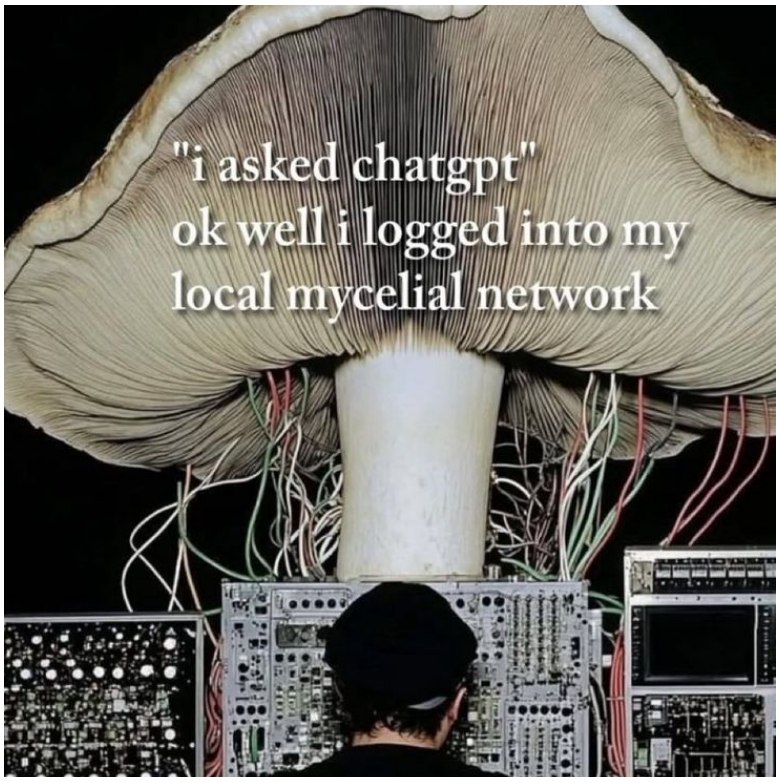
Sapienza Università di Roma

Intelligenza Artificiale

libro del corso:
Artificial Intelligence: A Modern Approach
(Norvig, Russell)



aglaia norza

9 aprile 2026

thisisaglaia@gmail.com
github.com/AglaiaNorza

Indice

A.1

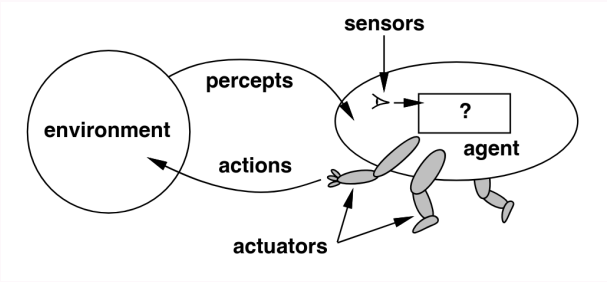
Agenti Intelligenti

A.1.1 Agenti, ambienti, razionalità

Def. 1: **Agente**

Un agente è qualsiasi sistema che:

- percepisce il suo ambiente mediante **sensori**
- agisce sull'ambiente mediante **attuatori**



Ci è utile introdurre alcuni termini specifici relativi agli agenti e ai loro ambienti:

- **percezioni** (insieme $\mathcal{P}$): input percettivi dell'agente in un dato istante
- **sequenza percettiva**: la storia di quello che l'agente ha percepito nella sua intera esistenza
- **funzione agente**: descrizione matematica del comportamento dell'agente, che può dipendere dalla sua intera sequenza percettiva (ma non da qualcosa che non ha percepito)

$$f : \mathcal{P}^* \to \mathcal{A}$$

  in cui $\mathcal{A}$ è l'insieme delle azioni che l'agente esegue

- **programma agente**: implementazione (interna al sistema fisico) della funzione agente

Def. 2: **Agente Razionale**

Definiamo (almeno per adesso) come **agente razionale** un agente che sceglie le azioni che *massimizzano il valore atteso della sua misura di prestazione*, data la sequenza di percezioni e ogni altra sua conoscenza pregressa.

La **misura di prestazione** è una misurazione oggettiva della desiderabilità della sequenza di

stati attraversati dall'ambiente a seguito delle azioni dell'agente.

- deve essere valutata oggettivamente sugli stati dell'ambiente, non dell'agente
- di solito è preferibile una misura che considera gli effetti delle azioni piuttosto che le azioni stesse

Esempio: agente aspirapolvere

Consideriamo un semplice agente aspirapolvere che si muove in un ambiente composto da due mattonelle. Un agente reattivo che aspira se la mattonella dove si trova è sporca e si sposta nell'altra mattonella se è pulita risulta razionale rispetto a una misura di prestazione che assegna semplicemente un punteggio per ogni riquadro pulito.

```
function agente_reattivo_aspirapolvere([posizione, stato]):
    if stato = Sporco then return Aspira;
    else
        if posizione = A then return Destra;
        else return Sinistra; /* posizione = B */
```

Tuttavia, un agente simile continuerà a muoversi in eterno tra una mattonella e l'altra anche quando entrambe sono pulite. Se la misura di prestazione fosse formulata in modo diverso, introducendo ad esempio penalità per ogni movimento effettuato, per il consumo di elettricità o per il rumore prodotto, questo stesso agente non sarebbe più razionale a causa del suo inutile dispendio di energie

Il primo passo nella progettazione di un agente è la definizione del suo **task environment**. Un task environment è composto da:

- **P**erformance (misura di prestazione)
- **E**nvironment (ambiente)
- **A**ctuators (attuatori)
- **S**ensors (sensori)

Esempi di PEAS

Agente	**P**restazione	**E**nvironment	**A**ttuatori	**S**ensori
Taxi automatico	Sicurezza, velocità, legalità, comfort, profitto	Strade, altri veicoli, pedoni, clienti	Sterzo, acceleratore, freni, frecce, sintesi vocale	Telecamere, sonar, tachimetro, GPS, accelerometro, microfono
Diagnosi medica	Guarigione, costi minimi, assenza di denunce	Paziente, staff medico	Schermo	Tastiera
Shopping online	Prezzo, qualità, attinenza ai desideri	Siti web dei negozi, corrieri	Schermo, vai-a-URL, riempimento form.	Pagine HTML da interpretare

È possibile classificare gli ambienti secondo alcune coordinate importanti per la progettazione di agenti:

