I giochi con avversario

Maria Simi

a.a. 2021/2022

Russell-Norvig

(AIMA, cap. 5 - Adversarial search)

Premessa

- "*Problem solving*" come ricerca
- Paradigma base
  - Ambiente osservabile, deterministico, utente singolo
  - Stati atomici
- Rilassamento delle assunzioni di base
  - [Azioni non deterministiche, ambiente parzialmente osservabile … viste]
  - Ambienti multi-agente, *competitivi* (ricerca con avversario)
  - Rappresentazioni degli stati più complesse

Specializzazioni del paradigma

- I giochi con avversario
  - I piani devono tenere conto dell’avversario
- I problemi di soddisfacimento di vincoli (cenni)
  - Lo stato ha una struttura *fattorizzata*
- I sistemi basati su conoscenza
  - Lo stato è una “base di conoscenza” a cui rivolgere domande sul mondo, rappresentato in un linguaggio espressivo.
  - Tecniche efficienti di “ragionamento” nell’ambito di formalismi logici noti: il calcolo proposizionale (PROP) e la logica del prim’ordine (FOL).

I giochi con avversario

- Regole semplici e formalizzabili
- Ambiente accessibile e determistico
- Due giocatori, turni alterni, *a somma zero*, informazione perfetta
- Ambiente multi-agente *competitivo*:
  - la presenza dell'avversario rende l'ambiente *strategico* ⇒ più difficile rispetto ai problemi di ricerca visti finora
- Complessità e vincoli di tempo reale:
  - la mossa migliore nel tempo disponibile

⇒ *i giochi sono un po' più simili ai problemi reali*

Ciclo *pianifica-agisci-percepisci*

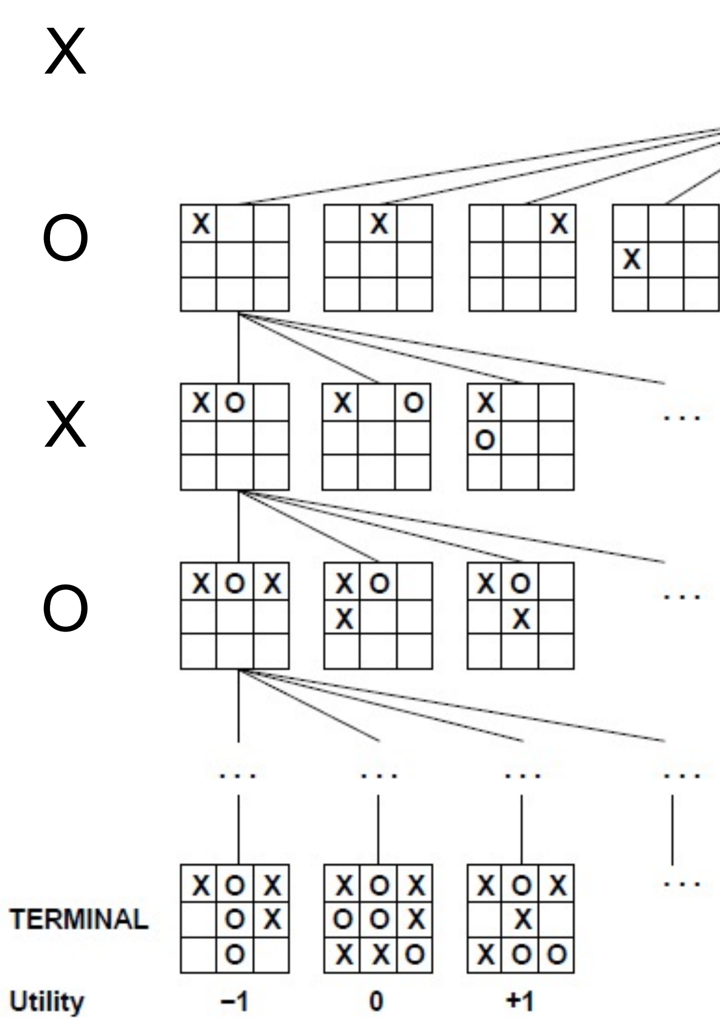
- Caso di due agenti che agiscono “a turno”:
  - si può ancora *pianificare* considerando le possibili risposte dell’avversario, e le possibili risposte a queste risposte …
- Una volta decisa la mossa migliore da fare:
  - Si esegue la mossa
  - Si vede cosa fa l’avversario
  - Si ri-pianifica la prossima mossa

Giochi come problemi di ricerca

- *Stati*: configurazioni o posizioni del gioco
  - $Player(s)$: a chi tocca muovere nello stato s
- *Stato iniziale*: configurazione iniziale del gioco
- $Actions(s)$: le mosse legali in s
- $Result(s, a)$: lo stato risultante da una mossa
- $Terminal\text{-}Test(s)$: determina la fine del gioco (se uno stato è terminale)
- $Utility(s, p)$: funzione *di utilità* (o *pay-off*), valore numerico che valuta gli stati terminali del gioco per p

  Es. 1|-1|0, conteggio punti, … somma costante

Tris/Filetto (Tic-tac-toe)

