità guidata. Si può riformulare lo stesso algoritmo per cercare un minimo invece che un massimo.

#### TEMPRA SIMULATA: ANALISI

La probabilità  $p$  di una mossa in discesa diminuisce col tempo e l'algoritmo si comporta sempre di più come *Hill Climbing*. Se  $T$  viene decrementato abbastanza lentamente con probabilità tendente ad 1, si raggiunge la soluzione ottimale

#### TEMPRA SIMULATA: PARAMETRI

Il valore iniziale ed il decremento di  $T$  *sono parametri*. I valori per  $T$  sono *determinati sperimentalmente*: il valore iniziale di  $T$  è tale che per valori medi di  $\Delta E$ ,  $p = e^{\frac{\Delta E}{T}}$  sia circa 0.5.

#### Ricerca local beam

È la versione locale della *beam search*: si tengono *in memoria  $k$  stati* anziché uno solo.

Ad ogni passo si generano i successori di tutti i  $k$  stati: se si trova un goal ci si ferma, altrimenti si prosegue con i  $k$  migliori tra questi

#### Nota

- diverso da  $K$  - *restart* (che riparte da zero)
- diverso da beam search

#### Beam Search Stocastica

Si introduce un elemento di casualità, come in un processo di *selezione naturale*, per diversificare la nuova generazione. Nella variante stocastica della local beam, si scelgono  *$k$  successori*, ma con probabilità maggiore per i migliori. Non si scelgono i successori migliori, ma quelli con maggiore probabilità di portare ai migliori.

#### Terminologia:

- *organismo* - stato
- *progenie* - successori
- *fitness* - valore della  $f$  - idoneità

Introduciamo questa terminologia perché vogliamo parlare di processi che simulano la selezione naturale.

#### Algoritmi genetici / evolutivi: funzionamento

La *popolazione iniziale* è formata da  $k$  stati/*individui* generati casualmente, ognuno dei quali è *rappresentato come una stringa*.

#### Example

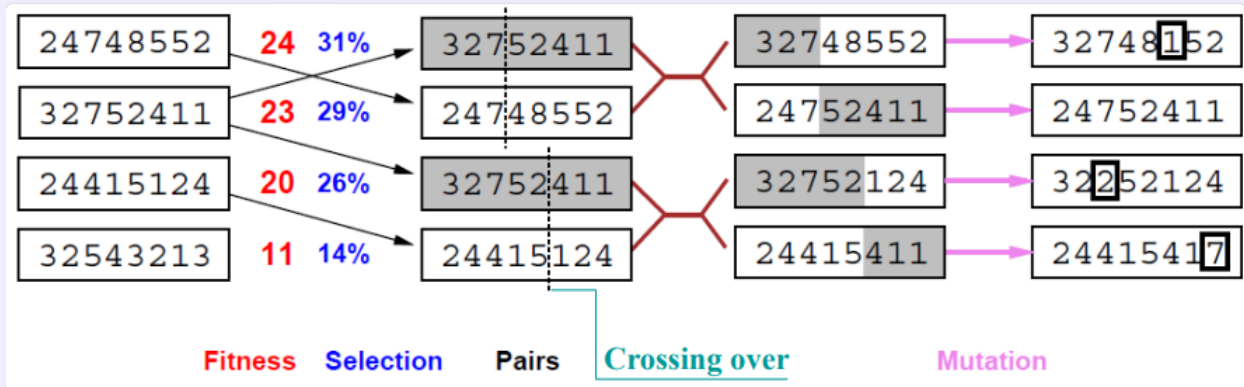
posizione nelle colonne ("24748552") stato delle 8 regine o con 24 bit

Gli individui sono *valutati da una funzione di fitness*.

Si *selezionano gli individui per gli "accoppiamenti"* con una probabilità proporzionale alla fitness. Le coppie danno vita alla *generazione successiva* combinando materiale genetico o con un meccanismo aggiuntivo di mutazione genetica.

