



Risoluzione automatica problemi

📅 Date @May 31, 2022 → June 2, 2022

Ricerca nello spazio di stati

Criteri di valutazione di una strategia di ricerca

Lista strategie di ricerca

Ricerca in ampiezza (BFS)

Ricerca a costo uniforme

Ricerca in profondità (DFS) con e senza backtracking

Iterative deepening

Ricerca bidirezionale

Ricerca informata

Best first search

Ricerca greedy

A*

RBFS (Recursive Best First Search)

Ricerca con avversario

Approccio maximax

Approccio maximin

Approccio minimax regret

Algoritmo minimax

Minimax con α - β pruning

Problemi con soddisfacimento di vincoli

Ricerca blind

Generate and test

Ricerca di soluzione con DFS

Ricerca informata grazie ai vincoli

Forward checking

Node consistency

Arc consistency

Path consistency

Vincoli speciali

Backjumping

Definizioni

Ricerca nello spazio di stati

La realtà può essere astratta ad un insieme di **stati**. Si passa da uno stato all'altro per mezzo delle **azioni**.

Un algoritmo di ricerca degli stati determina una **soluzione** che, a partire da uno **stato iniziale** permette di raggiungere un dato **stato obiettivo**.

Un problema di ricerca è formato da una quadrupla:

- **stato iniziale**: input del problema indica la situazione dalla quale si vuole partire a cercare una soluzione
- **funzione successore**: prende in input un qualsiasi stato e ci dice se esiste e qual è lo stato successivo
- **test obiettivo**: una funzione booleana che restituisce T o F a seconda che ci si trovi o meno in uno stato goal

- **funzione di costo del cammino**: legato al consumo di qualche tipo di risorsa ci permette di stabilire qual è il cammino *migliore*

Toy problem è un problema artificiale avente lo scopo di illustrare o mettere alla prova dei metodi di risoluzione.

Metodi di ricerca **blind** si appoggiano per lo più ad una struttura ad albero detta **albero di ricerca** dove i nodi corrispondono agli **stati** del problema e gli archi a transizioni di stati causate dall'applicazione di **azioni**.

L'albero diventa un grafo quando lo stesso nodo può essere raggiunto tramite più percorsi.

Ogni nodo dell'albero tiene con sé anche altre informazioni *di contesto* ossia quale nodo è suo genitore oppure il costo accumulato per arrivare a quel nodo.

```
function RICERCA-ALBERO(problema) returns una soluzione, o il fallimento
  inizializza la frontiera usando lo stato iniziale di problema
  loop do
    if la frontiera è vuota then return fallimento
    scegli un nodo foglia e rimuovilo dalla frontiera
    if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
    espandi il nodo scelto, aggiungendo i nodi risultanti alla frontiera
```

Definizione generale di strategia di ricerca, da notare

- 1) come scegli non indichi quale modo si adotta la scelta dei nodi della frontiera, quello varia in base alla strategia applicata
- 2) come non venga indicato secondo quale politica aggiungere i nodi alla frontiera (ad esempio se LIFO o FIFO)
- 3) venga restituita la prima soluzione trovata, non una specifica

Criteri di valutazione di una strategia di ricerca

- completezza \Rightarrow garanzia di trovare una soluzione se esiste
- ottimalità \Rightarrow garanzia di trovare una soluzione ottima (a costo minimo)
- complessità temporale \Rightarrow quanto tempo occorre a trovare una soluzione
- complessità spaziale \Rightarrow quanto spazio occorre ad effettuare la ricerca

Lista strategie di ricerca

Ricerca in ampiezza (BFS)

d = profondità minima del goal

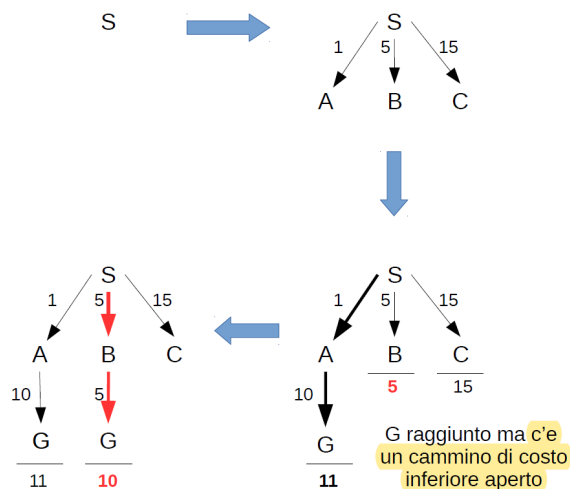
b = branching factor (ampiezza dell'albero, indica quanti successori può avere al massimo un nodo)

- espande tutti i nodi successori di un nodo
- inserisce i nodi in frontiera secondo la policy FIFO
- **completa** se esiste un percorso per un nodo goal d e b è finito
- la soluzione **ottima** è trovata solo se il costo di ogni stato non ha valori sia positivi che negativi nel suo dominio
- complessità temporale: $O(b^{d+1})$
- complessità spaziale: $O(b^{d+1})$ i nodi vanno tenuti in memoria poiché occorre ricostruire il percorso del nodo goal individuato

Ricerca a costo uniforme

simile alla ricerca in ampiezza con la differenza che la **frontiera è mantenuta ordinata**

