

Parleremo di :

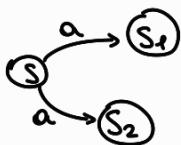
- Problemi di ricerca di una soluzione in uno spazio degli stati
- Problemi di soddisfacimento vincoli
- Problemi con avversario
- Ragionamento logico

Concetti base :

- STATO = rappresentazione di determinate condizioni che possono valere nel mondo di interesse
 - rappresentato da un'etichetta
- AZIONE = meccanismo di transizione da uno stato ad un altro

DETERMINISTICA se $\forall s \exists! s' \text{ t.c. } s \xrightarrow{a} s'$

NON DETERMINISTICA

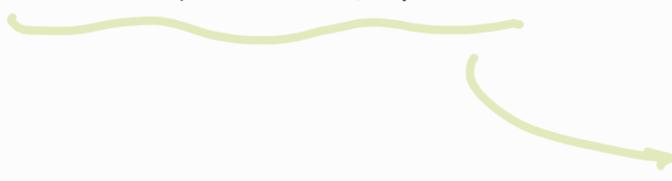


→ richiede approccio probabilistico
(non ne parleremo)

PROBLEMI DI RICERCA IN UNO SPAZIO DEGLI STATI

Il problema in se' verrà descritto come una quadrupla:

$\langle s_i, f_s, t_o, f_c \rangle$

- 
- s_i : stato iniziale
 - f_s : funzione successore
 - t_o : test obiettivo
 - f_c : funzione costo

SOLUZIONE = percorso che permette di raggiungere un nodo target partendo dallo stato iniziale

SOLUZIONE OTTIMA = soluzione di costo minimo

Un esempio può essere quello della ricerca di un percorso in una mappa (\rightarrow NAVIGATORE) dove:

stati del problema = località

costo = km oppure sicurezza oppure bellezza-paesaggio (etc...)

Considereremo principalmente dei **toy problem**

Come struttura di ricerca useremo **alberi** (principalmente) e **grafi** di ricerca

Dividiamo gli algoritmi in

blind

informati

Stato \neq nodo (dell'albero/graf di ricerca)

→ Nodo = Stato + altre informazioni

Archi : portano da un nodo ad un altro

↳ vi è legato il costo di transizione

FUNCTION RICERCA-ALBERO (Problema)

Returns Soluzione || Fallimento

Frontiera $\leftarrow \{ \text{stato iniziale} \}$

loop do

if (frontiera vuota) return Fallimento

scegli nodo \in Frontiera

Rimuovi nodo da frontiera

if (nodo contiene Stato goal)

return Soluzione

espandi nodo

Aggiungi successore a frontiera

↳ schema generale

La valutazione di un algoritmo viene fatta misurando 4 parametri

- completezza
- ottimalità
- complessità temporale
- complessità spaziale

STRATEGIE BLIND

RICERCA IN AMPIEZZA

- sfrutta code FIFO per implementare la frontiera

Completo sse b finito

b = branching factor
= numero medio dei figli
dei nodi

Ottimalità non garantita (in generale)

↳ si sse f_c monotona decrescente della profondità

Complessità $O(b^{d+1})$

d = profondità del goal

RICERCA A COSTO UNIFORME

- variante della ricerca in ampiezza
- disomogeneità nei costi \Rightarrow costo percorso \neq numero passi
- obiettivo: trovare soluzione ottima
- La frontiera è gestita con una coda di priorità ordinata in modo crescente di costo.
- Non fermiamo necessariamente la ricerca quando troviamo un nodo goal, perché cerchiamo una soluzione ottima finché la frontiera non contiene il target in testa.

Completo sse tutti i costi sono > 0

Ottimalità garantita sse completo

Complessità $O(b^{d+L^{\frac{c^*}{\epsilon}}})$

c^* = costo minimo della soluzione ottima

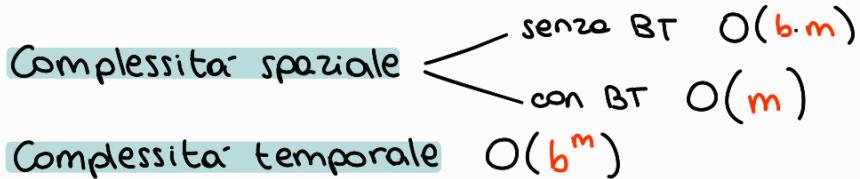
ϵ = costo minimo delle azioni

RICERCA IN PROFONDITÀ

- Usa una pila (LIFO) / Stack per implementare la frontiera
- Con Backtracking: costruisco uno dei possibili successori del nodo via via fino alla foglia.
Se non è goal, risalgo per esplorare alternative
- Senza Backtracking: produco tutti i possibili successori del nodo estratto dallo stack.

Completo se la profondità m è finita

Ottimalità non garantita



RICERCA A PROFONDITÀ LIMITATA

↳ volta ad aumentare la completezza della ricerca in profondità:

- Si impone un limite l per il quale, oltre tale livello, ogni nodo è trattato come una foglia. Se non è goal \Rightarrow backtracking
- l rende l'albero finito

PROBLEMA: se il goal si trova oltre il limite?