Dimensione	Tipi di ambienti	Cosa riguardano
Osservabilità	Totalmente, parzialmente, inosservabile	Capacità dei sensori di dare accesso a tutte le proprietà rilevanti dello stato dell'ambiente in ogni momento.
Agenti	Mono-agente, multi-agente	Presenza di altri agenti che possono *cooperare* o *competere* per obiettivi o risorse.
Determinismo	Deterministico, stocastico, non-deterministico	Indica se lo stato successivo è determinato univocamente da stato corrente e azione (*stocastico*: incertezza gestita con distribuzioni di probabilità; *non-deterministico*: azioni definite dal set di risultati possibili, spesso richiede successo per *tutti* gli esiti)
Episodicità	Episodico, sequenziale	Specifica se l'esperienza è divisibile in episodi atomici indipendenti o se le scelte influenzano il futuro.
Dinamicità	Statico, dinamico, semi-dinamico	Indica se l'ambiente (*dinamico*) o la misura di prestazione (*semi-dinamico*) muta mentre l'agente riflette.
Continuità	Discreto, continuo	Indica se stato, ambiente, percezioni, azioni sono discrete (interi) o continue (reali). Un ambiente continuo può essere discretizzato
Conoscenza	Noto, ignoto	Denota la conoscenza a priori delle regole "fisiche" dell'ambiente e delle conseguenze delle azioni.

A.1.1.1 Tipologie di agenti

Ci sono quattro principali **tipi di agente**:

- agenti reattivi semplici
- agenti reattivi basati su modello
- agenti basati su obiettivi
- agenti basati sull'utilità

Ad ognuno di questi tipi è possibile aggiungere la capacità di *apprendere*.

A.1.1.1.1 Agenti reattivi semplici

Le azioni degli agenti reattivi semplici sono determinate esclusivamente dalla **percezione corrente**.

```
function a_r_aspirapolvere([posizione, stato]): azione
    if stato = Sporco then return Aspira;
    else
        if posizione = A then return Destra;
        else return Sinistra;
```

Gli agenti reattivi semplici seguono una "condition-action rule" (`if condizione then azione`)

- è utile che agenti complessi abbiano parti puramente reattive

A.1.1.1.2 Agenti reattivi basati su modello

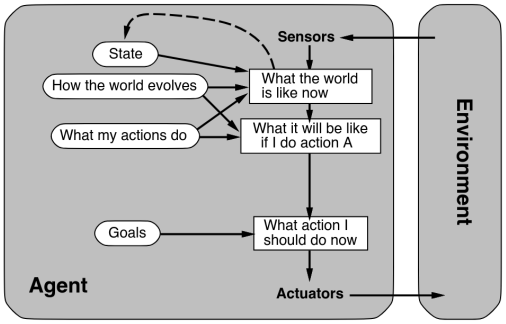
Le azioni degli agenti reattivi basati su modello sono detereminate dalla percezione corrente e da un "modello del mondo", una rappresentazione interna della parte del mondo che l'agente non può vedere all'istante corrente.

Il modello deve essere conosciuto a priori (in qualche modo "hard-coded" nell'agente)

```
function MODEL-BASED-REFLEX-AGENT(percept): action
   persistent: state, the agent's current conception of the world state
               model, a description of how the next state depends on current state and
                   action
               rules, a set of condition-action rules
               action, the most recent action, initially none

   state ← UPDATE-STATE(state, action, percept, model)
   rule ← RULE-MATCH(state, rules)
   action ← rule.ACTION
   return action
```

A.1.1.1.3 Agenti basati su obiettivi

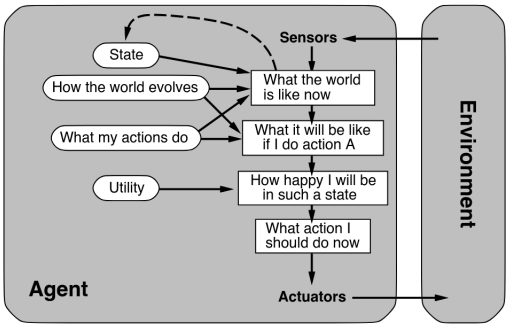


In questa tipologia, l'azione è determinata dalla percezione corrente, dallo stato e dagli obiettivi dell'agente (la sola conoscenza dello stato e delle percezioni spesso non è sufficiente per prendere una decisione)

> Consideriamo un pilota di taxi - se la percezione corrente è un incrocio, l'agente non può decidere se girare a destra o a sinistra senza conoscere la destinazione finale; l'obiettivo fornisce questa informazione.

Non sempre l'obiettivo è raggiungibile tramite una singola azione. In questi casi l'agente deve ricorrere a tecniche di **ricerca e pianificazione** per calcolare intere sequenze di azioni necessarie per raggiungere l'obiettivo desiderato (l'agente sa come sarà il mondo dopo ogni possibile mossa, e sceglie quelle che lo portano all'obiettivo).

A.1.1.1.4 Agenti basati sull'utilità



L'azione, in questo caso, è determinata dalla percezione corrente, dallo stato e dagli obiettivi dell'agente, ed è esplicitamente mirata a raggiungere un'alta **utilità**.

Mentre l'obiettivo definisce uno stato da raggiungere, l'utilità fornisce un criterio per valutare *come* arrivarci.

> Se ci sono più percorsi per arrivare alla meta, l'agente pilota di taxi dovrà chiedersi quale sia il percorso più veloce/affidabile/sicuro/ecc.

La funzione di utilità rappresenta un'interiorizzazione della misura di prestazione dell'ambiente (considera quanto sono "buoni" gli stati intermedi), e aiuta concretamente l'agente a comportarsi in modo razionale.

Se l'agente si trova a operare in ambienti non totalmente osservabili o non deterministici, esso agirà in modo da massimizzare l'**utilità attesa** (ovvero la media ponderata delle utilità dei vari risultati possibili).

A.1.1.1.5 Agenti in grado di apprendere

Gli agenti visti in precedenza funzionano bene solo se gli scenari rimangono sempre gli stessi. Se si introducono per esempio elementi *greedy*, il programma può sbagliare; per questo motivo è necessario introdurre un elemento di apprendimento che permetta all'agente di esplorare e sperimentare su stati **nuovi**.