- a ogni iterazione espande il nodo appartenente a un cammino di costo minimo
- quando trova il nodo goal non si ferma subito, ma controlla se ci sono cammini di costo inferiore aperti e in caso prova ad espanderli



Esempio slide 47

- completa se tutti i costi sono positivi
- la soluzione **ottima** è trovata solo se tutti i costi sono positivi (non riesce a gestire i *guadagni*)
- complessità spaziale e temporale dell'ordine del branching factor elevato al numero di passi del percorso ottimale se i costi dei passi fossero uniformi

Ricerca in profondità (DFS) con e senza backtracking

m = profondità massima albero

- espande sempre il nodo più profondo della frontiera, cioè il più lontano dalla radice
- quando la ricerca tenta di espandere un nodo che **non ha successori**, l'effetto è che il nodo **viene rimosso** e si "torna indietro" ad uno dei suoi fratelli
- inserisce i nodi in frontiera secondo la policy LIFO
- completo solo se b ed m sono finiti
- l'ottimalità non è garantita, viene individuato il primo nodo goal
- complessità temporale: $O(b^m)$
- complessità spaziale: $O(b * m)$
- complessità spaziale variante con **backtracking**: $O(b)$
- per evitare che la ricerca entri in percorsi infiniti può essere introdotto un limite l dove: un nodo viene espanso solo se la sua profondità $p \leq l$
altrimenti viene trattato come un nodo privo di successori

Iterative deepening

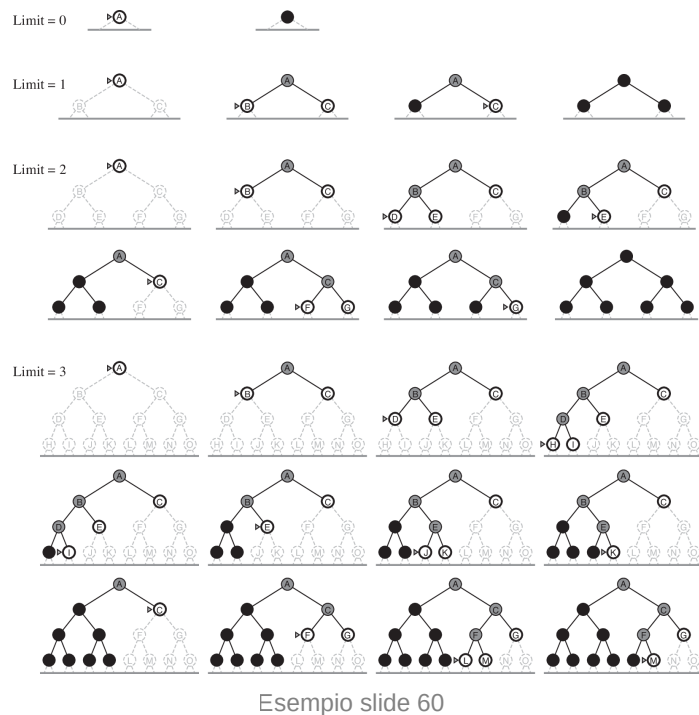
d = profondità minima del goal

b = branching factor (ampiezza dell'albero, indica quanti successori può avere al massimo un nodo)

un misto fra BFS e DFS

- esegue una ricerca in profondità limitata, con iterazioni successive in cui la profondità massima viene aumentata via via

- a ogni iterazione l'albero di ricerca viene ricostruito da zero



- complessità temporale: $O(b^d)$ a differenza di quella in ampiezza che ha complessità $O(b^{d+1})$
- complessità spaziale: $O(b * d)$
- è la migliore strategia di ricerca **blind** che si possa implementare poiché combina gli aspetti migliori di DFS e BFS
- **completa** se esiste un percorso per un nodo goal d e b è finito
- la soluzione **ottima** è trovata solo se il costo di ogni stato non ha valori sia positivi che negativi nel suo dominio

Ricerca bidirezionale

- attivare due ricerche in parallelo una che parte dallo stato iniziale e va verso il goal e una che parte dal goal e va verso lo stato iniziale
- termina quando le due ricerche si incontrano ovvero quando le frontiere hanno intersezione non vuota
- complessità spaziale e temporale: $2 * O(b^{d/2})$

Marcare i nodi già visitati e controllare sempre se i nodi da esplorare sono per caso già stati visitati rende (molto) più efficiente la ricerca \Rightarrow **grafi di ricerca**

Ricerca informata

Aggiungere l'euristica per permettere di concentrare l'attenzione su un sottoinsieme delle possibili alternative, cercando dunque di identificare le strade più **promettenti**.