ITERATIVE DEEPENING DEPTH FIRST SEARCH

FUNCTION RICERCA-APR-ITER (Problema)

RETURNS Soluzione || Fallimento

```
For profondità ← 0 to ∞ do
    risultato ← RICERCA-PROFONDITÀ-LIMITATA(Problema, profondità)
    if (risultato ≠ taglio) =limite
        return risultato
    l ~ d
```

- Ad ogni iterazione si genera l'albero di ricerca da zero.
- Occupa poca memoria

Completo sse b finito

Ottimalità?

Complessità temporale $O(b^d)$

Complessità spaziale $O(d)$

RICERCA BIDIREZIONALE

- Servono 2 processori
- 2 ricerche in parallelo : Forward + Backward (conoscendo i possibili goal)
↳ finisce se le ricerche si incontrano

Difficoltà :

- 1) Ci possono essere tanti stati goal → posso costruire una falsa radice che incorpora tutti i goal e dalle quale far partire la ricerca Backward
- 2) Inverte effetto e precondizioni → non sempre semplice
- 3) Nel mondo ideale, le due ricerche si incontrano a metà strada;
Nel mondo reale, potrebbero incontrarsi a fine esplorazione, senza guadagnare tempo e sprecando così due processori.

STRATEGIE INFORMATE

$\langle s_i, f_s, t_o, f_c \rangle + \text{conoscenza predittiva}$ (funzione euristica $f(n)$) $\forall n$)

funzione euristica : dato uno stato, calcola una stima del costo minimo per raggiungere un nodo goal

esempio : per un navigatore in cui $\text{costo} = \sum \text{costi azioni}$
dati dai km su strada
euristica = distanza in linea d'aria.

→ guida a scegliere il percorso più promettente, che mi permetterà di costruire una soluzione ottima

Nodo preferito di n : soddisfa t_o e t.c. $f(n) = \min_{\text{nodi goal}} \{f(n, G_i)\}$

RICERCA GREEDY

- esplora un albero di ricerca
- come ricerca a costo uniforme ma la priority queue ordina in base all'euristica.

A*

- cerca una soluzione ottima
- utilizza una funzione di valutazione $f(n) = \underbrace{g(n) + h(n)}$
// calcolata tramite f_c
costo minimo dei cammini esplorati $s_i \rightarrow n$
- Se ho un albero :
 $g(n)$, a differenza di $h(n)$, è un calcolo esatto ed è la somma delle funzioni costo nel cammino fino ad n
- Se ho un grafo (più strade per arrivare ad n) :
 $g(n) = \min_{i \in \text{predecessori}} \{g(n_i)\} \rightarrow$ diventa una stima

Vedi problema
della Romania

- è una sorta di ricerca in ampiezza in cui esploro per livelli di $f(n)$

Osservazione:

- se $g(n) = 0 \quad \forall n \Rightarrow A^* \rightarrow \text{Greedy}$
- se $h(n) = 0 \quad \forall n \Rightarrow A^* \rightarrow \text{ricerca a costo uniforme}$
- se $g(n) = 1 \quad \forall n \Rightarrow A^* \rightarrow \text{ricerca in ampiezza}$

A^* ottimamente efficiente :

Non esiste nessun altro algoritmo che garantisce di restituire la soluzione ottima esplorando un numero di nodi inferiore ad A^* .

$$\bullet \quad f^*(n) = \underline{g^*(n)} + \underline{h^*(n)}$$

costo reale min per andare da s_i a n

costo reale min per andare da n a un goal

VALORE REALE

Ottimalità : dipende dall'euristica scelta

- su alberi sse h ammissibile
- su grafi sse h consistente / monotona

• h AMMISSIBILE sse $\forall n \quad h(n) \leq h^*(n)$

• h MONOTONA / CONSISTENTE sse vale diseguaglianza triangolare
ovvero sse $\forall n \quad h(n) \leq h(n') + c(n, a, n')$

Osservazione: h consistente $\Rightarrow h$ ammissibile

Sono da sapere !!
(x orale)

Si dimostra che quando

- h ammissibile
- tutti costi restituiti da f_c sono $c > 0$

 allora A^* termina (se esploriamo struttura ad albero)

Attenzione: A^* non è efficiente in termini di spazio

RUOLO E QUALITÀ DELL'EURISTICA

- Alcune euristiche sono "più informative" di altre, e quindi permettono di raggiungere la soluzione ottima in maniera più efficiente

Euristica basata sulla "distanza di Manhattan":

conta quante mosse sono necessarie per passare dalla configurazione corrente alla configurazione desiderata.

→ è ammissibile

La valutazione delle euristiche tipicamente richiede un confronto fra euristiche diverse;

Può essere fatta in forma teorica (con dimostrazione matematica) oppure in forma sperimentale.