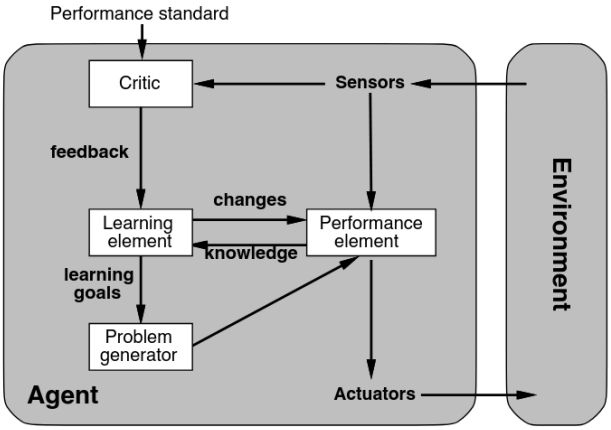
Ad ognuno dei tipi di agenti finora descritti è possibile aggiungere la capacità di apprendere. In generale, l'apprendimento negli agenti intelligenti può essere riassunto come un processo di modifica di ogni componente dell'agente per allinearli maggiormente alle informazioni di feedback disponibili, migliorandone così le prestazioni complessive.

L'architettura di un agente con capacità di apprendimento è suddivisa in quattro componenti fondamentali:

- **Performance element** (elemento esecutivo): l'"agente" vero e proprio descritto in precedenza (quello che riceve le percezioni e seleziona le azioni) - a questo punto, però, non sarà più costituito da codice statico, ma da una **struttura dati di rappresentazione**, in modo che il suo comportamento possa essere effettivamente modificato e aggiornato.
- **Critic** (elemento critico): Valuta le azioni compiute dall'agente confrontandole con un *performance standard* (standard di prestazione) e fornisce un *feedback*. È necessario perché le percezioni di per sé non indicano all'agente se ha avuto successo (ad esempio, un programma di scacchi sa di aver fatto scacco matto dalle percezioni, ma ha bisogno di uno standard per sapere che "vincere" è un evento positivo).

  - Lo standard di prestazione deve essere **fisso** e considerato concettualmente esterno all'agente: l'agente non deve poterlo modificare per adattarlo al proprio comportamento.
  - Distingue parte della percezione in entrata sotto forma di **premio** (*reward*) o **penalità** (*penalty*), fornendo un feedback diretto sulla qualità del comportamento.

- **Learning element** (elemento di apprendimento): riceve il feedback dal *critic* e conosce il programma agente. Il suo compito è suggerire i cambiamenti per generare modifiche nel modo di comportarsi dell'agente. Inoltre, acquisisce e memorizza nuova conoscenza (*knowledge*) sull'ambiente.
  - L'osservazione degli stati successivi all'azione consente all'agente di imparare "cosa fanno le azioni" e "come evolve il mondo" (ad esempio, capire quanto il taxi riesca effettivamente a frenare su una strada bagnata).
- **Problem generator** (generatore di problemi): genera nuovi obiettivi di apprendimento (*learning goals*) suggerendo azioni che portino ad esperienze nuove e "formative". È il componente che spinge l'agente a uscire dalla sua routine per sperimentare su stati nuovi.
  - Se l'elemento esecutivo fosse lasciato solo, continuerebbe a eseguire all'infinito unicamente le azioni che ritiene migliori in base alle sue limitate conoscenze attuali.
  - Il problem generator spinge l'agente a esplorare, compiendo azioni che nel breve termine potrebbero sembrare *sub-ottimali*, ma che portano a scoprire alternative migliori per il lungo termine.



A.2

Esplorazione sistematica di Spazi degli Stati

A.2.1 Agenti risolutori di problemi

Gli agenti risolutori di problemi sono agenti basati su obiettivi che considerano gli stati come **entità atomiche** (prive di struttura interna). Operano tipicamente in ambienti discreti, totalmente osservabili, noti, deterministici e statici.

In queste condizioni favorevoli, l'agente può calcolare in anticipo un'intera sequenza di azioni (**piano**) per raggiungere l'obiettivo ed eseguirla "ad occhi chiusi", certo del risultato.

A.2.1.1 Formulazione del problema

Def. 3: **Problema di ricerca**

Un problema di ricerca è definito da cinque componenti:

- **Stato iniziale**: lo stato in cui l'agente inizia la ricerca.
- **Azioni**: l'insieme delle azioni possibili in un dato stato s, denotato come $Azioni(s)$.
- **Modello di transizione**: funzione $Risultato(s, a)$ che descrive la fisica dell'ambiente, restituendo lo stato risultante dall'esecuzione dell'azione a nello stato s.
- **Test obiettivo**: condizione o funzione che determina se un dato stato soddisfa l'obiettivo.
- **Costo di cammino**: funzione che assegna un costo numerico a ogni cammino (solitamente calcolata come la somma dei costi di passo).

Una **soluzione** è una sequenza di azioni che individua un cammino dallo stato iniziale a uno stato obiettivo. Una **soluzione ottima** è la soluzione a costo minimale.

A.2.1.2 Stato vs. Nodo

Durante l'esplorazione, l'algoritmo mantiene una **frontiera** di ipotesi aperte. È fondamentale distinguere tra stato e nodo:

- **Stato**: una rappresentazione di una configurazione del mondo (spesso un'"etichetta")
- **Nodo**: struttura dati componente dell'albero di ricerca. Oltre allo stato, un nodo include il puntatore al nodo padre, l'azione generatrice, la profondità e il costo di cammino $g(n)$ calcolato fino a quel punto.

> Ricerca su albero vs su grafo
>
> La presenza di cammini ciclici o stati ripetuti può generare un albero di ricerca infinito anche in problemi con uno spazio degli stati finito. Per ovviare a ciò, la ricerca su grafo memorizza gli stati già visitati in un insieme **esplorati** (closed set), espandendo solo nodi che portano a stati non ancora visitati o non presenti in frontiera.

A.2.2 Algoritmi di Ricerca Non Informata

Le strategie non informate possiedono solo la formulazione del problema; l'algoritmo può solo generare successori e testare l'obiettivo, senza sapere se uno stato sia "meglio" di un altro in prospettiva.

A.2.2.1 Ricerca in ampiezza (BFS - Breadth-First Search)

- **Strategia**: espande il nodo più vicino alla radice. Usa una coda FIFO per la frontiera.
- **Completezza**: sì, se il fattore di ramificazione b è finito.
- **Ottimalità**: se il costo di passo è costante (sse un cammino è funzione monotona della profondità)
- **Complessità**: tempo e spazio $O(b^d)$ (dove d è la profondità della soluzione)
  il problema più grande è l'occupazione di memoria, che cresce esponenzialmente

A.2.2.2 Ricerca Uniform-Cost (Min-Cost)

- **Strategia**: espande il nodo con il costo di cammino $g(n)$ minimo. La frontiera è una coda di priorità.
- **Completezza/Ottimalità**: se b è finito e tutti i costi di passo sono $\geq \epsilon > 0$.
- **Complessità**: tempo e spazio $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

A.2.2.3 Ricerca in profondità (DFS - Depth-First Search)

- **Strategia**: espande sempre il nodo più profondo. Usa una coda LIFO (pila).
- **Completezza**: solo in spazi finiti e se evita i cammini ridondanti (ricerca su grafo). Si perde facilmente in rami infiniti.
- **Ottimalità**: no.
- **Complessità**: tempo $O(b^m)$ (dove m è la profondità massima dell'albero)
  il vantaggio principale è lo Spazio: se implementata ad albero, richiede solo $O(bm)$

A.2.2.4 Ricerca ad approfondimento iterativo (IDS)

L'IDS esegue iterativamente ricerche a profondità limitata, aumentando il limite ad ogni passaggio $(0, 1, 2, \dots)$ finché non trova l'obiettivo.