Best first search

Ricerca greedy



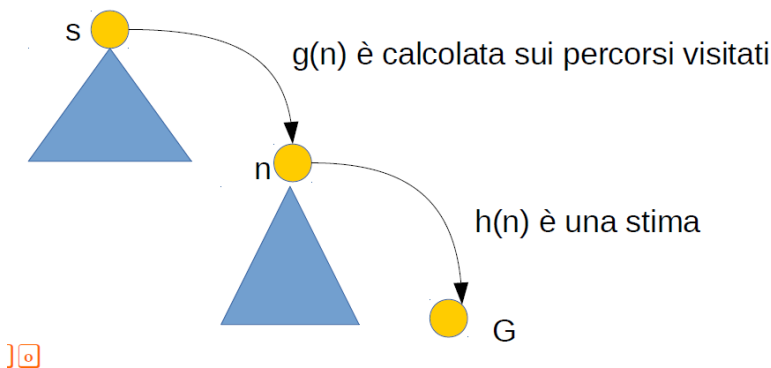
$$f(n) = h(n)$$

Viene scelto il nodo stimato più vicino a quello obiettivo

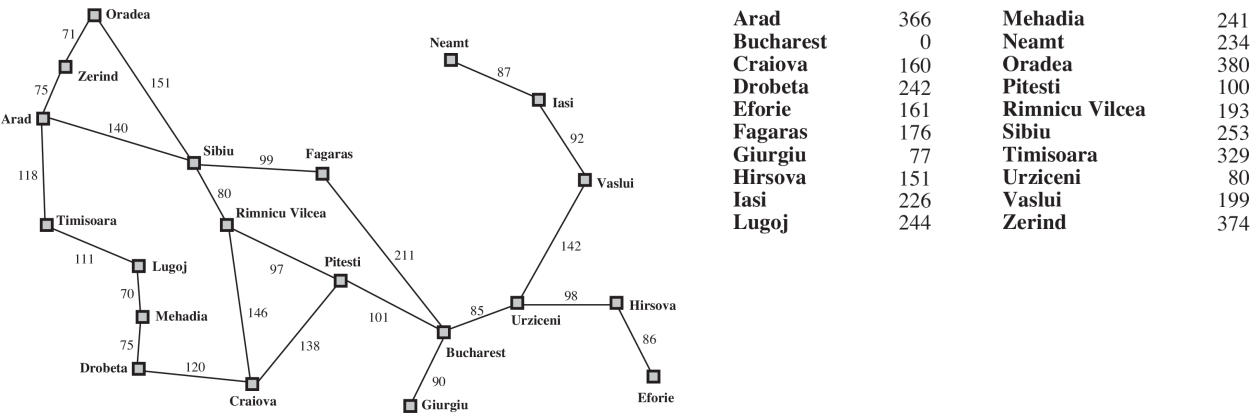
A*


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

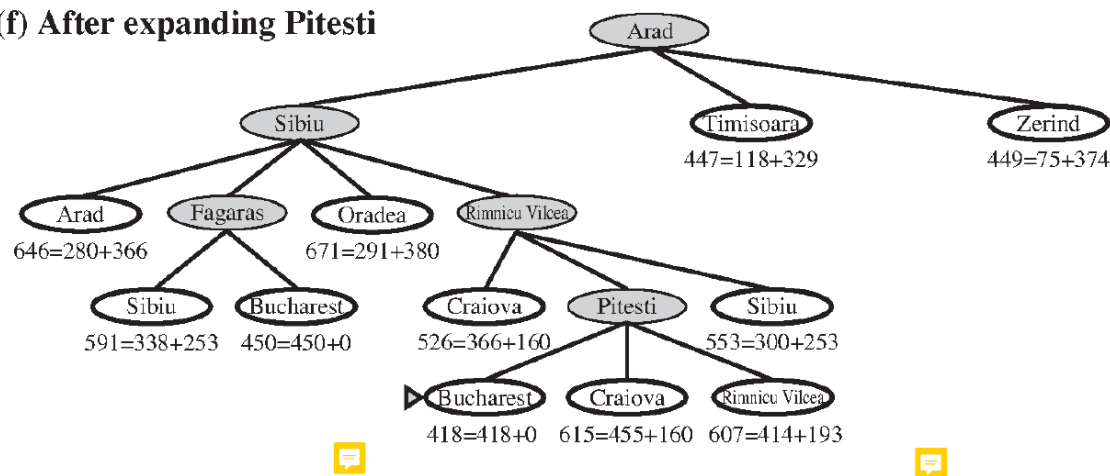
dove:
 $g(n)$: costo minimo di tutti i percorsi, visti fino ad ora, che consentono di raggiungere il nodo n a partire dallo stato iniziale s
 $h(n)$: stima del costo minimo del proseguimento di percorso che consente di raggiungere un goal preferito di n
 $f(n)$: stima del costo minimo per raggiungere un goal preferito di n partendo da s



- l'idea è di combinare l'informazione sul **futuro** con l'informazione che deriva da un'esperienza fatta (**passato**)
- combina la ricerca greedy con quella a costo uniforme
- è ottimo e completo quando tutti i costi per andare da un nodo ad un successore sono > 0 e l'euristica è **ammissibile**
- mantiene in memoria tutti i nodi generati (stessa complessità spaziale di BFS, $O(b^d)$)
- la complessità temporale dipende dall'euristica utilizzata



(f) After expanding Pitesti



Esempio di applicazione di A*: il problema della Romania (da Arad a Bucharest)



Quando siamo in presenza di un **grafo di ricerca** piuttosto che un albero per garantire l'ottimalità dell' algoritmo occorre aggiungere un ipotesi su h : h **deve essere monotona**

RBFS (Recursive Best First Search)