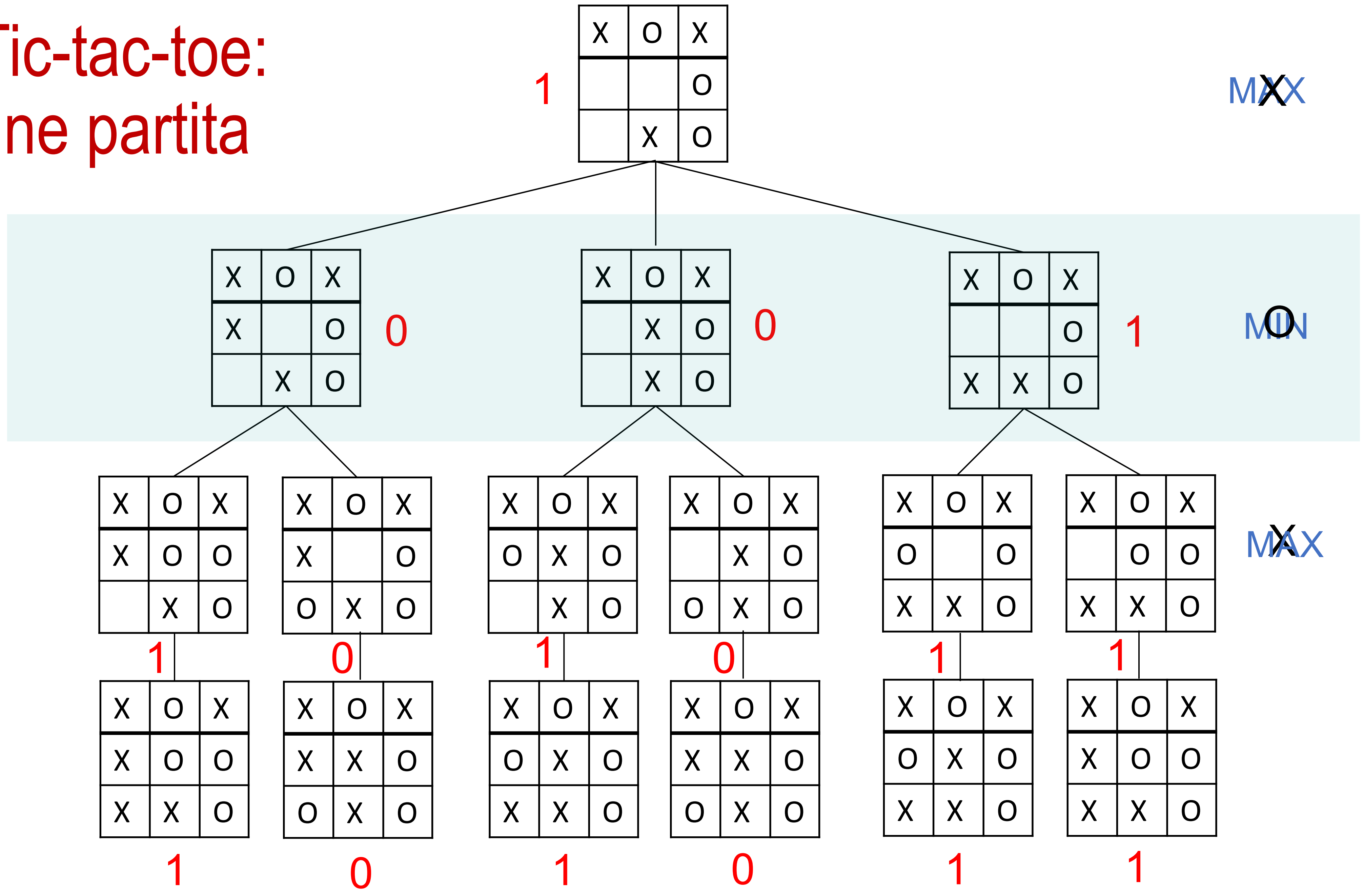


Albero di gioco parziale

Che cosa vedremo

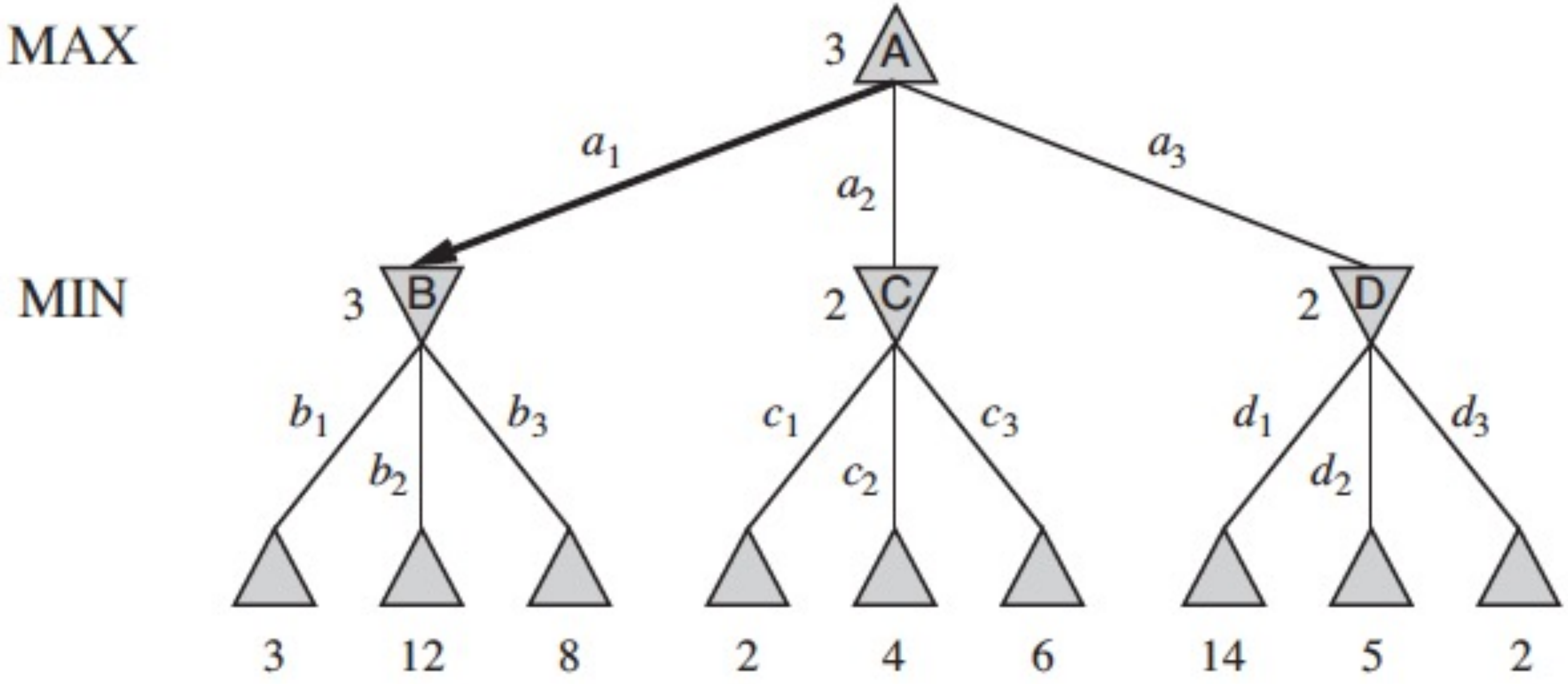
1. La decisione ottima teorica: come si sceglie la mossa migliore in un gioco con uno spazio di ricerca completamente esplorabile (MIN-MAX)
2. Estensione a giochi in cui, a causa della complessità, non è possibile una esplorazione esaustiva
3. Tecniche di ottimizzazione della ricerca (ALFA-BETA)
4. Solo cenni a giochi che richiedono soluzioni più complesse

Tic-tac-toe: fine partita



Ricerca minimax

Esempio di una mossa (+ contromossa)



Albero di gioco profondo “una mossa”, due strati o *ply*

Calcolo del valore *minimax* dei nodi

La decisione minimax nella radice è a_1

Valore *Minimax*

$\text{Minimax}(s) =$

- $\mathit{Utility}(s, \text{MAX})$ if $\mathit{Terminal\text{-}Test}(s)$
- $\max_{a \in \mathit{Actions}(s)} \text{Minimax}(\mathit{Result}(\text{s}, a))$ if $\mathit{Player}(s) = \text{MAX}$
- $\min_{a \in \mathit{Actions}(s)} \text{Minimax}(\mathit{Result}(s, a))$§ if $\mathit{Player}(s) = \text{MIN}$

- Gioco ottimo se MIN gioca in maniera ottima
- Ma la mossa ottima conviene sempre con un avversario subottimo?
- Come conviene esplorare l’albero di gioco?

L’algoritmo MIN-MAX ricorsivo

function MINIMAX-DECISION(*state*) **returns** *an action*
  return arg max$_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

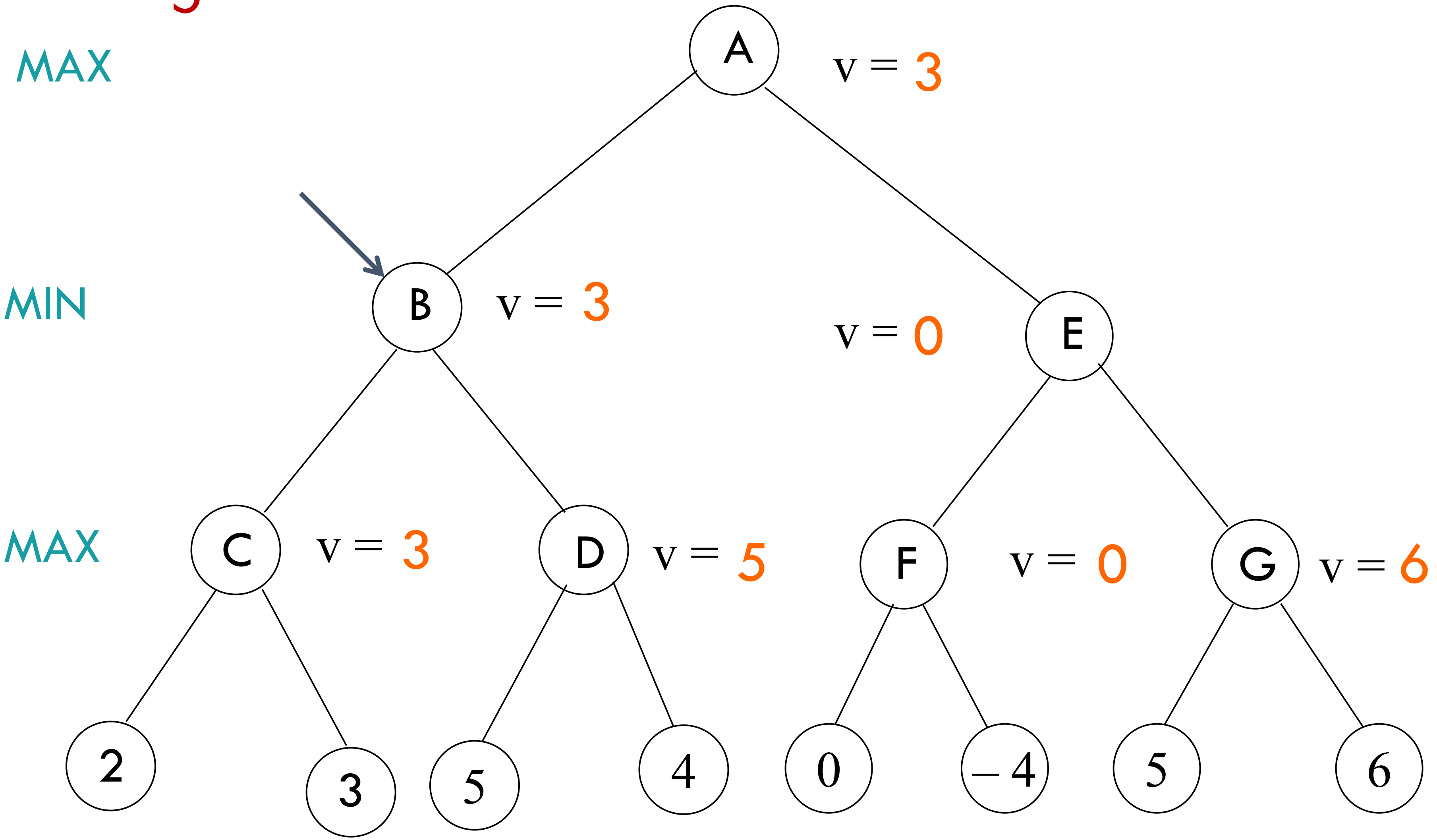
---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

Min-Max: algoritmo in azione



Costo

- Tempo: come DF, $O(b^m)$; spazio: $O(m)$.
- Scacchi: 35^{100} (35 mosse in media; 50 mosse per player)
- Grafo degli scacchi 10^{40} nodi → 10^{22} secoli!!! [Nilsson]
- Improponibile una esplorazione sistematica del grafo degli stati, se non per giochi veramente semplici (come il filetto)
- È necessario fare uso di euristiche per stimare la bontà di uno stato del gioco.