- **Pro**: unisce i vantaggi di BFS e DFS. È completa e ottima (come BFS) ma richiede una memoria lineare $O(bd)$ (come DFS).

- **Tempo**: $O(b^d)$ - la ripetizione dei livelli superiori sembra uno spreco, ma incide poco asintoticamente in quanto la maggior parte dei nodi si trova nei livelli più bassi (vengono ripetuti "pochi" nodi, solo quelli più vicini alla radice)

A.2.3 Algoritmi di Ricerca Informata

Gli algoritmi di ricerca informata possiedono conoscenza specifica del dominio, incarnata in una **funzione euristica** $h(n)$.

- $h(n)$: stima del costo minimo per raggiungere l'obiettivo dal nodo n. È 0 se il nodo è una soluzione, e > 0 altrimenti.

A.2.3.1 Ricerca Best-First Greedy

- **Strategia**: estrae dalla frontiera il nodo che *sembra* più vicino all'obiettivo, ordinando i nodi esclusivamente per $h(n)$.
- **Limitazioni**: dimentica il passato (il costo già sostenuto) e non è garantita per trovare soluzioni ottime; si tratta l'approccio greedy può rivelarsi fuorviante.

A.2.3.2 Ricerca A^*

L'algoritmo A^* risolve i problemi del Best-First Greedy combinando il costo effettivo passato e la stima futura. La frontiera è ordinata in base alla funzione di valutazione:

$$f(n) = g(n) + h(n)$$

Dove $f(n)$ rappresenta il costo stimato del cammino più economico che passa per n.

A.2.3.2.1 Ottimalità di A^*

La garanzia che A^* restituisca una soluzione ottima dipende strettamente dalle proprietà dell'euristica $h(n)$.

- **Ammissibilità**: un'euristica è *ammissibile* se non sovrastima mai il costo per raggiungere l'obiettivo ($h(n) \leq$ costo reale minimo). È in pratica una stima ottimistica.

> Thm. 1: **Euristica ammissibile su albero**
>
> Con h ammissibile, A^* su *albero* trova sempre la soluzione ottima

- **Consistenza** (o monotonia): un'euristica è *consistente* se rispetta la disuguaglianza triangolare: per ogni nodo n e suo successore n', vale: $h(n) \leq costo(n, n') + h(n')$.

  $\rightarrow$ un'euristica consistente è implicitamente anche ammissibile

> Thm. 2: **Euristica consistente**
>
> Con h consistente, A^* su *grafo* trova sempre la soluzione ottima. I valori di $f(n)$ lungo un cammino risultano non decrescenti.

Il problema principale di A^* è la complessità spaziale.

A.2.3.2.2 Sviluppare Euristiche

Se si dispongono di due euristiche ammissibili h_1 e h_2, e per ogni nodo $h_2(n) \geq h_1(n)$, si dice che h_2 **domina** h_1. L'euristica dominante (a valori più alti) è preferibile poiché farà espandere ad A^* un numero minore o uguale di nodi.

Le euristiche ammissibili spesso si derivano calcolando il costo esatto di una soluzione in una versione rilassata (con meno vincoli) del problema originale.

A.3 Algoritmi a Miglioramento Iterativo [WIP]

A.3.1 Ricerca Locale

Negli algoritmi di ricerca visti fino ad ora, l'obiettivo era trovare un *cammino* verso la soluzione. In molti problemi di ottimizzazione, tuttavia, il percorso per raggiungere la meta è irrilevante: ciò che conta è solo lo **stato finale** (es. problema n-queens).

In questi casi si usano algoritmi di **ricerca locale** che operano utilizzando una formulazione a *stato completo*: partono da una configurazione sub-ottimale e applicano modifiche iterative per migliorarne la qualità (o minimizzarne il costo), esplorando solo il vicinato dello stato corrente.

Vantaggi principali:

- usano pochissima memoria (spesso tengono traccia di un solo nodo corrente)
- possono trovare soluzioni ragionevoli in spazi degli stati infiniti o continui dove gli algoritmi sistematici classici fallirebbero

A.3.1.1 Hill Climbing (Ricerca in salita)

È un algoritmo di tipo **greedy** che si sposta sempre in direzione di un valore crescente (o decrescente, se sta minimizzando un costo), muovendosi verso il "vicino" migliore. Termina quando raggiunge un "picco" ("massimo locale") in cui nessun vicino ha un valore più alto.

> "cercare di raggiungere la cima del monte Everest in una fitta nebbia, soffrendo di amnesia".

- **Problemi ricorrenti**:
  - **massimi locali**: picchi che sono più alti dei vicini immediati, ma più bassi del massimo globale - l'algoritmo si blocca
  - **creste (ridges)**: sequenze di massimi locali non connesse direttamente tra loro - l'algoritmo è costretto a "oscillare" e procede a fatica
  - **plateaux**: aree piatte dello spazio degli stati; possono essere massimi locali piatti o "spalle" da cui è ancora possibile salire - l'algoritmo finisce per vagare casualmente

```
function hill_climbing(problema): stato ottimo locale
  nodo_corrente <- crea_nodo(problema.stato_iniziale);
  while true do
    mossa_trovata <- false;
    while mossa_trovata = false and nodo_corrente ha ancora vicini da considerare do
      vicino <- prossimo 'vicino' di nodo_corrente;
```

```
      if vicino è migliore o della stessa qualità di nodo_corrente then
        /* effettua mosse laterali */
        nodo_corrente <- vicino;
        mossa_trovata <- true;
    if mossa_trovata = false then
      return nodo_corrente.stato;
```

Listing A.3.1: Hill-climbing con prima scelta

finisci sezione

A.4

Problemi di Soddisfacimento di Vincoli (CSP)

Fino ad ora, abbiamo esaminato problemi di ricerca in cui gli stati erano visti come una scatola nera (**rappresentazione atomica**). Passiamo ora ad analizzare algoritmi che considerano gli stati come insieme di coppie attributo/valore (**rappresentazione fattorizzata**).