- (1) **FORMA TEORICA** : dimostrazione di quale fra le euristiche considerate è **DOMINANTE / PIÙ INFORMATIVA**

Siano h_1, h_2 ammissibili, allora

h_1 è dominante su h_2 sse $\forall n \quad h_2(n) \leq h_1(n) \leq h^*(n)$

ovvero h_1 approssima meglio il costo minimo reale

- (2) **VALUTAZIONE SPERIMENTALE** : bisogna generare un insieme di casi significativi facendo tante esecuzioni di h_1 e h_2 e tenendo conto del numero di nodi (=ampiezza degli alberi) in entrambi i casi a parità di profondità.

corrisponde al branching factor di un albero uniforme di profondità pari a quella dell'albero originario contenente

$n+1$ nodi

$$b^* \approx \sqrt[n]{n}$$

$$b^* \approx 1 \Rightarrow \text{MIGLIORE}$$

L'euristica migliore genera un albero di minori dimensioni.

Si considerano delle medie di tutti i casi in cui il goal si trova alla stessa profondità

Si utilizza il **branching factor effettivo b^*** , che indica quanto si allargano mediamente gli alberi che lavorano su certi casi.

Se ho h_1, h_2, \dots, h_m ammissibili ma nessuna è dominante sulle altre

prendo $\forall n \quad h(n) = \max(h_1(n), h_2(n), \dots, h_m(n))$

RBFS - Recursive Best-First Search/Strategy

- Ottimizzazione di A* nell'uso della memoria
- Variante di ricerca in profondità senza backtracking
- Utilizzo UPPERBOUND F e continuo a scendere in profondità finché non si supera F (ovvero interrompo la discesa quando $f > F$ e torno al nodo per cui era stato calcolato F).
- F varia nel tempo, diventa più preciso in base all'esplorazione già fatta
↳ ≠ ricerca a profondità limitata
- Per ogni nodo viene mantenuta un'informazione di costo stimato del 2° migliore e si continua ad esplorare il sottoalbero finché il 2° migliore non diventa il migliore, quindi si interrompe l'esplorazione e si salta all'altro ramo

RBFS ha 3 argomenti :

nodo N

$F(N)$

$B_{upperbound} \leftarrow \infty$

↳ calcolato con F (nodi fratelli)

$RBFS(N, F(N), B)$

if ($f(N) > B$) return $f(N)$

if ($N = GOAL$) TERMINA CON SUCCESSO

if (N non ha successori) return ∞ (percorso da evitare)

For each child N_i di N

[if ($f(N) < F(N)$)

$F[i] := \max(F(N), f(N_i))$

else $F[i] := f(N_i)$

SORT N_i AND $F[i]$ in ordine crescente di $F[i]$

if (ONLY ONE CHILD) $F[2] := \infty$

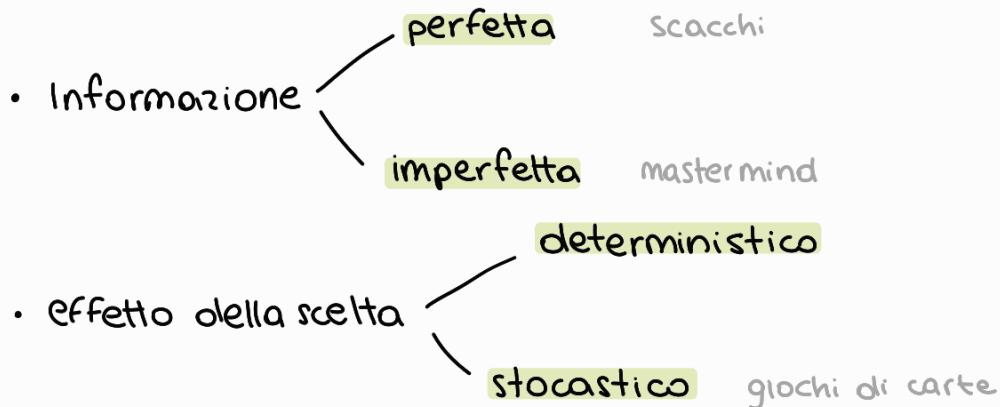
while ($F[1] \leq B \wedge F[1] < \infty$)

$F[1] := RBFS(N_1, F[1], \min(B, F[2]))$

 INSERT N_1 AND $F[1]$ in SORTED ORDER

STRATEGIE DI RICERCA CON AVVERSARIO

- Ambiente multi-agente e competitivo
- Ogni agente è guidato dai propri obiettivi
- Agenti diversi hanno obiettivi conflittuali
- Detti GIOCHI



Ci focalizziamo sui giochi a somma zero

(guadagno di uno \equiv perdita dell'altro)

- Il mondo della finanza è visto come un gioco a somma zero
(avversario = mercato)
- Tris = a turni, informazione perfetta , deterministico

Teoria delle decisioni

APPROCCIO MAXIMAX (ottimistico)

Guarda i payoff più alti per ogni possibile scelta e fa la scelta che promette di più in assoluto \rightarrow scommette sul fatto che si verifichi sempre l'evento migliore possibile.