Min-Max euristico (con orizzonte)

- In casi più complessi occorre una funzione di valutazione euristica dello stato, *Eval*(s).
- *Strategia*: guardare avanti d livelli
  - Si espande l'albero di ricerca un certo numero di livelli d (compatibile col tempo e lo spazio disponibili)
  - si valutano gli stati ottenuti e si propaga indietro il risultato con la regola del MAX e MIN
- Algoritmo MIN-MAX come prima ma …

if *Terminal-Test*(s) **then return** *Utility*(s) diventa

➔ **if** *Cutoff-Test*(s, d) **then return** *Eval*(s) *Cutoff-Test riconosce stati terminali …*

Valore H-Minimax

Se d è il limite alla profondità consentita ...

H-Minimax(s, d) =

- $Eval(s)$ if $Cutoff\text{-}Test(s, d)$
- $\max_{a \in Actions(s)}$ H-Minimax$(Result(s, a), d+1)$ if $Player(s)$=MAX
- $\min_{a \in Actions(s)}$ H-Minimax$(Result(s, a), d+1)$ if $Player(s)$=MIN

Nota: $Cutoff\text{-}Test(s, d)$ restituisce *True* su stati terminali.

Il filetto

$$Eval(s) = X(s) - O(s)$$

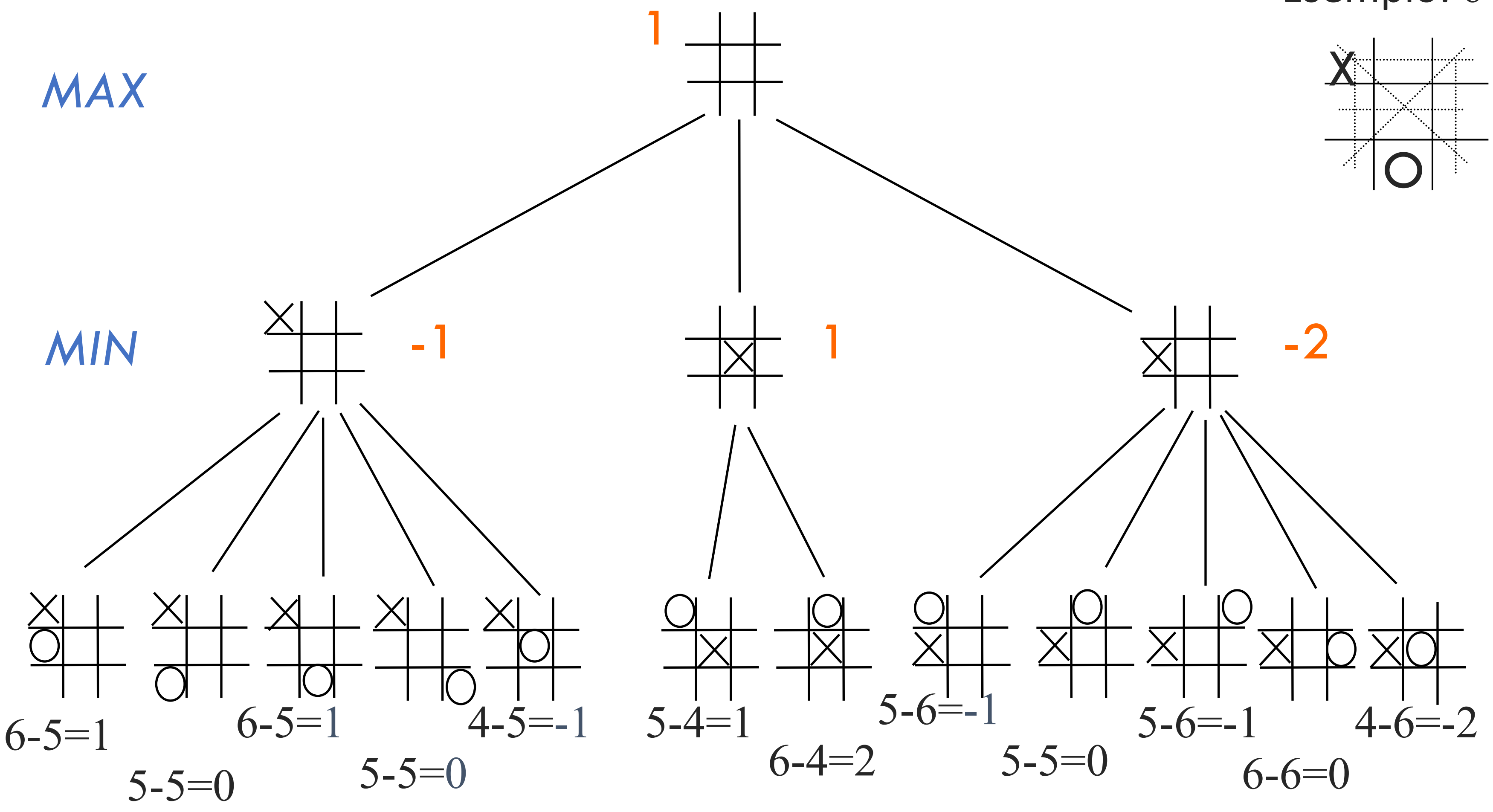
$X(s)$ righe aperte per X

$O(s)$ righe aperte per O

Una configurazione vincente per X viene valutata +∞

Una vincente per O, -∞

Esempio: 6-5=1



La funzione di valutazione

- La funzione di valutazione *Eval* è una stima della utilità attesa a partire da una certa posizione
- La qualità della *funzione* è determinante:
  - deve essere consistente con l’utilità se applicata a stati terminali del gioco (indurre lo stesso ordinamento).
  - deve essere efficiente da calcolare;
  - deve riflettere le probabilità effettive di vittoria (A valutato meglio di B se da A ci sono più probabilità di vittoria che da B)
  - valore “atteso” che combina probabilità con utilità dello stato terminale; può essere appreso con l’esperienza

Esempio

- Per gli scacchi si potrebbe pensare di valutare caratteristiche diverse dello stato:
  - Valore del materiale (pedone 1, cavallo o alfiere 3, torre 5, regina 9 …)
  - Buona disposizione dei pedoni, protezione del re
- Funzione lineare pesata

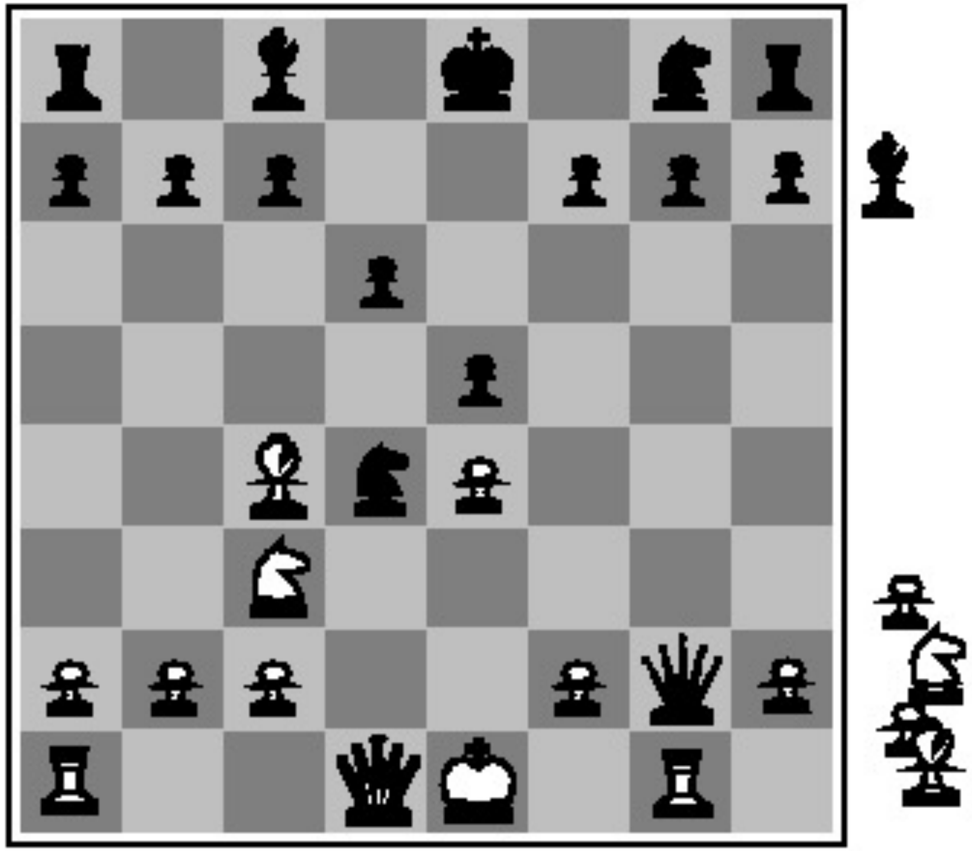
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- Combinazioni non lineari di caratteristiche