A.4.1 CSP

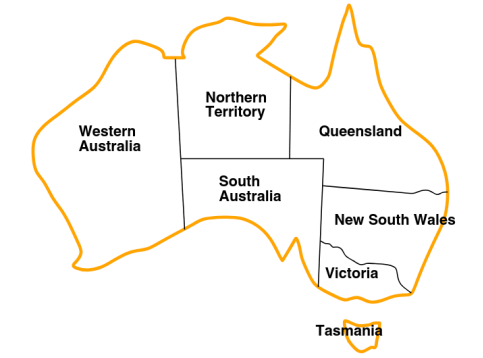
Def. 4: **CSP**

Un **constraint satisfaction problem** è definito da:

- un insieme di **variabili**: $X_1, \ldots, X_n$
- un insieme di **domini** (valori possibili per le variabili): $D_1, \ldots, D_n$
- un insieme di **vincoli** (su sottoinsiemi di variabili), ovvero dei requisiti che un assegnamento deve rispettare: $C_j = (Y_j, R_j)$
  - con $Y_j \subseteq X$ e R_j una relazione tra i domini delle variabili Y_j (combinazioni ammesse)

Esempio: colorazione di una mappa

Data la mappa dell'Australia e tre colori, colorare tutti gli stati di modo che stati adiacenti abbiano colore diverso.



- **Variabili:** `WA, NT, Q, NSW, V, SA, T`

- **Domini:** tutti pari a $\{r, g, b\}$
- **Vincoli:**
  - $\langle \{\texttt{WA}, \texttt{NT}\}, \overbrace{\{(r,g),(r,b),(g,r),(g,b),(b,r),(b,g)\}}^{\text{comb. ammesse}} \rangle \implies \texttt{WA} \neq \texttt{NT}$ *(si può scrivere così)*
  - $\langle \{\texttt{WA}, \texttt{SA}\}, \{(r,g),(r,b),(g,r),(g,b),(b,r),(b,g)\} \rangle \implies \texttt{WA} \neq \texttt{SA}$
  - ...

Def. 5: **Assegnamento consistente, soluzione**

Un **assegnamento** ad alcune variabili dai valori dei relativi domini ($\langle \ldots, X_i = v_i, X_j = v_j \rangle X_i, X_j \in X, v_i \in D_i, v_j \in D_j$) si dice **consistente** se *non viola alcun vincolo*, ovvero se per ogni vincolo C_j le cui variabili Y_j sono tutte coinvolte dall'assegnamento, la relazione R_j (le "coppie ammesse") contiene la tupla dei valori dati dall'assegnamento alle variabili.

Una **soluzione** è un assegnamento consistente per tutte le variabili X.

A.4.1.1 Vantaggi

I principali vantaggi dei CSP sono:

- formalizzazione naturale di molti problemi (ogni problema di ricerca in NP è formalizzabile come CSP)
- le tecniche di ricerca sono generali e non dipendono dal singolo problema: il problema viene modellato e risolto con solver general-purpose (le euristiche sono indipendenti dal dominio, dipendono da variabili e vincoli)
- la rappresentazione fattorizzata degli stati permette di alternare ragionamento e ricerca tramite la **propagazione dei vincoli** (una volta assegnato un valore ad una variabile si possono fare delle "inferenze" sui futuri assegnamenti)
- non è necessario calcolare per intero assegnamenti parziali non consistenti: nonappena un assegnamento viola i vincoli, tutte le clausole in cui è in AND (tutti gli assegnamenti che lo estendono) si possono scartare
- si può ragionare sulle cause delle violazioni dei vincoli

A.4.1.2 Tipi di vincoli

- **Vincoli unari:** su una sola variabile.
  esempio: $\texttt{SA} \neq g$
  - un vincolo unario si può gestire riformulando il problema (cambiando direttamente il dominio della variabile)
- **Vincoli binari:** su due variabili.
  esempio: $\texttt{WA} \neq \texttt{NT}$
- **Vincoli di globali:** su tre o più variabili
  esempio: alldiff(X, Y, Z, W, T)
  - ogni CSP si può riscrivere con vincoli al massimo binari (es: alldiff$(X, Y, Z) \equiv (X \neq Y) \wedge (Y \neq Z) \wedge (X \neq Z)$)

  - conviene comunque usare i vincoli globali, in quanto offrono una componente riusabile con librerie e algoritmi ottimizzati (quindi il solver va più veloce), ed rappresentano un approccio migliore per la modellazione del problema

A.4.1.3 Grafo dei vincoli

I vincoli si possono rappresentare tramite un **grafo** (che rispecchia quindi la struttura del problema) in cui

- i nodi rappresentano le variabili
- gli archi legano coppie di variabili che partecipano ad un vincolo

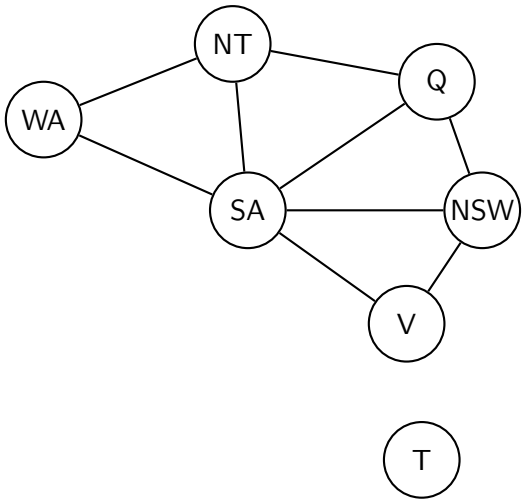


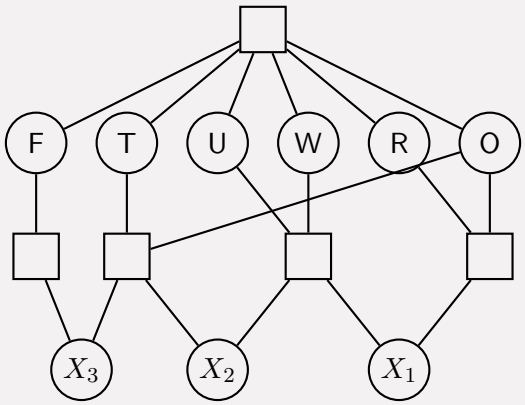
Figura A.4.1: Grafo dei vincoli per il problema della colorazione dell'Australia

Se i vincoli sono globali, il grafo dei vincoli diventa un **iper-grafo** (in cui gli archi sono sostituiti da *iper-archi*, formati da insiemi di nodi).