La popolazione ottenuta dovrebbe essere migliore, non è sempre detto, ma di base i genitori migliori generano una progenie migliore. La cosa si ripete fino ad ottenere stati abbastanza buoni (stati obiettivo) o finché non miglioriamo più.

#### Example



Per ogni coppia (scelta con *probabilità* proporzionale alla *fitness*) viene scelto un punto di crossing over e vengono generati due figli scambiandosi pezzi (del DNA).

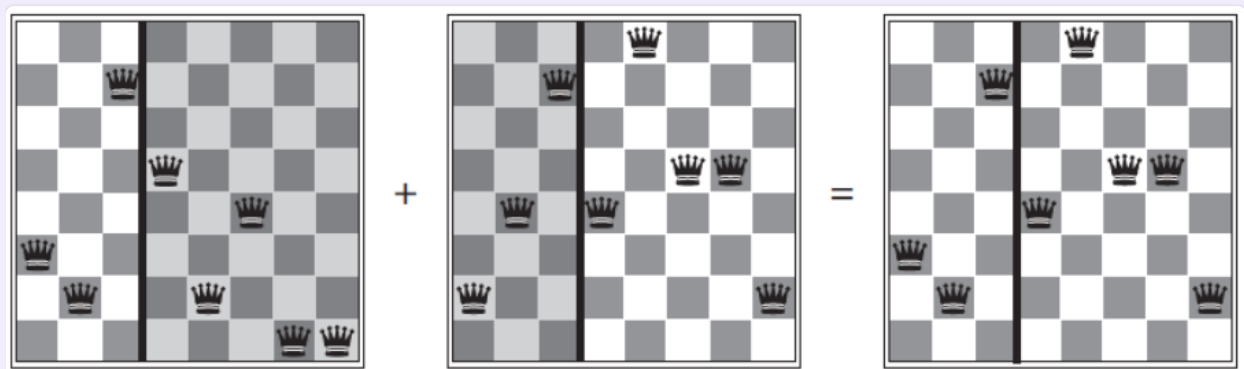
Viene infine effettuata una mutazione casuale che dà luogo alla prossima generazione.

La fitness progressivamente tenderà a favorire generazioni migliori.

Emula i meccanismi genetici ma anche l'evoluzione della specie.

#### NASCITA DI UN FIGLIO

#### Example



Le parti chiare sono passate al figlio, quelle grigie si perderanno.

Se i genitori sono molto diversi anche i nuovi stati lo saranno, se sono molto simili, l'evoluzione tende a rallentare. All'inizio gli spostamenti saranno maggiori, si raffinano andando avanti.

#### ALGORITMI GENETICI

Il **Natural Computing** prende ispirazione dai meccanismi naturali: combinano la tendenza a salire della *beam search* *stocastica* e l'interscambio di informazioni (indiretto) tra thread paralleli di ricerca (blocchi utili che si combinano).

Funziona meglio se il problema (e le sue soluzioni) ha componenti significative rappresentate in *sottostringhe*, questa necessità è anche il suo *punto critico*.

#### Example

Le formiche inizialmente vanno in giro a caso, quando una raggiunge l'obiettivo rilascia dei feromoni per comunicare con le altre. La formica che ha raggiunto l'obiettivo facendo il percorso più breve, rilascia la scia di feromoni più intensa, quindi seguendo quella, tutte le formiche faranno il percorso più breve.

## SPAZI CONTINUI

Fino ad ora ci siamo mossi su spazi discreti, ma molti problemi reali hanno spazi di ricerche continui (fondamentale per il Machine Learning).

Lo **stato** è descritto da **variabili continue**  $x_1, \dots, x_n$ , rappresentabili con un **vettore**  $x$ . In questo caso il **fattore di ramificazione è infinito**, ma abbiamo molti strumenti matematici che portano ad approcci efficienti.