- simile alla ricerca ricorsiva in profondità con una differenza importante: usa un **upper bound dinamico** che consente di focalizzare la ricerca sul percorso più promettente
- questo upper bound (limite superiore) ricorda la migliore alternativa fra i percorsi attualmente aperti
- mantiene in memoria solo i nodi del percorso di ricerca corrente e i loro fratelli (stessa complessità del DFS, $O(b * m)$)
- è ottimo se l'euristica è **ammissibile**

L'algoritmo ha 3 argomenti:

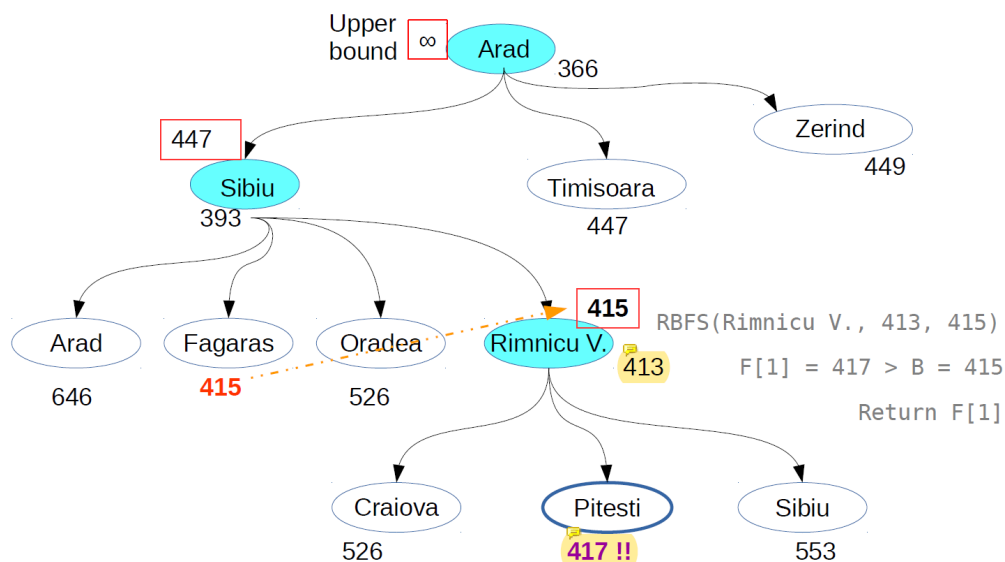
- un nodo N
- un valore $F(N)$ associato al nodo N
- un upper bound B

$$F(n) = \begin{cases} f(n), & \text{se } N \text{ esplorato per la prima volta} \\ \min(F(S_N)), & \text{altrimenti} \end{cases}$$

con S_N sottoalbero di N



Nella pratica lavora in maniera simile ad A*, con la differenza che sospende la ricerca lungo un cammino quando questo non appare più il migliore e il cammino **viene dimenticato** (nodi cancellati)



Pitesti ha una stima di costo più elevata dell'alternativa più conveniente a Rimnicu Vilcea, la condizione del while fallisce: si cambia percorso

Esempio del problema della Romania con risoluzione mediante RBFS, da notare come l'upperbound si aggiorni dinamicamente ogni volta che si trovano nuovi successori

La qualità di un'euristica può essere calcolata computando il **branching factor effettivo** b^* . Più **piccolo e compatto** è il branching factor, migliore è la ricerca dal punto di vista temporale. $b^* = 1$ significa che c'è solo un percorso da esplorare e che porta direttamente al nodo goal (è un caso ideale, quasi sempre impossibile da raggiungere).

Ricerca con avversario

Studio di strategie che tengono conto della presenza di altri agenti che possono manipolare l'ambiente in cui operiamo. Le loro azioni non saranno quindi cooperative ma **competitive**.

Tipologie di giochi:

- ad **informazione perfetta** \Rightarrow gli stati del gioco sono *totalmente* espliciti per tutti gli agenti (ad esempio gli scacchi)
- ad **informazione imperfetta** \Rightarrow gli stati del gioco sono solo *parzialmente* esplicitati (ad esempio i giochi con le carte)

Obiettivo del giocatore è determinare una strategia (sequenza di mosse) che porti alla vittoria.

soluzione \neq strategia perché cerco di influenzare il gioco \Rightarrow non ho controllo di tutto

Approccio maximax



Guarda i payoff più alti per ogni possibile scelta e fa la scelta che promette di più in assoluto.

L'approccio scommette sul fatto che si verifichi sempre l'evento migliore possibile.

Approccio maximin



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

Approccio minimax regret


alternative	sale	stabile	scende
fondo1	40	45	5
fondo2	70	30	-20
azioni	55	40	-10

Tabella originale dei payoff

alternative	sale	stabile	scende
fondo1	30	0	0
fondo2	0	15	25
azioni	15	5	15

Tabella dei regret

Per calcolare le caselle della tabella dei regret occorre sottrarre il valore più alto della colonna ad ogni valore payoff. Ad esempio nella prima casella della prima colonna si avrà 70-40=30, nella prima casella della seconda 45-45=0 e così via..



Si calcolano i massimi regret per ogni scelta alternativa e poi si seleziona l'alternativa che porta al regret minimo.

L'approccio cerca di minimizzare il pentimento.