Un esempio di iper-grafo si ha nei problemi di cripto-aritmetica, dove bisogna sostituire delle lettere con delle cifre in modo che l'operazione aritmetica risulti corretta.

$$\begin{array}{cccccc} & X_3 & X_2 & X_1 & \\ & & T & W & O \\ + & & T & W & O \\ \hline & F & O & U & R \end{array}$$

- **Variabili:** T, W, O, F, U, R e i riporti X_1, X_2, X_3.
- **Domini:** per ogni variabile il dominio è $\{0, 1, 2, \ldots, 9\}$.
- **Vincoli:**
  - alldifff(T, W, O, F, U, R) (lettere diverse devono corrispondere a cifre diverse).
  - $O + O = R + 10 \cdot X_1$ (vincolo per la prima colonna, che considera il riporto).
  - $X_1 + W + W = U + 10 \cdot X_2$ (vincolo per la seconda colonna).
  - etc



Iper-grafo per il problema `TWO + TWO = FOUR`.
I quadrati rappresentano gli iper-archi (vincoli) che legano insiemi di variabili (cerchi).
Usare l'iper-grafo permette di mantenere la visione d'insieme su vincoli unici complessi senza doverli frammentare in archi binari

A.4.1.4 Risolvere un CSP: Ricerca nello spazio degli stati

Ogni problema di soddisfacimento di vincoli può essere affrontato formulandolo come un classico problema di ricerca. La formulazione base è **identica per tutti i CSP**:

- **Stati:** sono gli assegnamenti **consistenti** (anche parziali).
- **Successori:** si ottengono aggiungendo una **nuova variabile** assegnata ad un valore che non confligge con i vincoli.
  - *Nota:* non conviene definire i successori tramite l'operazione di "modifica di un assegnamento già fatto", perché questo ci porterebbe ad esplorare inutilmente stati che esistono già in un altro ramo dell'albero di ricerca.
- **Test obiettivo:** l'assegnamento corrente è completo?

Poiché l'albero di ricerca ha una profondità limitata e sappiamo per certo che una soluzione si troverà esattamente a profondità n (dove n è il numero totale di variabili), l'approccio ideale è usare una **ricerca in profondità (DFS)**.

A.4.1.4.1 Approccio naive e commutatività

Se applicassimo una DFS cieca su domini di taglia d, esploreremmo un numero di nodi foglia pari a $n! \cdot d^n$ (ad esempio, con 10 variabili e domini di 10 valori, avremmo $3.6 \cdot 10^{16}$ foglie).

Questo fattoriale $n!$ è dovuto al fatto che l'algoritmo, in questo modo, arriva alla stessa identica foglia (stessa combinazione finale) percorrendo cammini diversi (cambiando solo l'ordine in cui ha assegnato le variabili).

In realtà, possiamo fare molto meglio sfruttando una proprietà fondamentale dei CSP: la **commutatività**. L'ordine con cui si estendono gli assegnamenti è totalmente ininfluente ai fini della consistenza.

Pertanto:

(1) in ogni livello dell'albero possiamo decidere di considerare **una sola variabile**. Questo elimina completamente il fattore $n!$ dalla formula.

(2) la ricerca in profondità può **tornare indietro (backtrack)** non appena si accorge che un assegnamento parziale diventa inconsistente, potando fin da subito enormi sotto-alberi morti.

Algoritmo di Backtracking

Il **backtracking** è la versione base per la risoluzione dei CSP.

A differenza di altri algoritmi di ricerca, *non rappresenta in modo esplicito la frontiera* in memoria e lavora *modificando il nodo corrente* anziché generare materialmente i nodi figli. Inoltre, non attraversa mai stati non consistenti.

function `backtracking_ric`(ass, csp): soluzione $\vee$ fallimento
  if ass è completo **then return** ass;
  $var \leftarrow$ seleziona una variabile che non occorre in ass;
  foreach $val \in$ dominio di var **do**
    if val è consistente con ass dati i vincoli **then**
      aggiungi $\{var = val\}$ ad ass;
      $risultato \leftarrow$ `backtracking_ric`(ass, csp);
      if $risultato \neq fallimento$ **then return** risultato; // è una sol
      else rimuovi $\{var = val\}$ da ass; // fallimento: torno su e disfo
  return fallimento;

L'algoritmo base funziona, ma si può rendere molto più efficiente agendo su tre direzioni:

- **propagazione dei vincoli:** aggiungere ad ogni passo un'attività di inferenza logica per ridurre i domini delle variabili non ancora assegnate
- scelta, ad ogni passo, della **variabile** da assegnare (quale variabile mi conviene valutare per prima?)
- scelta, ad ogni passo, dell'**ordine dei valori** (quale valore del dominio provo per primo?)

A.4.1.5 Consistenza locale

Per ridurre lo spazio dei valori locali senza perdere soluzioni, si può **forzare la consistenza** locale del CSP. Questo si può fare sia come preprocessamento prima dell'inizio della ricerca, e sia durante la ricerca stessa.

Esistono diversi "gradi" di consistenza.

Def. 6: **Node-consistency**

Una variabile è **node-consistent** se tutti i valori del suo dominio soddisfano i suoi vincoli unari.

- i vincoli unari si possono, come già menzionato, eliminare prima della ricerca rendendo ogni variabile node-consistent

Def. 7: **Arc-consistency**

Una variabile X si dice **arc-consistent** se, per ogni vincolo binario C che coinvolge X e un'altra variabile Y, si ha che, per ogni valore $v_X \in D_X$ esiste un valore $v_Y \in D_Y$ che soddisfa C.

Esempio Se si ha $Y = X^2$ e $D_X = D_Y = \{0, \dots, 9\}$, per rendere X arc-consistent, si riduce D_X a $\{0, 1, 2, 3\}$ (gli altri valori non potranno produrre un quadrato ≤ 9); per rendere Y arc-consistent, si riduce D_Y a $\{0, 1, 4, 9\}$ (gli unici quadrati).

Def. 8: **Generalised arc consistency (GAC)**

Una variabile X è **generalised-arc-consistent** (GAC) se per ogni vincolo $C(X, Y_1, \dots, Y_k)$ che coinvolge X e altre variabili $Y_1, \dots, Y_k$, si ha che per ogni valore $v_X \in D_X$ esistono $v_{Y_1}, \dots, v_{Y_k}$ che soddisfano C.