  Alfiere vale più nei finali di partita, 2 alfieri valgono più del doppio di 1

Problemi con MIN-MAX: stati non quiescienti

- *Stati non quiescienti*: l'esplorazione fino ad un livello può mostrare una situazione molto vantaggiosa
- Alla mossa successiva la regina nera viene catturata.
- *Soluzione*: applicare la valutazione a stati *quiescenti* stati in cui la funzione di valutazione non è soggetta a mutamenti repentini (ricerca di quiescenza)



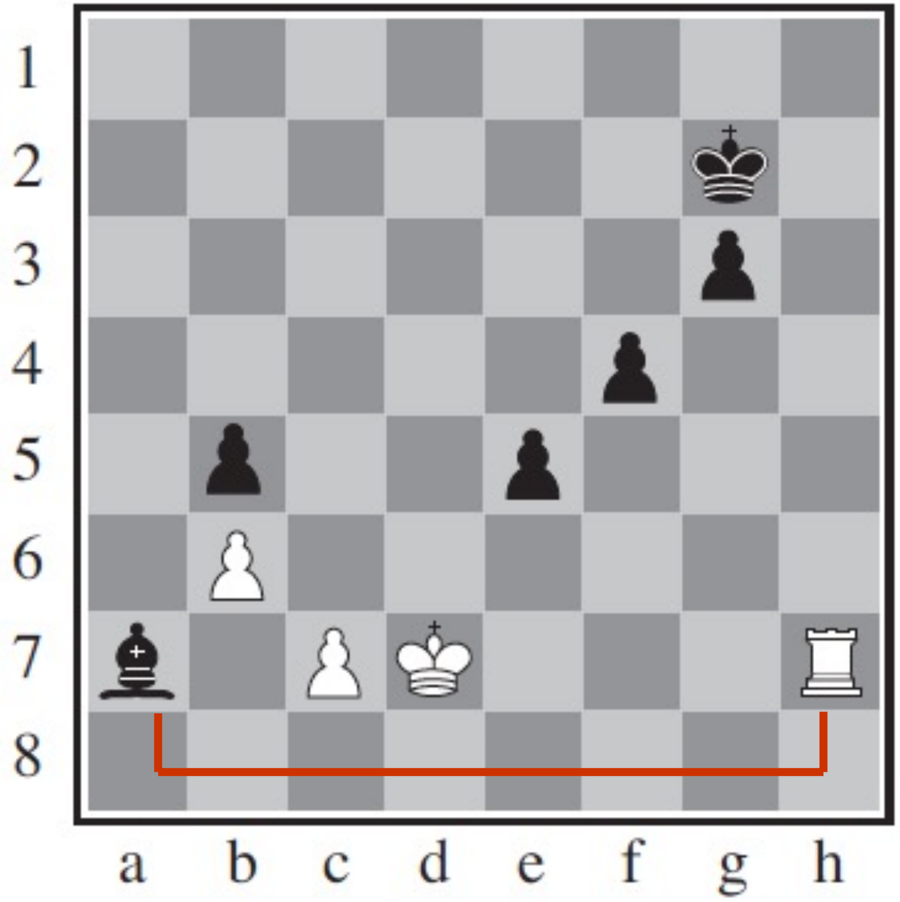
(b) White to move

Problemi con MIN-MAX: effetto orizzonte

Effetto orizzonte: può succedere che vengano privilegiate mosse diversive che hanno il solo effetto di spingere il problema oltre l'orizzonte

L'alfiere in a7, catturabile in 3 mosse dalla torre, è spacciato.

Mettere il re bianco sotto scacco con il pedone in e5 e poi con quello in f4 … evita il problema temporaneamente, ma è un sacrificio inutile di pedoni.

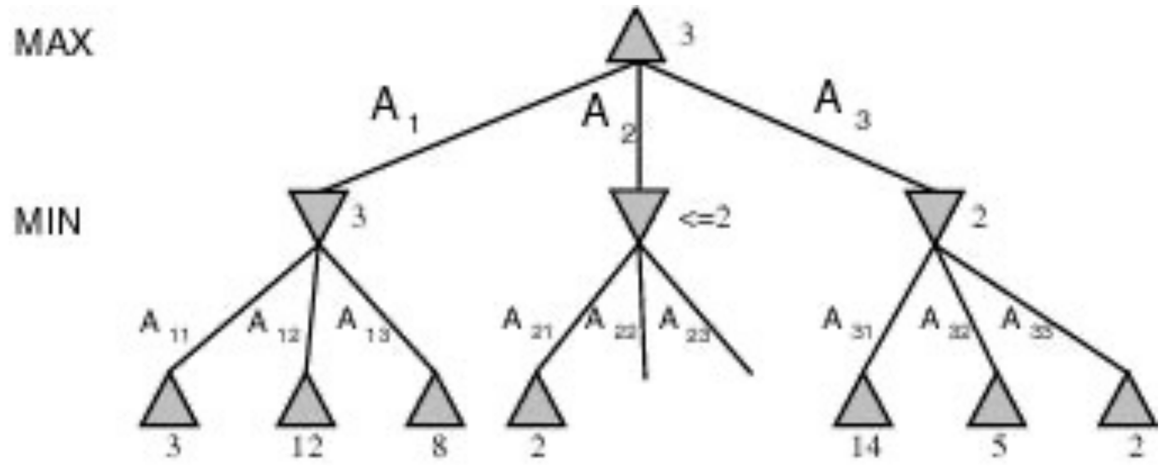


Ottimizzazione

- Ma dobbiamo necessariamente esplorare ogni cammino?
- NO, esiste un modo di dimezzare la ricerca pur mantenendo la decisione minimax corretta
- Potatura alfa-beta (McCarthy 1956, per scacchi)

Potatura alfa-beta: l'idea

- Tecnica di *potatura* per ridurre l'esplorazione dello spazio di ricerca in algoritmi MIN-MAX.



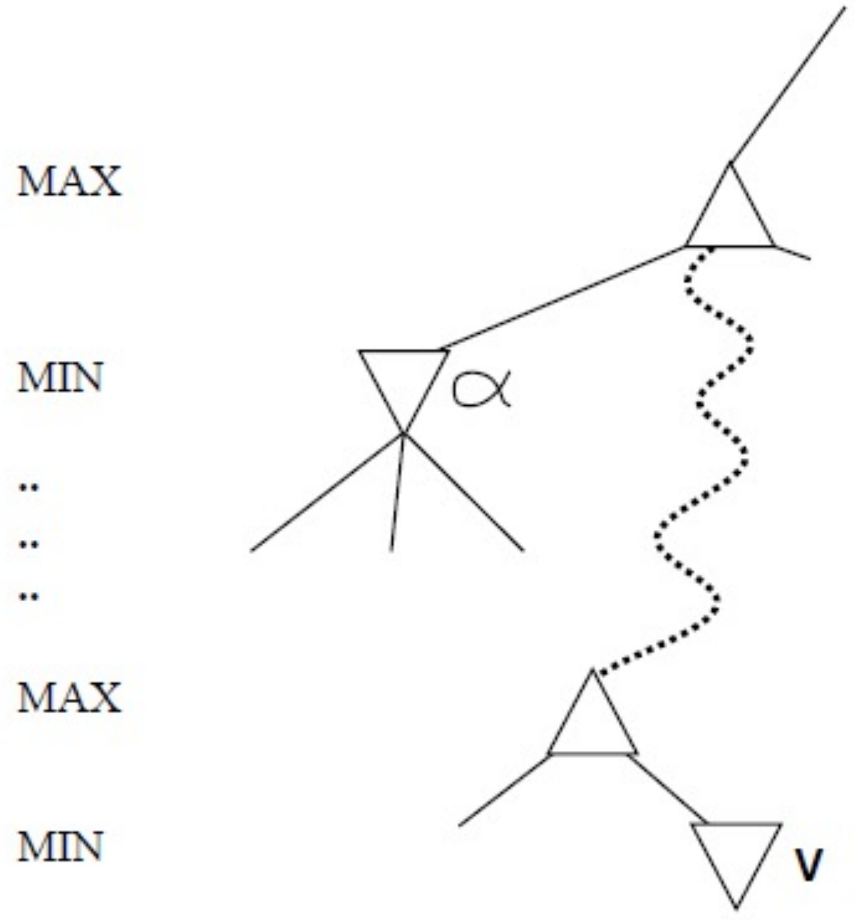
MINMAX(*radice*) = max(min(3, 12, 8), min(2, x, y), min(14, 5, 2))

= max(3, min(2, x, y), 2)

= max(3, z, 2) = 3 con z ≤ 2

Più in generale

- Consideriamo v
- Se c'è una scelta migliore sopra, quel v non sarà mai raggiunto
- MAX, passerà da α piuttosto che finire a v



Potatura alfa-beta: implementazione

- Si va avanti in profondità fino al livello desiderato e propagando indietro i valori si decide se si può abbandonare l'esplorazione nel sotto-albero.
  - MaxValue e MinValue vengono invocate con due valori di riferimento: α (inizialmente $-\infty$) e β (inizialmente $+\infty$)
  - Rappresentano rispettivamente la migliore alternativa per MAX e per MIN fino a quel momento.
  - I tagli avvengono quando nel propagare indietro:

    $v \geq \beta$ per i nodi MAX (taglio β)