### Gradient

Se la  $f$  è continua e differenziabile (es. quadratica rispetto al vettore  $x$ ), il minimo o il massimo si può cercare usando il **gradiente**, che restituisce la direzione di massima pendenza del punto (per noi in che direzione andare/di quanto).

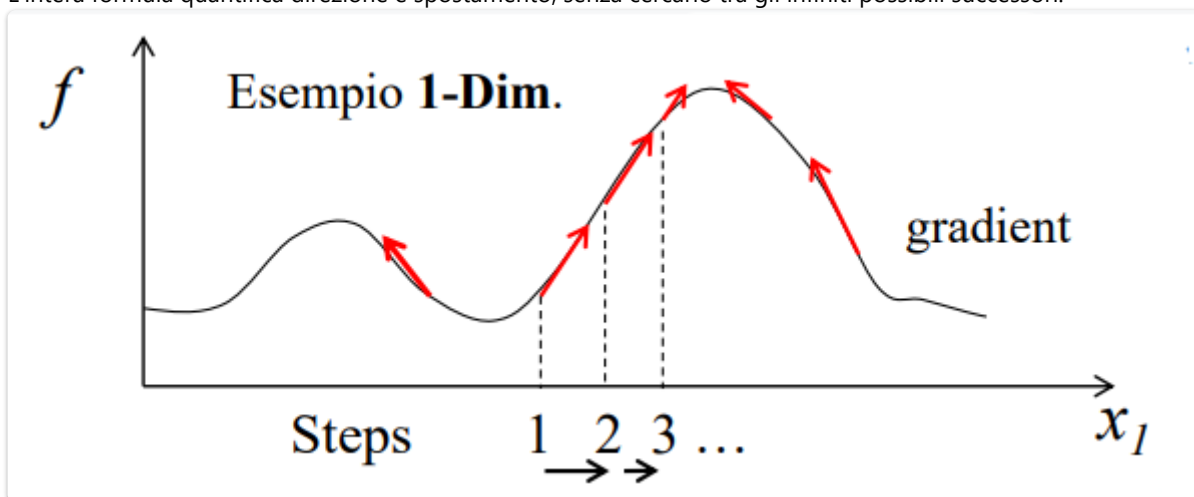
Data una  $f$  obiettivo su 3D:

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3} \right)$$

**Hill Climbing Iterativo:**  $x_{new} = x \pm \eta \nabla f(x)$

In questa equazione si usa  $+$  per salire (maximization) e  $-$  per scendere (minimization), la  $\eta$  rappresenta la step size, è una costante tipicamente minore di 1.

L'intera formula quantifica direzione e spostamento, senza cercarlo tra gli infiniti possibili successori.



## Oltre la ricerca classica - Cenni

### Ambienti più realistici

Gli **agenti risolutori di problemi "classici"** assumono:

- Ambienti completamente osservabili
- Azioni/ambienti deterministici
- Il piano generato è una sequenza di azioni che può essere generato offline e eseguito senza imprevisti
- Le percezioni non servono se non nello stato iniziale

### Soluzioni più complesse

In un ambiente parzialmente osservabile e non deterministico **le percezioni sono importanti**: restringono gli stati possibili e informano sull'effetto dell'azione.