A.4.1.5.1 Algoritmo GAC-3

L'algoritmo GAC-3 funziona solo su domini finiti, ma ne esistono estensioni che lavorano su domini infiniti. GAC-3 impone GAC *prima* di avviare la ricerca.

Generalised arc-consistency: GAC-3

function `GAC-3`(csp): false (trovata inconsistenza) o true
  input csp, un CSP con componenti (X, D, C)
  $Q \leftarrow \{(X_i, c) \mid c(\dots, X_i, \dots) \in C\}$;
  // (Q deve essere senza duplicati ma ordinato)
  while Q non è vuoto **do**
    $(X_i, c) \leftarrow$ estrai un elemento da Q;
    if `rimuovi_valori_inconsistenti`(csp, X_i, c) **then**
      if D_i è vuoto **then return** false;
      foreach $X_k \neq X_i$ e vincolo $c'(\dots, X_i, \dots, X_k, \dots) \neq c$ **do**
        inserisci (X_k, c') in Q;
        // devo ri-aggiungere i vincoli con X anche se già visti
        // perche il suo dominio è appena cambiato
  return true;

function `rimuovi_valori_inconsistenti`(csp, X_i, c): bool
  output: true se viene rimosso un valore, false altrimenti
  $Y_1, \dots, Y_m \leftarrow$ variabili coinvolte in c eccetto X_i;
  $rimosso \leftarrow$ false;
  foreach $v_i \in D_i$ **do**
    if nessuna m-pla di valori $(Y_1 = v_1, \dots, Y_m = v_m) \in D_{Y_1} \times \dots \times D_{Y_m}$ è
      tale che $(X_i = v_{X_i}, Y_1 = v_1, \dots, Y_m = v_m)$ soddisfa c **then**
      rimuovi v_i da D_i;
      $rimosso \leftarrow$ true;
  return rimosso;

Def. 9: **Path-consistency (consistenza di cammino)**

Un insieme di due variabili $\{X_i, X_j\}$ è **path-consistent** rispetto a una terza variabile X_m se, per ogni assegnamento $\{X_i = v_i, X_j = v_j\}$ consistente con i vincoli su $\{X_i, X_j\}$, esiste un assegnamento $X_m = v_m$ tale che:

- (v_i, v_m) soddisfa i vincoli su $\{X_i, X_m\}$
- (v_m, v_j) soddisfa i vincoli su $\{X_m, X_j\}$

Mentre la *arc-consistency* agisce riducendo i domini delle singole variabili, la *path-consistency* agisce sui vincoli binari. Se per una coppia di valori (v_i, v_j) non esiste un valore "intermediario" v_m che soddisfi il triangolo, la coppia viene rimossa dalla relazione ammissibile tra X_i e X_j. Questo processo permette di esplicitare vincoli impliciti e restringere ulteriormente lo spazio di ricerca.

Def. 10: **k-consistency**

Un CSP è **k-consistent** se, per ogni insieme di $k-1$ variabili e ogni loro assegnamento consistente, è possibile assegnare un valore consistente ad ogni k-esima variabile.

- **1-consistent:** node-consistency
- **2-consistent:** (generalised) arc-consistency
- **3-consistent:** path-consistency

Un CSP si dice **strong k-consistent** se è i-consistent per ogni $i \leq k$. Se un CSP con n variabili è strong n-consistent, possiamo risolverlo in tempo polinomiale. Tuttavia, forzare la strong n-consistency richiede tempo esponenziale.

N.B.: imporre tutte le consistenze equivale di fatto a risolvere il problema. Nella pratica, conviene fermarsi a forzare la GAC, non necessariamente spingendosi fino alla path-consistency.

A.4.1.6 Backtracking + propagazione

L'algoritmo di backtracking può essere potenziato con l'aggiunta della propagazione dei vincoli ad ogni assegnamento.

```
...
    if val è consistente con ass dati i vincoli then
        aggiungi {var = valore} ad ass;
        inferenze ← propagazione(csp, var, val); // es. rende le altre var GAC rispetto a var
        if inferenze ≠ fallimento then
            aggiungi inferenze ad ass;
            risultato ← backtracking_prop_ric(ass, csp);
...
```

Effettuare queste inferenze permette un pruning dei sottoalberi.

La funzione `propagazione()` può essere implementata in vari modi:

- **Forward checking (FC):** ogni volta che si aggiunge l'assegnamento $X_i = v$, si considerano tutti i vincoli che hanno *una sola* variabile non assegnata Y e si rimuovono dal dominio di Y i valori non consistenti con $X_i = v$.
- **Maintaining Arc Consistency (MAC):** ogni volta che si assegna $X_i = v$, si richiama GAC-3 (è sufficiente far partire l'algoritmo dai soli vincoli a cui X_i partecipa con variabili non ancora assegnate)

A.4.1.7 Euristiche per il Backtracking

L'ordine in cui si esplorano variabili e valori cambia drasticamente le prestazioni della ricerca.

A.4.1.7.1 Scelta della variabile da assegnare

- **Minimum Remaining Values (MRV):** ad ogni passo, sceglie la variabile con il dominio attivo più piccolo (detta anche euristica *fail-first*, perché cerca di far fallire subito i rami ciechi riducendo il fattore di ramificazione).
- **Max-degree:** ad ogni passo sceglie la variabile coinvolta nel maggior numero di vincoli. Di solito si usa MRV come euristica principale e Max-degree per rompere gli eventuali pareggi. (MRV da sola non è d'aiuto al primissimo livello, dove i domini sono tutti pieni).

A.4.1.7.2 Scelta dell'ordinamento dei valori

Una volta scelta la variabile, bisogna decidere quale valore provare per primo.

- **Valore meno vincolante (Least constraining value):** assegna il valore che lascia più libertà di scelta alle variabili adiacenti nel grafo dei vincoli.
- *Nota:* mentre per le variabili si usa un approccio *fail-first*, per i valori si usa un approccio *fail-last*: dato che cerchiamo una sola soluzione, ha senso esplorare per primi i valori più "probabili" che minimizzano i conflitti.