    $v \leq \alpha$ per i nodi MIN (taglio α)

L'algoritmo Alfa-Beta: max

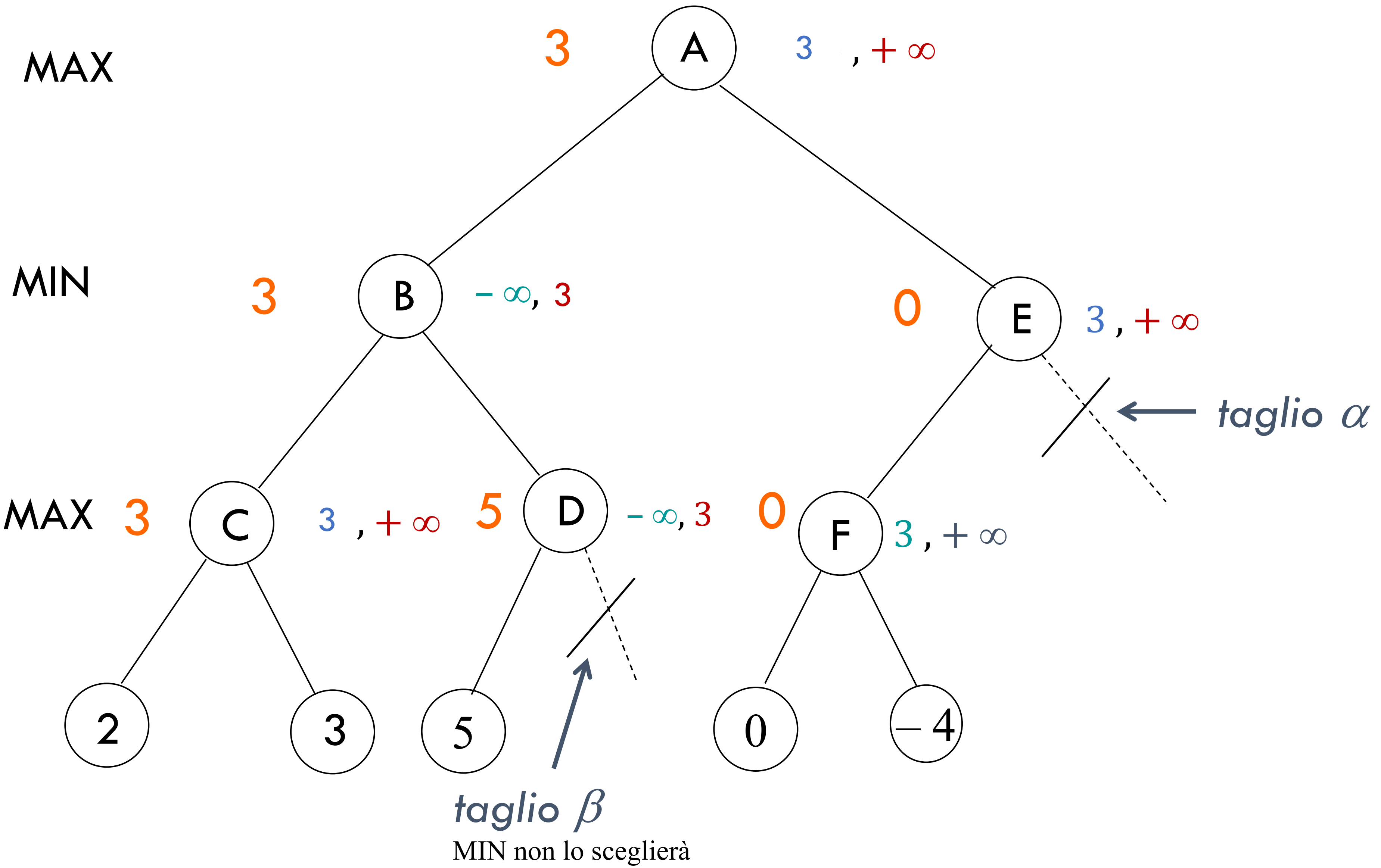
function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v ← *taglio* β
    $\alpha \leftarrow$ MAX(α, v)
  return v

L'algoritmo Alfa-Beta: min

function $\textsc{Min-Value}(state, \alpha, \beta)$ **returns** *a utility value*
  if $\textsc{Terminal-Test}(state)$ **then return** $\textsc{Utility}(state)$
  $v \leftarrow +\infty$
  for each a **in** $\textsc{Actions}(state)$ **do**
    $v \leftarrow \textsc{Min}(v, \textsc{Max-Value}(\textsc{Result}(s,a), \alpha, \beta))$
    if $v \leq \alpha$ **then return** v $\leftarrow$ *taglio* α
    $\beta \leftarrow \textsc{Min}(\beta, v)$
  return v

Alfa-beta in azione



Ordinamento delle mosse

- La potatura ottimale si ottiene quando ad ogni livello sono generate prima le mosse migliori per chi gioca:
  - Per nodi MAX sono generate prima le mosse con valore più alto
  - Per i nodi MIN sono generate prima le mosse con valore più basso (migliori per MIN)
- Complessità: $O(b^{m/2})$ anziché $O(b^m)$
- Alfa-Beta può arrivare a profondità doppia rispetto a Min-Max!
- Ma come avvicinarsi all'ordinamento ottimale?

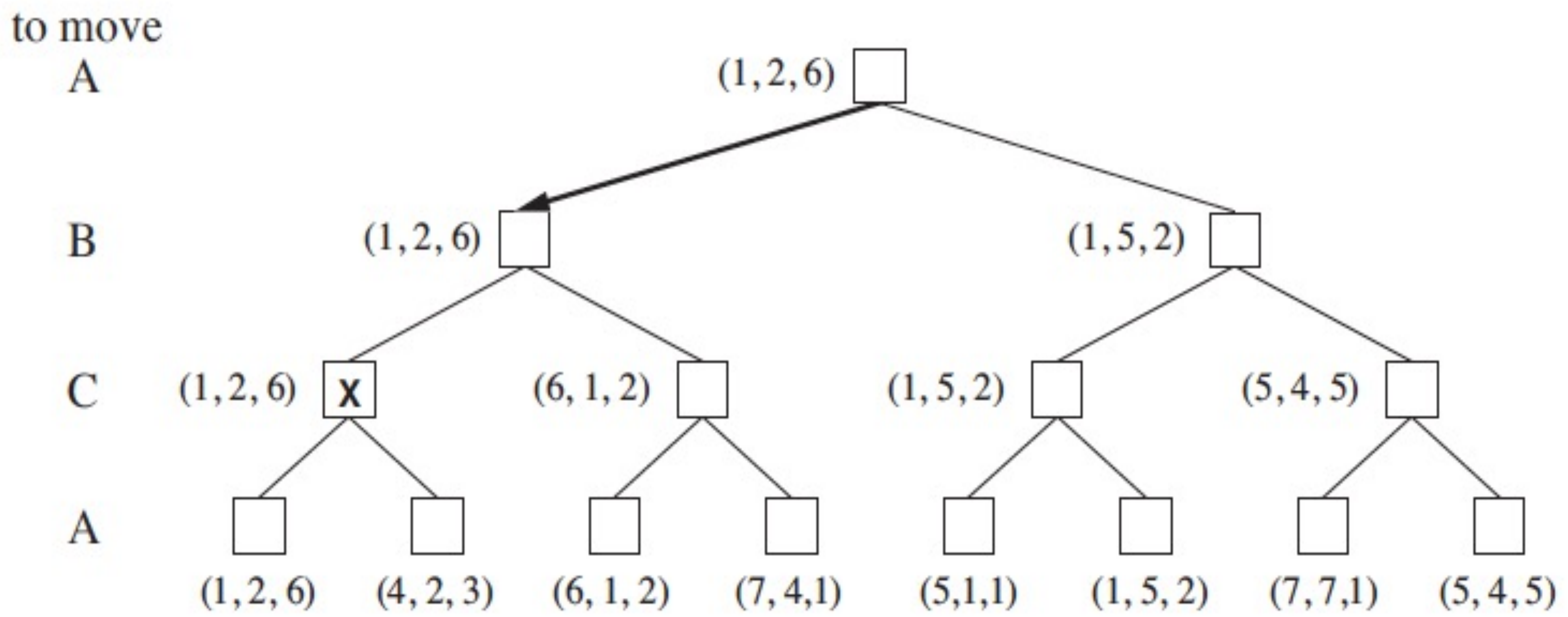
Ordinamento dinamico

1. Usando l'*approfondimento iterativo* si possono scoprire ad una iterazione informazioni utili per l'ordinamento delle mosse, da usare in una successiva iterazione (mosse killer).
2. Tabella delle trasposizioni: per ogni stato incontrato si memorizza la sua valutazione
   - Trasposizione: [a_1, b_1, a_2, b_2] e [a_1, b_2, a_2, b_1] portano nello stesso stato
3. Strategia in ampiezza (di tipo A) vs strategia in profondità (di tipo B)