Algoritmo minimax

m = profondità massima albero

b = branching factor (ampiezza dell'albero, indica quanti successori può avere al massimo un nodo)

MAX è il giocatore che muove per primo

MIN è il giocatore avversario, le cui mosse non possono essere controllate da MAX

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.

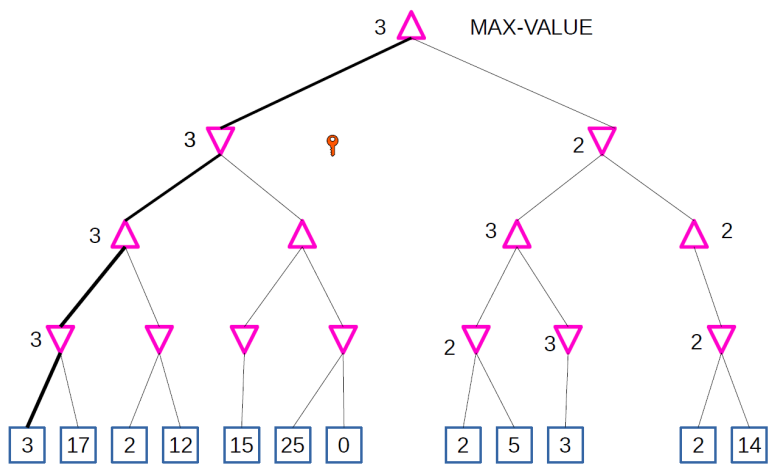
$$\text{minimax}(n) = \begin{cases} \text{utilità}(n), & \text{se } n \text{ è terminale} \\ \max_{s \in \text{SUCC}(n)}(\text{minimax}(s)), & \text{se } n \text{ è un nodo MAX} \\ \min_{s \in \text{SUCC}(n)}(\text{minimax}(s)), & \text{se } n \text{ è un nodo MIN} \end{cases}$$

minimax(*n*) 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 a meno l'utilità per l'altro giocatore.

L'algoritmo minimax ha:

- complessità temporale: $O(b^m)$
- complessità spaziale: $O(b * m)$



Intuizione:

quando siamo su un nodo MAX (triangolo) si prende il valore massimo dei figli

quando siamo su un nodo MIN (cono) si prende il valore minimo dei figli

Minimax con α - β pruning

Per ridurre i tempi di ricerca: potare i rami **meno promettenti** senza seguirli.

α e β sono valori associati a ciascun nodo che viene esplorato:

- α : **massimo lower bound** delle soluzioni possibili (è posto dai nodi MAX) e indica la prestazione migliore che MAX è sicuro di garantire fino a quel punto
- β : **minimo upper bound** delle soluzioni possibili (è posto dai nodi MIN) ed è una speculazione poiché non sappiamo cosa farà l'avversario



Ha senso esplorare un nodo se e solo se il suo **valore stimato** N , $\alpha \leq N \leq \beta$

```

function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

   $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$ 
  return the action in SUCCESSORS(state) with value  $v$ 

function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  inputs: state, current state in game
          $\alpha$ , the value of the best alternative for MAX along the path to state
          $\beta$ , the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$ 
  for  $a, s$  in SUCCESSORS(state) do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, \alpha, \beta))$ 
    if  $v \geq \beta$  then return  $v$ 
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
  return  $v$ 

function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  inputs: state, current state in game
          $\alpha$ , the value of the best alternative for MAX along the path to state
          $\beta$ , the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$ 
  for  $a, s$  in SUCCESSORS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, \alpha, \beta))$ 
    if  $v \leq \alpha$  then return  $v$ 
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return  $v$ 

```

Pseudocodice dell'algoritmo minimax con α - β pruning

L'algoritmo minimax **può avere**:

- complessità temporale: $O(b^{\frac{m}{2}})$
- complessità spaziale: $O(b * m)$

Problemi con soddisfacimento di vincoli

CSP, Constraint Satisfaction Problem sono definiti da un **insieme di variabili** e un **insieme di vincoli**.

In alcuni casi è richiesta la soddisfazione di una **funzione obiettivo**.

Gli **stati** di un CSP sono dati da tutti gli assegnamenti possibili per le variabili del CSP.

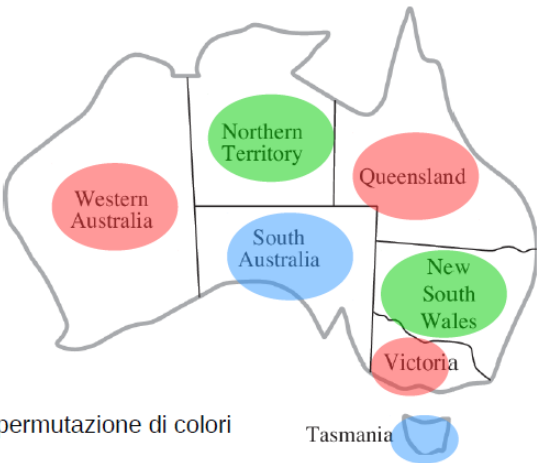
L'**azione** corrisponde sempre ad un assegnamento.