APPROCCIO MAXIMIN (pessimistico / conservativo)

Guarda le perdite maggiori legate a ciascuna scelta e poi esegue l'azione che minimizza le perdite.

APPROCCIO MINIMAX REGRET

Si calcolano i massimi regret per ogni scelta alternativa e poi si seleziona l'alternativa che porta al regret minimo \rightarrow cerca di minimizzare il pentimento

$$\cdot \text{best regret} = \text{best payoff} - \text{real payoff}$$

ALGORITMO MINIMAX

- Suggerisce la prossima mossa da fare (teoria delle decisioni)
- MAX : giocatore che muove per primo
- MIN : giocatore avversario → le sue mosse non sono controllabili da MAX
- simula che MIN sia un giocatore perfetto (atteggiamento pessimista)

Partendo da uno stato iniziale è possibile sviluppare un albero di possibili evoluzioni (albero di gioco), applicando le azioni eseguibili e calcolando così gli stati successori.

$$\text{minimax}(n) = \begin{cases} \text{utilità}(n) & \text{se } n \text{ è terminale} \\ \max_{s \in \text{succ}(n)} (\text{minimax}(s)) & \text{se } n \text{ è un nodo MAX} \\ \min_{s \in \text{succ}(n)} (\text{minimax}(s)) & \text{se } n \text{ è un nodo MIN} \end{cases}$$

- è un valore calcolato per ogni nodo.
- permette all'agente la scelta della mossa da eseguire.
- nei giochi a due, l'utilità per un giocatore è uguale all'utilità dell'altro ma con segno opposto
- Utilità(n) : funzione calcolata solo per le foglie
↳ dalla prospettiva di MAX

MINIMAX-DECISION (STATE) RETURN ACTION

$v \leftarrow \text{MAX-VALUE (STATE)}$

return ACTION in SUCCESSORS (STATE) with VALUE v

MAX-VALUE (STATE) RETURNS UTILITY-VALUE
if (TERMINAL-STATE (STATE))
return UTILITY (STATE)

$v \leftarrow -\infty$

FOR a, s in SUCCESSORS (STATE)
 $v \leftarrow \max(v, \text{MIN-VALUE}(s))$
return v

MIN-VALUE (STATE) RETURNS UTILITY-VALUE
if (TERMINAL-STATE (STATE))
return UTILITY (STATE)

$v \leftarrow +\infty$

FOR a, s in SUCCESSORS (STATE)
 $v \leftarrow \min(v, \text{MAX-VALUE}(s))$
return v

Completo se l'albero delle evoluzioni del gioco è finito

Ottimale se sia MAX che MIN giocano in maniera ottimale

complessità temporale: $O(b^m)$ → costoso

complessità spaziale: $O(m)$

$m = \text{profondità massima}$

MINIMAX CON ALPHA-BETA PRUNING

- Variante per migliorare la complessità temporale

α = massimo lower bound delle soluzioni possibili (controllato da MAX)

β = minimo upper bound delle soluzioni possibili (controllato da MIN)

- Si esplora un nodo N sse $v \in [\alpha, \beta]$ e $\alpha < \beta$

α = alternativa migliore per MAX tra radice e STATE

β = alternativa migliore per MIN

ALPHA-BETA-SEARCH (STATE)

$v \leftarrow \text{MAX-VALUE}(\text{STATE}, -\infty, +\infty)$

Return ACTION a in SUCCESSORS(STATE) WITH VALUE v

MAX-VALUE (STATE, α , β)

if (TERMINAL-STATE(STATE)) return UTILITY(STATE)

$v \leftarrow -\infty$

FOR a, s in SUCCESSORS(STATE)

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, \alpha, \beta))$

if ($v \geq \beta$) return v

$\alpha \leftarrow \text{MAX}(\alpha, v)$

Return v

MIN-VALUE (STATE, α , β)

if (TERMINAL-STATE(STATE)) return UTILITY(STATE)

$v \leftarrow +\infty$

FOR a, s in SUCCESSORS(STATE)

$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, \alpha, \beta))$

if ($v \leq \alpha$) return v

$\beta \leftarrow \text{MIN}(\beta, v)$

Return v

Complessità temporale nel caso migliore diventa $O(b^{m/2})$

Nel caso medio:
 $O(b^{\frac{3}{4}m})$

$\alpha\text{-}\beta$ PRUNING Si comporta meglio o peggio a seconda dell'identificazione delle KILLER MOVE.

Si sono sviluppate delle strategie, in aggiunta all'algoritmo di gioco, volte a lavorare solo sulle killer move.

Ad esempio ci sono delle sequenze di mosse che, trasposte (cambiando l'ordine), portano in ogni caso al successo → si mantengono delle Tabelle delle Trasposizioni per migliorare l'efficacia dell'algoritmo.

Nei giochi Real Time, in cui le prossime mosse deve essere proposte in un tempo limite, l'albero di gioco potrebbe essere così grande che in questo tempo limite non si riesce ad esplorarlo completamente.