Altri miglioramenti

1. Potatura in avanti: esplorare solo alcune mosse ritenute promettenti e tagliare le altre
   - Beam search
   - Tagli probabilistici (basati su esperienza). Miglioramenti notevoli in Logistello [Buro]
2. Database di mosse di apertura e chiusura
   - Nelle prime fasi ci sono poche mosse sensate e ben studiate, inutile esplorarle tutte
   - Per le fasi finali il computer può esplorare off-line in maniera esaustiva e ricordarsi le migliori chiusure (già esplorate tutte le chiusure con 5 e 6 pezzi …)

Giochi multiplayer



1. (v_a=1, v_b=2, v_c=6) valutazioni per A, B, C
2. Il valore backed-up in x è il vettore migliore per C

Giochi più complessi

- Giochi stocastici: i giochi in cui è previsto un lancio di dadi come il backgammon
- Giochi parzialmente osservabili
  - *deterministici*
    - ✓ Le mosse sono deterministiche ma non si conoscono gli effetti delle mosse dell’avversario. Es. Battaglia navale, Kriegspiel
  - *stocastici*
    - ✓ Le carte distribuite a caso in molti giochi di carte. Es. Bridge, whist, peppa, briscola …

Problemi di soddisfacimento di vincoli

Maria Simi

a.a. 2021/2022

Problemi di soddisfacimento di vincoli (CSP)

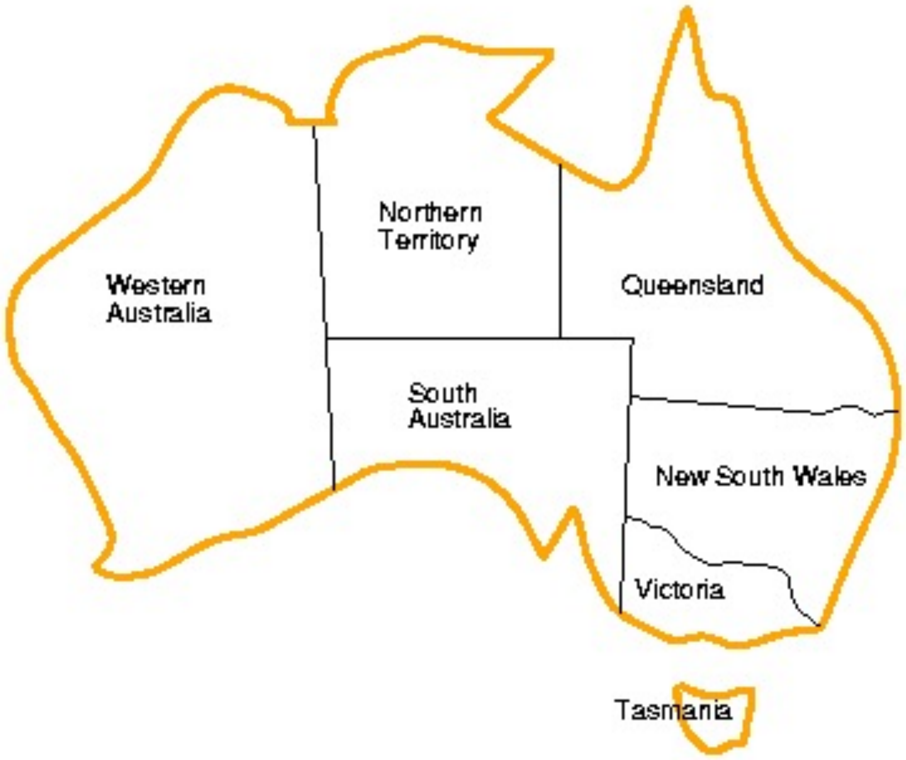
- Sono problemi con una struttura particolare, che si prestano ad algoritmi di ricerca specializzati
- È un esempio di rappresentazione fattorizzata, in cui si comincia a dire qualcosa sulla struttura dello stato
- Esistono euristiche generali che si applicano e che consentono la risoluzione di problemi di dimensioni significative per questa classe
- La classe di problemi formulabili in questo modo è piuttosto ampia: layout di circuiti, scheduling, …

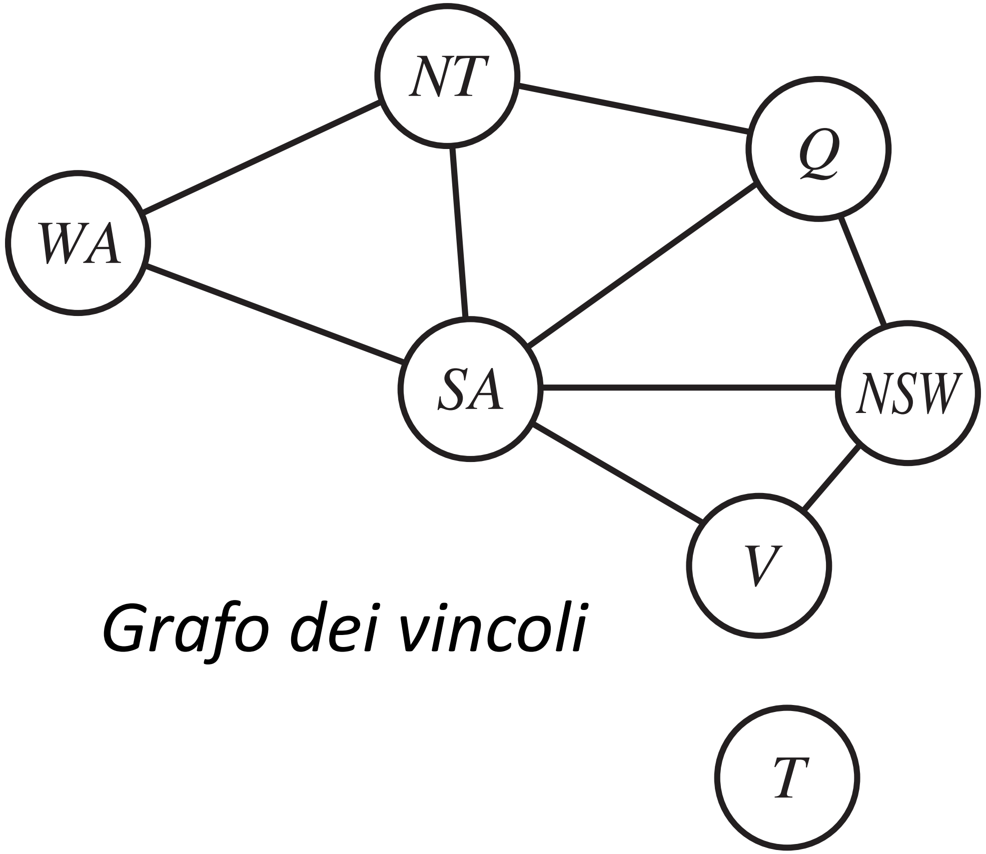
Formulazione di problemi CSP

- *Problema*: descritto da tre componenti
  1. X un insieme di variabili
  2. D un insieme di domini
     dove D_i è l' insieme dei valori possibili per X_i
  3. C un insieme di vincoli (relazioni tra le variabili)
- *Stato:* un assegnamento [parziale|completo] di valori a variabili
  $\{X_i = v_i , X_j = v_j \ldots\}$
- *Stato iniziale*: { }
- *Azioni*: assegnamento di un valore a una variabile (tra quelli leciti)
- *Soluzione (goal test)*:
  - un assegnamento completo (le variabili hanno tutte un valore) e consistente (i vincoli sono tutti soddisfatti)

Colorazione di una mappa

Si tratta di colorare i diversi paesi sulla mappa con tre colori in modo che paesi adiacenti abbiano colori diversi





Grafo dei vincoli

Le 8 regine

- Il problema delle 8 regine
  - $X = \{Q_1, \ldots, Q_8\}$ *una regina per colonna*
  - $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$ *numero di riga*
  - C, i vincoli di non attacco
- Esempio di vincolo tra Q_1 e Q_2

  $<(Q_1, Q_2) , \{(1, 3)\ (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (2, 4), \ \ldots \}>$
- Formulazione incrementale o a stato completo

Strategie per problemi CSP

- Finora potevamo solo ricercare la soluzione nel grafo degli stati (guidati da una metrica definita sullo stato).
- Adesso possiamo
  - Usare delle euristiche specifiche per questa classe di problemi
  - fare delle inferenze che ci portano a restringere i domini e quindi a limitare la ricerca: *propagazione di vincoli*
  - Fare backtracking *intelligente*
- Tipicamente un misto di queste strategie.