> Simmetrie e ottimizzazioni
>
> In molti problemi, alcune scelte sono equivalenti. La simmetria rappresenta una relazione di equivalenza (o un gruppo).
>
> Nel problema della mappa d'Australia, ad esempio, se il primo assegnamento $\texttt{SA} = r$ fallisce, sappiamo già che falliranno anche $\texttt{SA} = g$ e $\texttt{SA} = b$ per simmetria. Si può fare **symmetry breaking**: si sceglie un rappresentante per ogni classe di equivalenza in modo da restringere le soluzioni possibili ed eliminare interi rami di branching simmetrici.
>
> - Il *symmetry breaking* aiuta molto quando il problema è UNSAT (insoddisfacibile, perché evita di esplorare lo spazio simmetrico per dimostrarlo), ma aiuta meno quando è SAT.
> - Scegliere un rappresentante per le simmetrie potrebbe però "rompere" l'andamento di alcune euristiche. I solver avanzati permettono di calcolare le scelte "during search" a seconda dell'euristica desiderata, anche se questo comporta costi computazionali maggiori.

A.4.1.8 Backjumping (cenni)

Il backtracking standard fa marcia indietro cronologicamente: se fallisce, torna all'ultima variabile assegnata. Ma spesso non è questa ad essere la causa del fallimento.

Il **Conflict-driven backjumping** salta direttamente all'"ultima" variabile (la più in basso nell'albero) colpevole del fallimento, ignorando i livelli intermedi non correlati.

Questa tecnica si abbina al **no-good learning**:

- quando si arriva a un caso di fallimento, si apprende che quello specifico sotto-assegnamento "non è buono" (*no-good*) e non potrà mai essere completato.
- poiché lo stesso sotto-assegnamento potrebbe ripresentarsi in un'altra area dell'albero di ricerca,

si aggiungono dinamicamente nuovi vincoli durante la ricerca per ricordare il conflitto.

- per non esaurire la memoria, questi vincoli appresi vengono spesso "swappati" (gestiti come una cache, ad esempio con politiche LFU - Least Frequently Used).

A.4.1.9 CSP e Ricerca Locale

Gli algoritmi di ricerca locale possono essere molto utili per risolvere i CSP.

- **Stati:** si parte da un assegnamento iniziale *completo* (anche casuale) in cui sono ammesse violazioni dei vincoli.
- **Euristica Min-conflicts:** ad ogni mossa, si sceglie casualmente una variabile che si trova attualmente in conflitto, e le si riassegna il valore che **minimizza il numero di vincoli violati**.

È fondamentale avere una metrica di violazione dei vincoli ("quanto" un assegnamento sia buono); serve calcolare esattamente il grado di violazione di un vincolo per orientare la ricerca, altrimenti l'algoritmo si bloccherebbe in plateau.

Vantaggi prestazionali Se si esclude il tempo per generare l'assegnamento casuale iniziale, *min-conflicts* risolve problemi complessi (come le n-regine) in un tempo quasi indipendente da n. I solver moderni impiegano pochissimo tempo a risolvere istanze casuali di CSP, tranne che in una specifica "zona critica" definita dal rapporto $R = \frac{\text{numero di vincoli}}{\text{numero di variabili}}$. Al di fuori di questo ristretto intervallo, il problema è facilmente risolvibile o facilmente riconoscibile come insoddisfacibile; solo nella fascia "in mezzo" la probabilità di incappare in un problema davvero "difficile" si aggira intorno al 50%.

A.4.2 Modellazione con MiniZinc

MiniZinc è un linguaggio di modellazione ad alto livello utilizzato per formulare problemi di ottimizzazione e di soddisfacimento di vincoli (CSP).

A.4.2.1 Parametri e Variabili di Decisione

In MiniZinc è fondamentale distinguere ciò che è noto a priori da ciò che il solver deve calcolare. Questa distinzione si traduce in due macro-categorie:

- **parametri** (dichiarati implicitamente o con la keyword `par`): rappresentano i dati di input del problema. Hanno un valore è fisso e immutabile durante la risoluzione.
- **variabili di decisione** (keyword `var`): rappresentano le incognite dello spazio degli stati. Il solver manipola questi valori all'interno dei loro domini per rispettare i vincoli.

```
int: n_prodotti = 5; /* parametro: valore fisso e noto */
var 1..100: quantita; /* variabile di decisione: dominio tra 1 e 100 */
```

> Tipi di dato e domini
>
> MiniZinc è fortemente tipizzato. Ogni variabile deve avere un tipo e, nel caso delle variabili di decisione, un dominio limitato per restringere lo spazio di ricerca.
>
> - **`int`** e **`float`**: numeri interi e a virgola mobile.
> - **`bool`**: valori di verità (`true` o `false`)
> - **`set of <tipo>`**: insiemi di valori, spesso usati per definire indici di iterazione (es. `set of int: Citta = 1..5;`).
> - **`array`**: collezioni n-dimensionali di elementi.

A.4.2.2 Vincoli (Constraints)

I vincoli definiscono le regole del mondo del problema, restringendo le assegnazioni valide. Si dichiarano con la parola chiave `constraint` seguita da un'espressione logica/matematica.

```
constraint quantita > 10;
constraint spesa_totale <= budget_massimo;
constraint x != y;
```

A.4.2.3 Risoluzione (Solve Item)

Ogni modello deve terminare indicando al solver il tipo di ricerca desiderata:

- `solve satisfy;`: cerca *una qualsiasi* assegnazione che rispetti tutti i vincoli
- `solve maximize <expr>;`: cerca l'assegnazione ammissibile che *massimizza* il valore dell'espressione fornita
- `solve minimize <expr>;`: cerca l'assegnazione ammissibile che *minimizza* l'espressione

Esempio: Colorazione di una mappa

Vogliamo assegnare a tre regioni (WA, NT, SA) uno tra due colori disponibili, garantendo che regioni confinanti abbiano colori diversi.

```
/* parametri: numero di colori */
int: nc = 2;

/* variabili di decisione: assegnazione dei colori (dominio 1..nc) */
var 1..nc: wa;
var 1..nc: nt;
var 1..nc: sa;

/* vincoli di confinamento */
constraint wa != nt;
constraint wa != sa;
constraint nt != sa;

/* obiettivo */
solve satisfy;

/* output formattato */
output ["WA=\(wa),␣NT=\(nt),␣SA=\(sa)\n"];
```

A.4.2.4 Iterazioni e Array

Nei problemi scalabili, dichiarare ogni vincolo singolarmente è impossibile. MiniZinc utilizza strutture ad array e **list comprehension** tramite il costrutto `forall`.

```
/* array di 5 variabili incognite */
array[1..5] of var 1..10: x;

/* applica un vincolo a tutti gli elementi: tutti devono essere pari */
constraint forall(i in 1..5)(x[i] mod 2 == 0);

/* vincolo relazionale: elementi adiacenti devono essere diversi */
constraint forall(i in 1..4)(x[i] != x[i+1]);
```