Essendo che solo gli stati terminali hanno associati dei valori di utilità, è possibile che l'esplorazione venga interrotta prima di raggiungerli ⚠

⇒ Si utilizzano funzioni di valutazione che permettono di dirci qualcosa sui nodi intermedi

- apprendimento automatico / statistico / probabilistico
(associare nodi a "classi")
- conoscenza specifica sul gioco

MANCANZA DI QUIESCENZA DEGLI STATI

- Ci sono stati "più affidabili" = quiescenti
- altri sono non quiescenti → possibilità di vittoria / sconfitta elevata
ma con una mossa si può ribaltare la situazione

CONSTRAINT SATISFACTION PROBLEMS (CSP)

- Ricerca di una soluzione
- Ci interessiamo della natura degli stati, la cui informazione è esplicitata

Un CSP è definito in termini di

- Variabili X_1, X_2, \dots, X_n ognuna con un proprio dominio.
(Negli esempi che vedremo hanno tutte lo stesso dominio)
- Vincoli $C_1, C_2, \dots, C_m \rightarrow$ sono condizioni

Risolvere un CSP = trovare un **ASSEGNAZIONE COMPLETO** (attribuisce un valore ad ogni variabile) e **CONSISTENTE** (le scelte fatte soddisfano tutti i vincoli)

Contesti di applicabilità :

- Formare equipaggi
- Logistica
- Taglio dei materiali
- Allocazione risorse

Vedi problema dell'Australia

Possiamo vedere il CSP come un problema di ricerca in uno spazio degli stati, immaginando che ogni stato corrisponda ad un assegnamento (anche incompleto).

In tal caso, dobbiamo definire $\langle s_i, f_s, t_o, f_c \rangle$:

- $s_i = \{\}$ assegnamento vuoto
- $f_s : S \rightarrow S'$ se $s \equiv$ assegnamento incompleto
e $s' \equiv s \cup \{x_i = v_i\}$ dove $x_i \notin s$
- $t_o = \text{True}$ se s è completo e consistente
- $f_c ?$

N.B. Nella costruzione dell'albero CSP, l'ordine degli assegnamenti è irrilevante

I vincoli possono essere di varie arieta':

UNARI : 1 variabile $\text{ETA}^+ > 18$

BINARI : 2 variabili $WT \neq NT$

A TRE O PIÙ VARIABILI

Vedi esempio CRIPTO-ARITMETICA

Vincolo ALLODIFFERENT (x_1, x_2, \dots, x_k) $k \geq 2$

ogni variabile deve avere un valore \neq da quello di tutte le altre

Vedremo due approcci: 1) Forza bruta

2) Ricerca in profondità con backtracking

1) GENERATE-AND-TEST

- genera in maniera casuale un assegnamento completo
- Fa un test per verificare se l'assegnamento generato è soluzione

loop

→ NON INTELLIGENTE

2) Ricerca in profondità con backtracking (non informato)

Siano $n = \#$ variabili CSP

$d = \#$ valori di tali variabili

$$\Rightarrow \text{numero di foglie} = n! d^n$$

Io posso diminuire

PROPRIETÀ COMMUTATIVA DEI CSP:

l'ordine in cui vengono fatti gli assegnamenti è ininfluente sul raggiungimento di una soluzione

Per far sì che non vengano generate tutte le permutazioni di uno stesso assegnamento:

Fisso una variabile per ogni livello \Rightarrow numero foglie = d^n

BACKTRACKING-SEARCH (CSP) RETURNS Solution || Failure
 return RECURSIVE-BACKTRACKING ({}, CSP)

RECURSIVE-BACKTRACKING (Assegnam., CSP) RETURNS Solution || Failure
 if (Assegnam. is complete) return Assegnam.
 var \leftarrow SELECT-UNASSIGNED-VARIABLE (VARIABLES (CSP), Assegnam., CSP)
 For each VALUE in ORDER-DOMAIN-VALUES (var, Assegnam., CSP)
 if (VALUE consistent with Assegnam.)
 ADD $\{var = VALUE\}$ to Assegnam.
 RESULT \leftarrow RECURSIVE-BACKTRACKING (Assegnam., CSP)
 if (RESULT \neq Failure) return RESULT
 REMOVE $\{var = VALUE\}$ from Assegnam.
 return Failure

Non informato \Rightarrow non si ha memoria di un assegnamento che genera fallimento
 \Rightarrow THRASHING

Soluzione: uso di euristiche (\neq da quelle già viste)

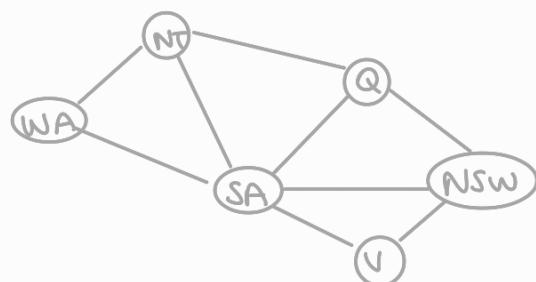
\hookrightarrow sono dei criteri di "buon senso"