Ricerca in problemi CSP

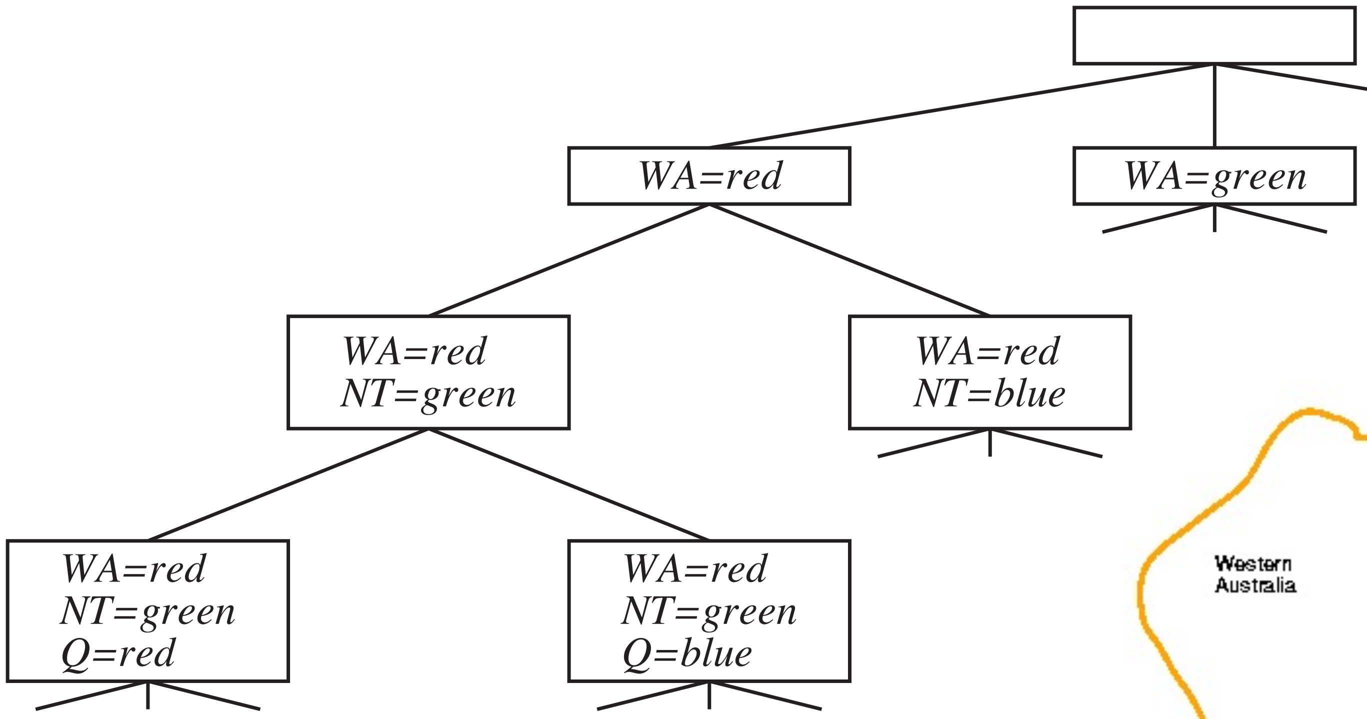
- Ad ogni passo si assegna una variabile
  - La massima profondità della ricerca è fissata dal numero di variabili *n*
- Versione «ingenua»
  - L'ampiezza dello spazio di ricerca è $|D_1| \times |D_2| \times \ldots \times |D_n|$ dove $|D_i|$ è la cardinalità del dominio di X_i
  - Il fattore di diramazione è pari a *nd al primo passo; (n*-1) *d al secondo ... le foglie sarebbero n! .* d^n
- Riduzione drastica dello spazio di ricerca dovuta al fatto che il *goal-test* è commutativo (l'ordine con cui si assegnano le variabili non è importante)
  - *In realtà il fattore di diramazione è pari* alla dimensione dei domini *d* (*d^n foglie)*

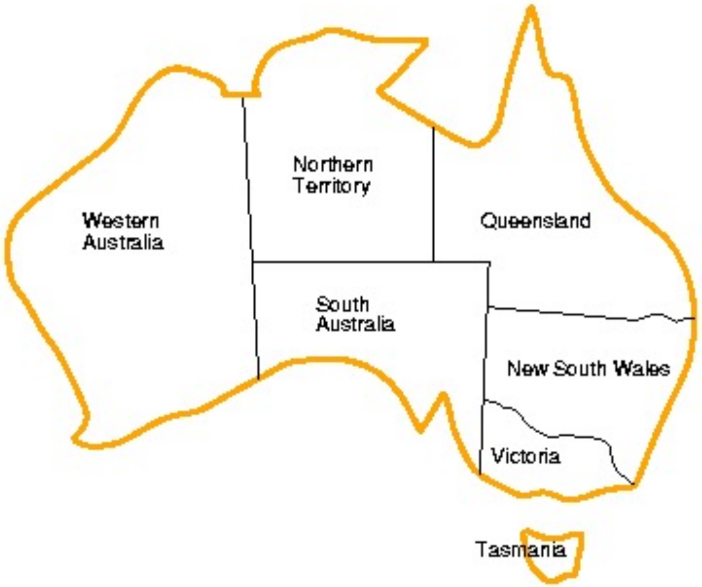
Strategia di ricerca

Ricerca con backtracking (BT) a profondità limitata

- *Controllo anticipato* della violazione dei vincoli: è inutile andare avanti fino alla fine e poi controllare; si può fare *backtracking* non appena si scopre che un vincolo è stato violato.
- La ricerca è limitata naturalmente in profondità dal numero di variabili quindi il metodo è completo.