Un assegnamento è detto:

- **completo** quando assegna valori a tutte le variabili del CSP
- **consistente** quando non viola alcun vincolo del CSP

7 variabili, una per territorio, i cui domini sono tutti uguali {R, G, B}

WA = R
NT = G
SA = B
Q = R
NSW = G
V = R
T = B



NB: qualunque permutazione di colori è una soluzione

Esempio di risoluzione di un CSP: problema dell’Australia

L'**ordine** (il cammino) con cui i valori vengono assegnati alle variabili è **irrelevante sul risultato**.

Ricerca blind

Generate and test

- è il primo approccio alla risoluzione di un CSP, approccio **blind**
 - Finché non hai una soluzione o non hai alternative:
 - 1) Genera un assegnamento completo
 - 2) Controlla se è consistente
 - 3) Se sì è una soluzione, esci dal ciclo
 - 4) Se no torna al passo 1
 - Se hai una soluzione restituiscila
 - Altrimenti fallimento

Algoritmo generate and test

- molto oneroso e l'insieme delle alternative deve **essere finito**, altrimenti non termina

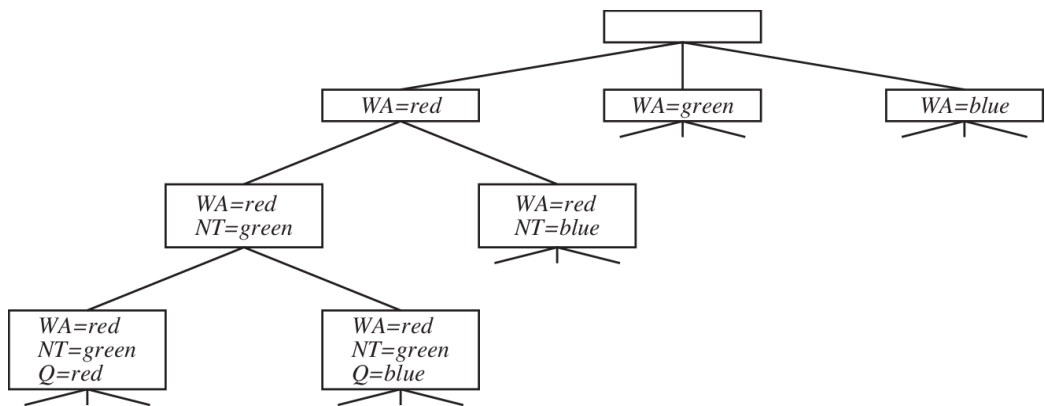
Ricerca di soluzione con DFS

n = numero delle variabili

d = numero medio dei valori possibili per ciascuna variabile

L'albero avrà $n! * d^n$ foglie.

Visto che i CSP sono **commutativi** per far in modo di generare meno foglie possibile: prima si sceglie una variabile e poi un valore per questa variabile.



Nel sottoalbero di sinistra, ad esempio, WA varrà sempre rosso.


Non è particolarmente efficiente, in particolare è soggetta a thrashing, ossia quando lo stesso assegnamento errato può essere ripetuto in più punti dell'albero.

Ricerca informata grazie ai vincoli

Se, invece di una scelta casuale, si adotta un'**euristica di scelta** si possono migliorare notevolmente le prestazioni. Ecco un elenco di euristiche di scelta:

- ▼ euristiche di scelta della variabile
 - **euristica minimum remaining values** (o **fail-first**): sceglie una delle variabili con il minor numero di valori alternativi consistenti
 - **euristica di grado**: sceglie la variabile coinvolta in più vincoli
- ▼ euristiche di scelta del valore
 - **euristica del valore meno vincolante**: prediligere il valore che lascia più libertà alle variabili adiacenti sul grafo dei vincoli

Forward checking



Si percorrono gli archi che collegano il nodo, corrispondente alla variabile assegnata, con i suoi **vicini diretti** e si riduce il range dei possibili valori di tali vicini in maniera conforme al vincolo

	WA	NT	Q	NSW	V	SA	T
INIZIO	R G B	R G B	R G B	R G B	R G B	R G B	R G B
WA=R →	R	G B	R G B	R G B	R G B	G B	R G B
Q=G →	R	B	G	R B	R G B	B	R G B
V=B →	R	B	G	R	B		R G B

Confinano tutti con SA

SA non ha più valori legali quindi si ottiene un fallimento e di conseguenza il backtracking

- è un approccio più che una **proprietà di consistenza**
- ha una **visibilità limitata**

Node consistency



Un grafo di vincoli è **node consistent** quando lo sono tutte le sue variabili

- riguarda singole variabili, vale quando i vincoli unari sono soddisfatti da tutti i valori dei domini delle rispettive variabili