Nell'algoritmo $var \leftarrow$ SCEGLI-VARIABILE (...)

non ancora assegnata

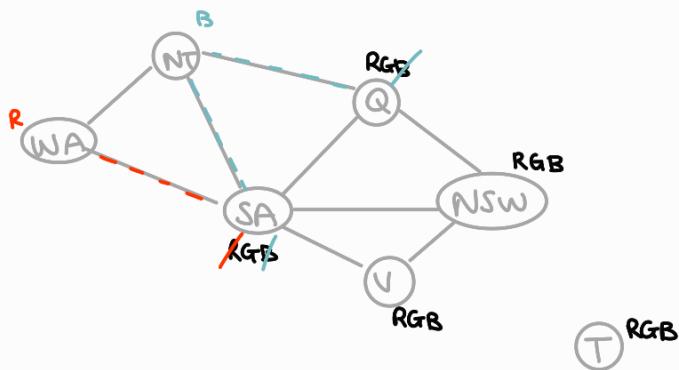
\rightarrow Quando i vincoli sono binari, si costruisce un GRAFO DEI VINCOLI i cui nodi corrispondono alle variabili e un arco tra due variabili corrisponde al vincolo che le collega

Nell'esempio dell'Australia:



Euristica Minimum Remaining Values

(o Fail-First)



Sceglie la variabile per cui rimane il minor numero di valori assegnabili consistenti con le scelte fatte fino a quel momento.

IDEA: Se le scelte fatte fino a questo momento non permettono di arrivare ad una soluzione, voglio saperlo il prima possibile.

Non funziona sempre! (scelta della prima variabile o parimerito)

Euristica di grado

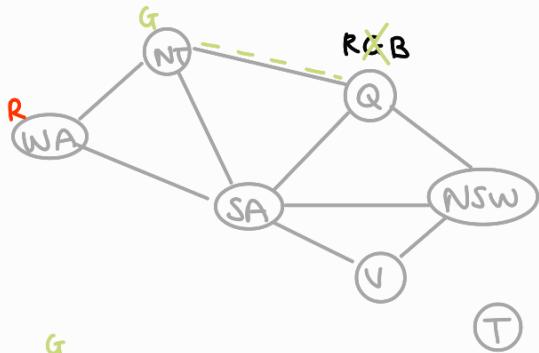
→ Preferire la variabile più vincolata (= coinvolta nel maggior numero di vincoli)

Euristica di Scelta del valore meno vincolante

Scelta del prossimo valore da considerare

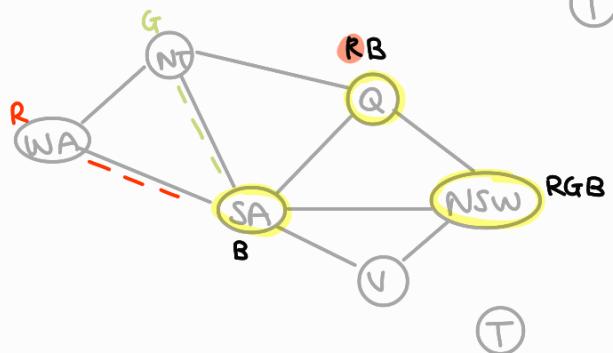
→ usata in combinazione con un'altra euristica

→ Preferire un valore che lascia lo spazio di scelta più ampio sulle variabili legate da vincolo a quella presa in considerazione



Supponiamo di aver assegnato $WA=R$ e $NT=G$ e dover scegliere un valore per Q .

→ Guardo qual è l'impatto delle possibili scelte sui territori confinanti (NSW, SA)



Se $Q=R$: impatto su SA : nessuno
impatto su NSW : no R

Se $Q=B$: impatto su SA : $SA=\{ \}$!
impatto su NSW : no B

⇒ Scelgo $Q=R$

Aggiungiamo all'algoritmo di Backtracking il meccanismo di **inferenza**

- Fare inferenza = applicare un **ragionamento** per costruire nuova conoscenza
- Scopo = **ridurre la dimensione del problema**

Quale inferenza è possibile applicare in un CSP?

Metodi di consistenza locale :

- Forward Checking
- Node consistency
- Arc consistency
- Path consistency

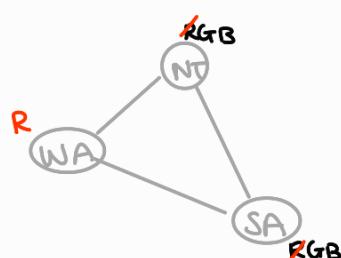
} K-consistency

Forward Checking

Consiste nel percorrere il grafo dei vincoli e si toglie dai domini degli adiacenti i valori non consistenti con la scelta fatta.