Esempio di backtracking





Backtracking ricorsivo per CSP

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment                      trovata soluzione
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then                        controllo anticipato
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)                          riduce i domini
        if inferences ≠ failure then                                     nessun dominio è vuoto
           add inferences to assignment
           result ← BACKTRACK(assignment, csp)                          richiama sé stessa
           if result ≠ failure then
              return result
     remove {var = value} and inferences from assignment                si disfa lo stato
  return failure
```

Euristiche e strategie per CSP

- SELECTUNASSIGNEDVARIABLE
  - Quale variabile scegliere?
- ORDERDOMAINVALUES
  - Quali valori scegliere?
- INFERENCE
  - qual è l'influenza di un assegnamento sulle altre variabili? come restringe i domini?
  - *propagazione di vincoli, riduzione problema*
- BACKTRACK
  - Come evitare di ripetere i fallimenti?
  - *backtracking intelligente*

Scelta delle variabili

1. MRV (*Minimum Remaining Values*)

   scegliere la variabile che ha meno valori legali [residui], la variabile *più vincolata*. Si scoprono prima i fallimenti (*fail first*).

2. Euristica *del grado:* scegliere la variabile coinvolta in più vincoli con le altre variabili (la *variabile più vincolante* o di *grado maggiore*)

   Da usare a parità di MRV

Scelta dei valori

Una volta scelta la variabile come scegliere il valore da assegnare?

1. Valore *meno vincolante*: quello che esclude meno valori per le altre variabili direttamente collegate con la variabile scelta
   - Meglio valutare prima un assegnamento che ha più probabilità di successo
   - Se volessimo tutte le soluzioni l’ordine non sarebbe importante

Propagazione di vincoli

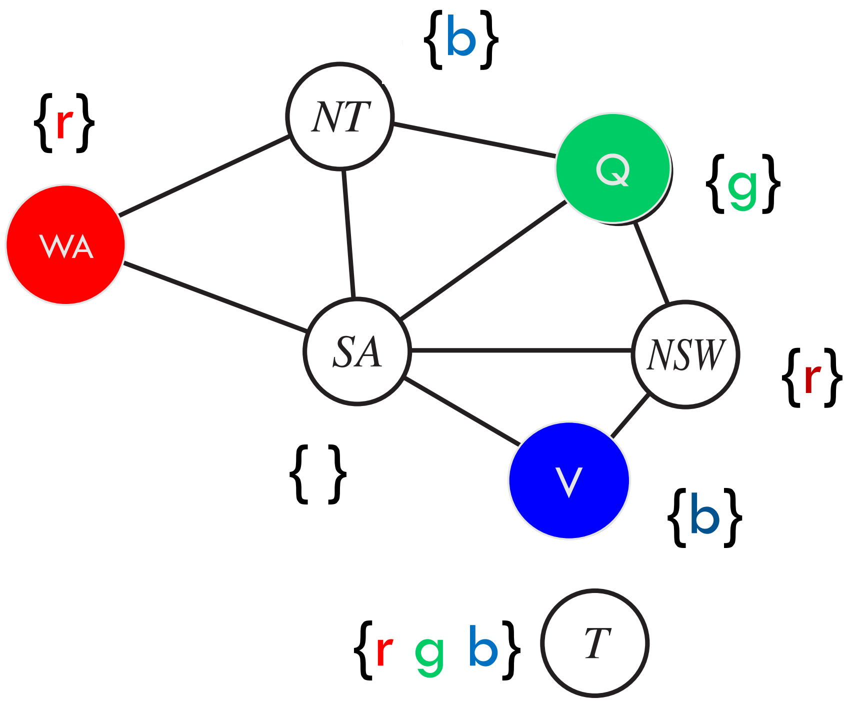
1. Verifica in avanti (*Forward Checking* o *FC*)
   - assegnato un valore ad una variabile si possono eliminare i valori incompatibili per le altre var. *direttamente collegate* da vincoli (non si itera)
2. Consistenza di nodo e d'arco
   - si restringono il valori dei domini delle variabili tenendo conto dei vincoli unari e binari su tutto il grafo (si itera finché tutti i nodi ed archi sono consistenti)

Esempio di *FC*

$WA = \mathrm{r}$

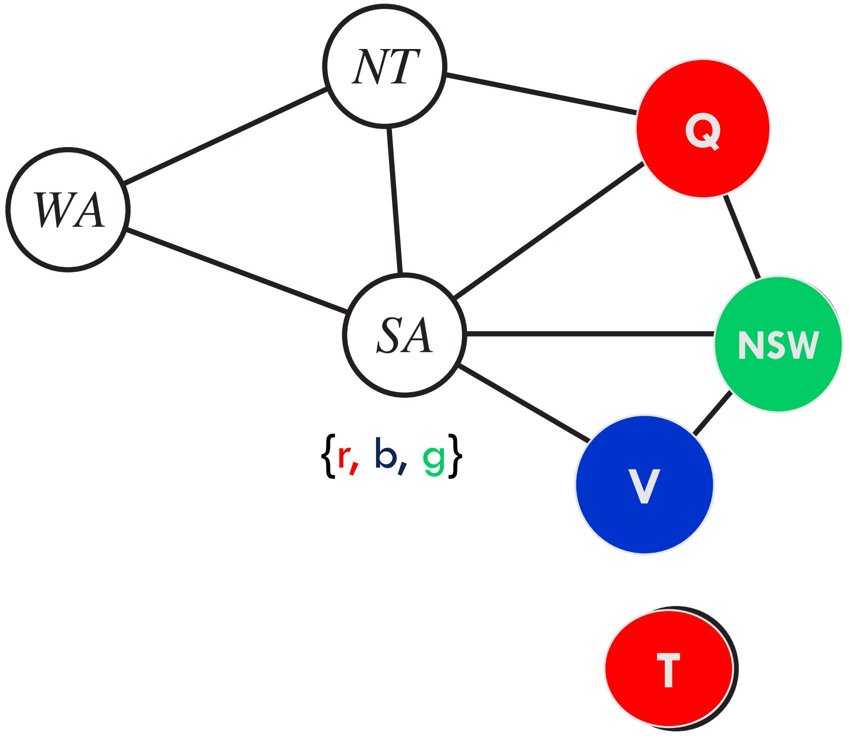
$Q = \mathrm{g}$

$V = \mathrm{b}$



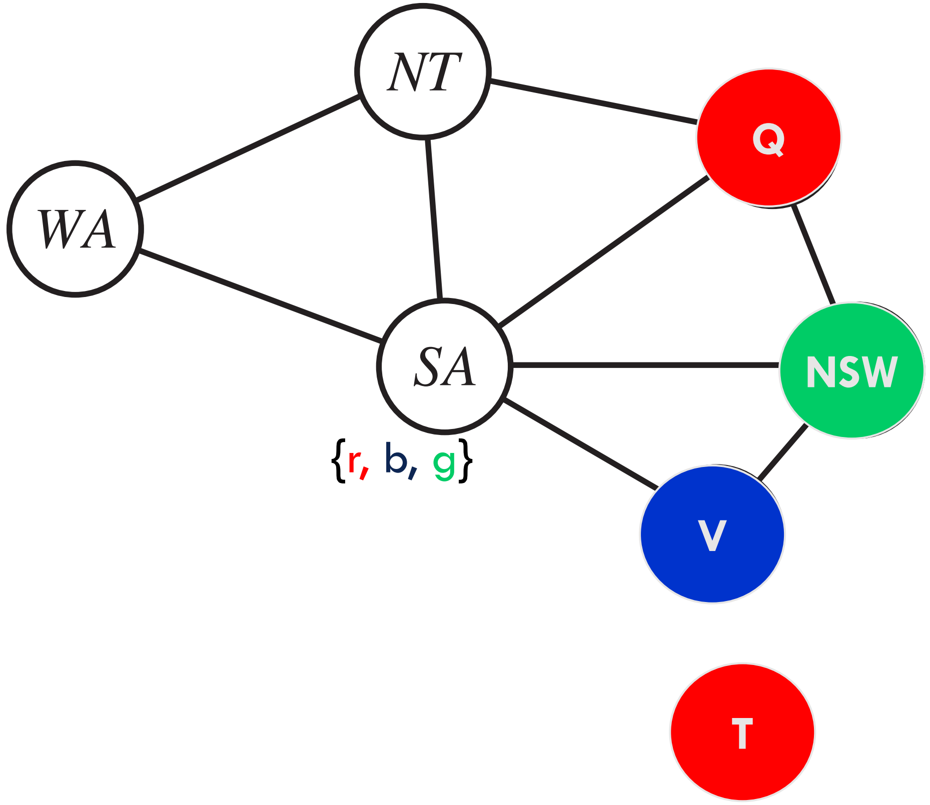
Backtracking “cronologico”

- Supponiamo di avere {Q=red, NSW=green, V=blue, T=red}
- Cerchiamo di assegnare SA
- Il fallimento genera un backtracking “cronologico”
- … e si provano tutti i valori alternativi per l’ultima variabile, T, continuando a fallire



Backtracking “intelligente”

- Si considerano alternative solo per le variabili che hanno causato il fallimento {Q, NSW, V}, *l’insieme dei conflitti*
- *Backtracking guidato dalle dipendenze*

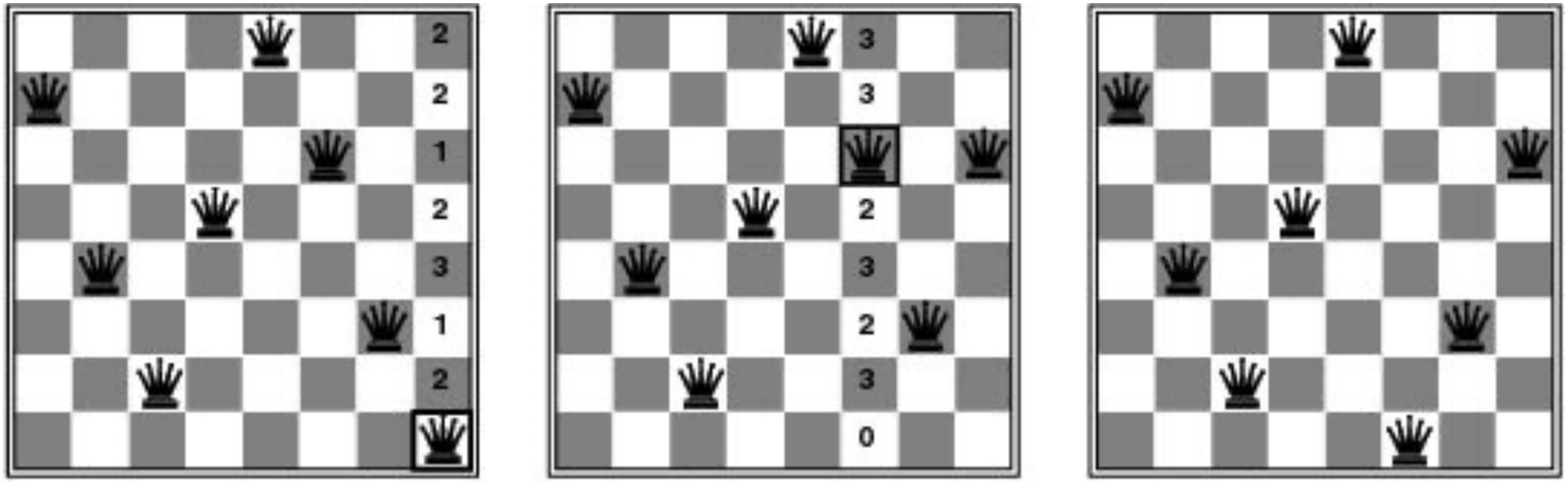


Metodi CSP locali: le regine

- Si parte con tutte le variabili assegnate (tutte le regine sulla scacchiera)
- ad ogni passo si modifica l'assegnamento ad una variabile per cui un vincolo è violato (si muove una regina minacciata su una colonna).
- È un algoritmo di *riparazione euristica*.

Min-conflicts

- Un’euristica nello scegliere un nuovo valore potrebbe essere quella dei *conflitti minimi*: si sceglie il valore che crea meno conflitti.



- *Molto efficace*: 1 milione di regine in 50 passi!

Conclusioni

- Abbiamo visto due domini specifici per il paradigma di risoluzione dei problemi come ricerca
  - I giochi con avversario, con ambienti strategici e vincoli di tempo reale
  - I CSP, in cui le tecniche di *problem solving* possono essere specializzate e usate per risolvere istanze di problemi di dimensioni maggiori.
- Prossimamente: i sistemi basati su conoscenza
  - Conoscenza implica capacità inferenziali
  - L' inferenza è anch'essa un problema di ricerca in uno spazio di stati.

Riferimenti

- Giochi con avversario (*Adversary search*)
  - AIMA Cap 5: 5.1, 5.2, 5.3, 5.4
- Constraint Satisfaction Problems (CSP)
  - AIMA Cap 6: 6.1, 6.2 (solo cenni a FC), 6.3, 6.4
- Per la seconda esercitazione:
  - Testo degli esercizi da svolgere (es2.pdf)
- Successivamente:
  - Soluzioni (es2_sol.pdf)
  - Python Notebooks per esecuzione guidata e interattiva del codice