Arc consistency



Un grafo di vincoli è **arc consistent** quando rispetta tutti i vincoli binari

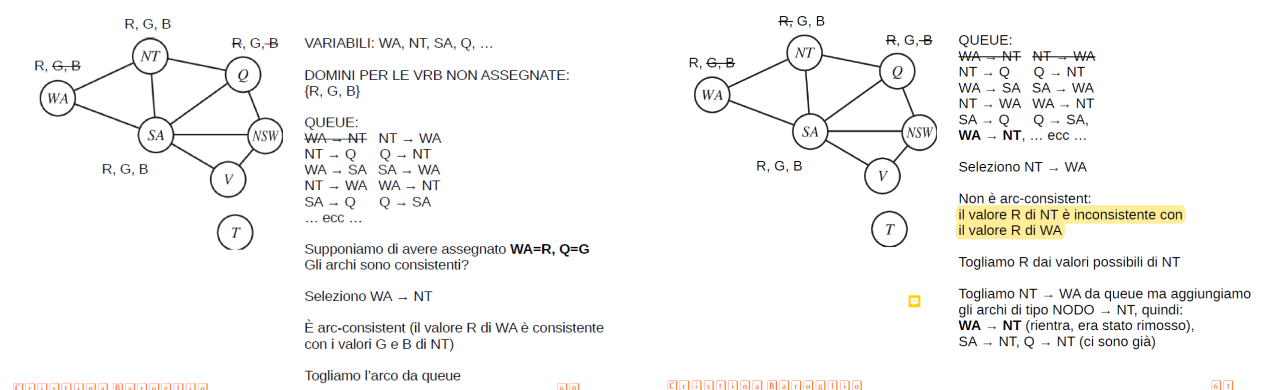
Esempio:

- se il dominio di **WA** è **{R, G}** e il dominio di **NSW** è **{R}**
- **WA → NSW** non è consistente
se WA assume valore R, NSW non ha valori assegnabili
- **NSW → WA** è consistente

WA → NSW non è consistente se WA=R perché a quel punto NSW non può valere R (dato che gli assenamenti devono essere tutti diversi) e non può valere nient'altro perché nel suo dominio ha solo R

- proprietà direzionale relativa a un vincolo binario, tutti i valori consistenti di una variabile x possono essere estesi a y tramite i vincoli
- **non** è però generalmente sufficiente a determinare una soluzione o ad accorgersi dell'irrisolvibilità di un CSP

AC-3 è un algoritmo di arc consistency che prende in input un csp e restituisce un csp con i **domini ridotti sulle variabili**



L'idea è quella di valutare se un insieme di assegnamenti risulta valido o meno, e se sì; cercare di ridurre il numero di valori del dominio di ogni variabile.

Si procede quindi inserendo tutti gli archi in una coda, si estrae un arco e si verifica che sia **consistente**: se lo è verrà semplicemente rimosso dalla coda, altrimenti:

- 1) si **rimuove il valore** che crea inconsistenza dal dominio della variabile (nodo di partenza, nella seconda immagine NT)
- 2) si **rimuove l'arco** dalla coda
- 3) si **aggiungono in coda** tutti gli archi della forma *NODO qualsiasi* → *NODO sorgente inconsistente* (nella seconda immagine sempre NT)

Se una variabile arriva ad ottenere **dominio vuoto** allora significa che l'assegnamento che si stava provando a fare era **inconsistente**.

AC guarda un arco per volta, quindi ha una **visione molto locale** del problema

Path consistency



Una coppia di variabili $\{x_1, x_2\}$ è **path consistent rispetto a una terza variabile x_3** quando:
per ogni assegnamento $\{x_1 = a, x_2 = b\}$ **consistente** con i vincoli su x_1 e x_2 , c'è un assegnamento di x_3 che soddisfa i vincoli esistenti sulle coppie di variabili $\{x_1, x_3\}$ e $\{x_2, x_3\}$

- proprietà più forte di arc consistency
 - proprietà che lega una coppia di variabili a una terza tramite i vincoli
-
- **1-consistency**: node consistency
 - **2-consistency**: arc consistency
 - **3-consistency**: path consistency

Vincoli speciali

▼ *All different*(x_1, \dots, x_n)

Se le n variabili possono assumere al più m valori distinti con $m < n$ il vincolo non può essere soddisfatto

▼ *Atmost*(N, A_1, \dots, A_k)

- N è il numero di risorse disponibili
- A_1, \dots, A_k rappresentano le quantità di risorse assegnate alle varie attività

Backjumping

L'obiettivo è migliorare il backtracking dato che è un meccanismo **blind**.



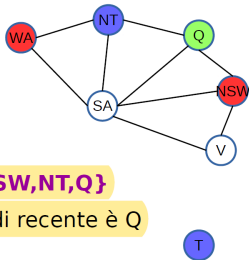
Il **backjumping** è una variante del backtracking che utilizza i **conflict set** per decidere a quale variabile ritornare in caso di vicolo cieco, in particolare torna indietro alla **variabile assegnata più di recente** che ha un valore che partecipa a creare il conflitto

- Supponiamo di assegnare le variabili in quest'ordine:

WA = R;
NSW = R;
T = B;
NT = B;
Q = G

- **SA = ?**
conf(SA) = {WA, NSW, NT, Q}

- La vrb aggiunta più di recente è Q



Esempio che mostrare come calcolare il conflict set di una variabile

- Backjump a Q:
 - $\text{conf}(Q) = \{\text{NSW}, \text{NT}\}$
 - Lo aggiorniamo arricchendolo dell'informazione di conflitto fornita da SA: $\text{conf}(Q) \leftarrow \text{conf}(Q) \cup \text{conf}(\text{SA}) - \{Q\}$
 $\text{conf}(Q) \leftarrow \{\text{NSW}, \text{NT}\} \cup \{\text{WA}, \text{NSW}, \text{NT}, Q\} - \{Q\}$
 - Quindi **conf(Q) = {WA, NSW, NT}**
- **NB:** WA non confina con Q ma prende parte al generarsi del conflitto e allora viene ricordato anche in associazione a Q. Viene rilevato un "vincolo implicito"