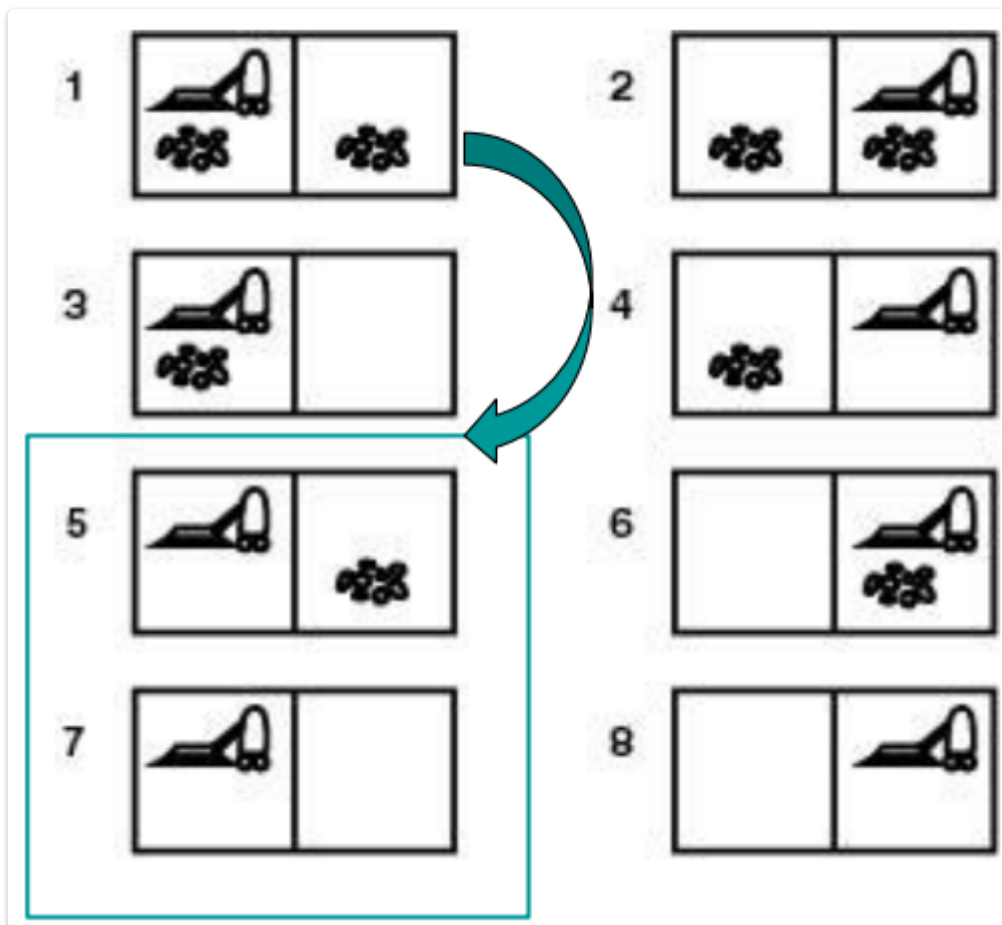
Più che un piano l'agente può elaborare una **"strategia"**, che tiene di conto delle diverse eventualità: un **piano con contingenza**.

| succede una cosa, ma se succede questa faccio quest'altra

## Azioni non deterministiche - aspirapolvere imprevedibile

**Comportamento:** Se aspira in una stanza sporca, la pulisce ma talvolta pulisce anche una stanza adiacente. Se aspira in una stanza pulita, a volte rilascia sporco.

**Varianzioni necessarie al modello:** Il modello di transizione restituisce un *insieme di stati*, l'agente non sa in quale si troverà. Il piano *di contingenza* sarà un piano condizionale e magari con cicli.