Supponiamo di avere

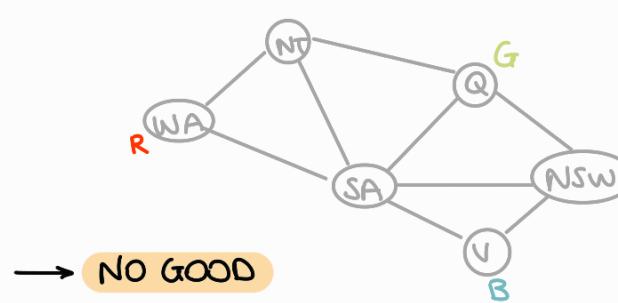
$$\boxed{WA = R} \rightarrow NT? \quad SA?$$



- Spesso utilizzato insieme a Minimum Remaining Value

ATTENZIONE : non sempre funziona bene !

WA	NT	Q	NSW	V	SA	T
RGB						
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB
R	B	G	R	B	RGB	RGB



→ NO GOOD

(T)

K-consistency

Node consistency \leadsto 1-consistency

Arc consistency \leadsto 2-consistency

Path consistency \leadsto 3-consistency

- 1) Un CSP è Node-consistent quando le sue variabili soddisfano i vincoli unari
- 2) Un CSP è Arc-consistent quando le variabili soddisfano i vincoli binari



$\forall u_x \in D_x \exists u_y \in D_y$

e u_x, u_y consistenti rispetto a c



$\forall u_y \in D_y \exists u_x \in D_x$

e u_x, u_y consistenti rispetto a c

Attenzione: deve essere soddisfatto in entrambi i versi !

Prendiamo WA e NT



Abbiamo la consistenza



NON CONSISTENTE

Algoritmi **AC-3** \rightarrow riducono il dominio delle variabili

- in alcuni casi trovano la soluzione
- se il dominio di qualche variabile diventa vuoto, incompleto non ha soluzione

AC3 (csp) RETURNS CSP

QUEUE : code di archi

while (QUEUE NOT EMPTY) DO

$(x_i, x_j) \leftarrow \text{REMOVE-FIRST(QUEUE)}$

if (REMOVE-INCONSISTENT-VAEVES(x_i, x_j))

FOREACH $x_k \in \text{Neighbours}(x_i)$

ADD(x_k, x_i) TO QUEUE

REMOVE-INCONSISTENT-VALUES(x_i, x_j) RETURNS TRUE IFF WE REMOVE A, FALSE OTHERWISE

REMOVED \leftarrow FALSE

FOREACH $x \in \text{DOMAIN}[x_i]$ DO

 IF ($\exists y \in \text{DOMAIN}[x_j]$ CONSISTENT WITH x)

 DELETE x FROM DOMAIN [x_i]

 REMOVED \leftarrow TRUE

RETURN REMOVED

- 3) Si dice che la coppia di variabili $\{x_1, x_2\}$ è path-consistent rispetto a x_3 quando:

\forall assegnamento consistente di x_1 e x_2

\exists assegnamento possibile di x_3 che soddisfa i vincoli esistenti fra $\{x_1, x_3\}$ e $\{x_2, x_3\}$

- Un CSP è k -consistent quando:

\forall insieme costituito da $(k-1)$ delle sue variabili

e \forall loro assegnamento consistente

\exists un valore per la k -esima variabile consistente

- Un CSP è Fortemente k -consistente quando è 1-2-3-consistent

Se CSP Fortemente k -consistent \Rightarrow risolvibile in tempo lineare

Osservazione: l'algoritmo di ricerca in profondità con backtracking lavora facendo back-propagation cronologico
 \Rightarrow si fa sempre la revisione della scelta più recente.

E' possibile lavorare su forme di back-propagation per cui è possibile saltare a scelte che non sono le più recenti

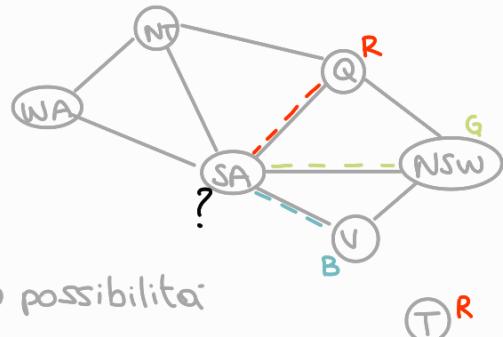
Variante del Backtracking : **Backjumping**

permette di saltare ad una scelta precedente trascurando una serie di scelte intermedie che non sono significative.