Immagine che mostra come aggiornare il conflict set di una variabile dopo aver fatto backjump su di essa

Definizioni

- ▼ Dato

simbolo da elaborare e contestualizzare

▼ Informazione

elaborazione di più dati

▼ Conoscenza

insieme di informazioni correlate fra loro

▼ Automazione vs autonomia

automazione: sequenza di passi che possono andare avanti senza l'intervento umano

autonomia: dare ad un computer un compito di più alto livello (un **obiettivo**) cui spetta l'onere di trovare il modo di raggiungere

▼ Obiettivo

risultato verso il quale gli sforzi sono diretti

▼ Nodo di un albero/grafico di ricerca

è una struttura dati che mantiene molte informazioni:

(1) lo stato del problema cui si riferisce;

(2) informazioni di struttura (genitore, figli);

(3) informazioni di contabilità. Direzione degli archi: da figlio a genitore

▼ Soluzione in un problema di ricerca su alberi

percorso che porta dal nodo iniziale ad un nodo obiettivo

▼ Approccio blind vs approccio informato

blind: si usa esclusivamente la struttura del problema per cercare (e trovare) una soluzione

informato: si usa la struttura del problema + ulteriore conoscenza (ad esempio euristiche) per guidare la ricerca

▼ Frontiera

insieme dei nodi generati ma non ancora espansi

▼ Variante con backtracking

i successori di un nodo vengono aggiunti alla frontiera soltanto dopo aver scoperto che quel nodo non ha più successori

▼ Soluzione di un grafo di ricerca

si intende il cammino che porta da uno stato iniziale ad uno stato goal.

▼ Funzione euristica

associata ad ogni nodo, ci dà una stima del minimo fra i costi dei percorsi che congiungono lo stato corrispondente al nodo a uno stato goal

▼ Euristica ammissibile

Una funzione euristica h è detta **ammissibile** quando $\forall n, h(n) \leq h^*(n)$

dove $h^*(n)$ è il costo minimo reale per raggiungere il nodo goal a partire dal nodo n .

▼ Euristica consistente (o monotona)

dati un qualsiasi nodo n e un qualsiasi suo successore n' prodotto eseguendo l'azione a in n vale che $h(n) \leq c(n, a, n') + h(n')$

▼ Euristica dominante

posso costruire l'**euristica dominante** da un insieme di euristiche prendendo sempre l'**euristica maggiore**

▼ Giochi a somma zero

è un contesto di interazione multiagente in cui la perdita (o il guadagno) di un agente è compensata dal guadagno (perdita) degli altri

▼ Osservabilità di un gioco

totale: nel caso di giochi con turno, i giocatori conoscono i risultati delle mosse precedenti

parziale: nel caso di giochi ad azione simultanea: i giocatori non conoscono le mosse che i giocatori eseguono simultaneamente alla loro

▼ Funzione di utilità

valuta solo gli stati terminali (del gioco): ci dice se lo stato della partita è una vittoria, una sconfitta o un pareggio per MAX (non è quindi assoluta ma **relativa al giocatore**)

▼ Trasposizioni

sequenze di mosse che con ordine differente mi portano allo stesso stato

▼ Nodo quiescente

la funzione di valutazione si mantiene costante

▼ Dominio di una variabile

insieme dei valori che una variabile può assumere

▼ Arità dei vincoli

vincoli **unari**: coinvolgono una variabile ed un valore

vincoli **binari**: coinvolgono due variabili

etc...

▼ Vincoli vs criteri di preferenza

Affinché un CSP sia risolto tutti i vincoli devono essere soddisfatti. I criteri di preferenza, che possono essere violati da una soluzione, invece, permettono di ordinare le soluzioni identificando quelle preferibili e quelle meno preferibili.

▼ Inferenza

andare a propagare informazioni locali sul nostro grafo di ricerca

▼ Consistenza locale

focalizzarsi su una **proprietà di consistenza** relativa ad una parte del CSP (un sottoproblema) e poi propagare questa informazione in tutto il CSP cercando di costruire una soluzione

▼ Variabile node consistent

Una variabile è **node consistent** quando rispetta tutti i vincoli unari che la riguardano

▼ Arco consistente

Un arco è detto **consistente** quando **per ogni** valore del dominio del nodo sorgente (da cui parte l'arco) **esiste** almeno un valore nel dominio del nodo destinazione che permette di rispettare i vincoli

▼ Conflict set di una variabile

Sia A un assegnamento parziale consistente, sia x una variabile non ancora assegnata. Se l'assegnamento $A \cup \{x = v_i\}$ risulta inconsistente per qualsiasi valore v_i appartenente al dominio di x si dice che A è un **conflict set** di x



Il conflict set è un **insieme di assegnamenti**

▼ Conflict set minimo

Un conflict set per una variabile è minimo quando togliendo uno qualsiasi degli assegnamenti che lo costituiscono non si ottiene più un conflict set

▼ Stato insicuro di un CSP

assegnamenti **NOGOOD**: sono assegnamenti parziali che non corrispondono a un blocco ma portano necessariamente ad un blocco