**Esempio:**  $Risultati(Aspira, 1) = \{5, 7\}$

**Piano possibile:** `[Aspira, if stato = 5 then [Destra, Aspira] else []]`

**Soluzione:** da una sequenza di azioni a piano (albero)

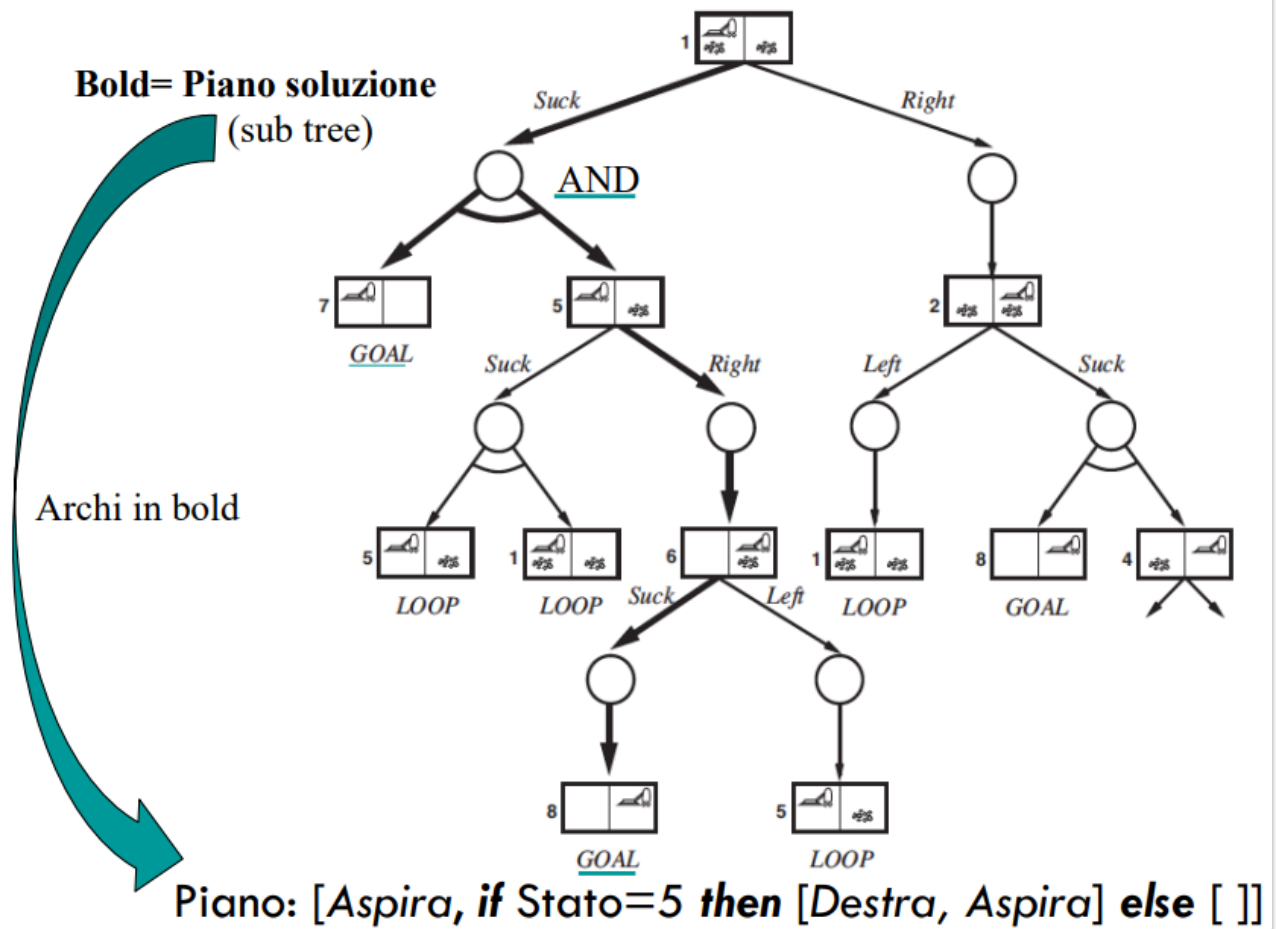
### Come si pianifica: alberi di ricerca AND-OR

In un **albero di ricerca AND-OR**

- I **nodi OR** rappresentano le scelte dell'agente (1 sola azione)
  - I **nodi AND** le diverse contingenze (scelte dell'ambiente, più stati possibili). Sono **tutte da considerare**
- Una **soluzione** a un problema di ricerca AND-OR è un **albero** che:
- ha un **nodo obiettivo** in ogni foglia
  - specifica un'**unica azione** nei nodi OR

- include tutti gli *archi uscenti da nodi AND* (tutte le contingenze)

## Esempio di ricerca AND-OR



Piano: [Aspira, if Stato=5 then [Destra, Aspira] else [ ]]