\Rightarrow + efficienza

Abbiamo assegnato (in quest'ordine)
 $\{ Q=R, NSW=G, V=B, T=R \}$
 \rightarrow consistente

Supponiamo di voler assegnare SA \rightarrow Non ci sono possibilità



STACK ESECUZIONE:
 (con backtracking)



T ininfluente (no vincoli con SA)

\rightarrow esploro tutto lo stack
 \rightarrow perdo molto tempo

Per comprendere il backjumping ci servono alcune nozioni:

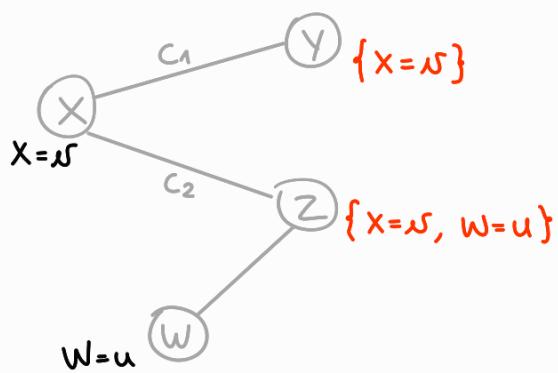
- **CONFFLICT SET**: Sia A un assegnamento parziale, consistente per un CSP;
 Sia X una variabile del CSP non ancora assegnata.

$$A = \{x_1 = v_1, x_2 = v_2, \dots, x_k = v_k\} \quad \forall x_i \in [1, k] \quad x \neq x_i$$

A è un CONFFLICT SET di X se $\forall v_i$ di X l'assegnamento $A \cup \{x = v_i\}$ è inconsistente

Come si costruisce un conflict set :

- Sfrutta il forward checking → il c.s. contiene, per ogni variabile, gli assegnamenti che creano conflitto con le scelte fatte fin'ora.
(ovvero tutti i valori rimossi dal dominio grazie al F.C.)



⇒ Per ogni variabile e' associata una
annotazione che comprende le scelte che
hanno avuto un impatto su essa.

→ Se ad un certo punto la variabile che viene scelta non può essere assegnata, il backjumping va a lavorare su una delle variabili del conflict set relativo.

⇒ BJ : - Sia X_j la variabile corrente

- Sia $\text{CONF}(x_j)$ il conflict set di x_j
- Se tutti i valori possibili di x_j falliscono, si fa backjump alla variabile X_i che è stata aggiunta per ultima a $\text{CONF}(x_j)$
- Si aggiorna $\text{CONF}(x_i)$ in questo modo :

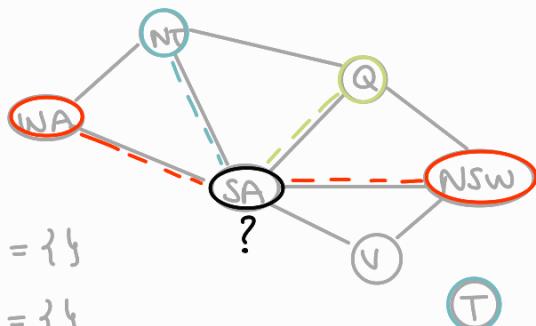
$$\text{CONF}(x_i) \leftarrow \text{CONF}(x_i) \cup \text{CONF}(x_j) - \{x_i\}$$

Assumiamo di aver fatto le seguenti scelte:

$WA = R$	}
$NSW = R$	
$T = B$	
$NT = B$	
$Q = G$	

fino ad ora consistente

$SA = ?$



$$CONF(WA) = \{\}$$

$$CONF(NSW) = \{\}$$

$$CONF(T) = \{\}$$

$$CONF(NT) = \{WA = R\}$$

$$CONF(Q) = \{NSW = R, NT = B\}$$

• $CONF(SA) = \{WA = R, NSW = R, NT = B, \underbrace{Q = G}\}$

↑ saltiamo all'ultima var del C.S.

Q: $CONF(Q) = \{NSW = R, NT = B\} \cup \{WA = R, NSW = R, NT = B, Q = G\} - \{Q\}$
 $= \{WA = R, NSW = R, NT = B\}$ (in ordine di assegnam.)

Su Q non è possibile cambiare scelta (il suo dominio è $\{G\}$)

⇒ ulteriore Backjump $\{WA = R, NSW = R, \underline{\underline{NT = B}}\}$
 ↑ vado a NT

NT: $CONF(NT) = \{WA = R\} \cup \{WA = R, NSW = R, NT = B\} - \{NT\}$
 $= \{WA = R, \underline{\underline{NSW = R}}\}$

... Anche le alternative esplorate su NT falliscono → backjump su NSW

↓
 rispetto allo stack degli assegnam.
 salto T che è ininfluente

Alcuni vincoli specifici dei CSP:

- **ALLDIFFERENT** (x_1, \dots, x_n) $\rightarrow x_1 \neq x_2 \neq \dots \neq x_n$

osservazione: se numero dei valori distinti residui nei domini delle variabili coinvolte < n
 \Rightarrow vincolo non soddisfatto

- **ATMOST** (N, A_1, \dots, A_k)

in cui: N = numero risorse equivalenti disponibili

A_1, \dots, A_k = intervalli che dicono quante risorse servono ad un particolare agente

Sudoku può essere risolto con un CSP

Ciascuna cella $= x_{ij}$ (i : riga, j : colonna)

- ALLDIFFERENT ($x_{11}, x_{12}, \dots, x_{19}$)
...
ALLDIFFERENT ($x_{11}, x_{21}, \dots, x_{91}$)
- Alcuni vincoli sono già fissati dallo schema.