Cataldo Cianciaruso – matricola 189070

Appunti di Intelligenza artificiale – a.a. 2018/2019

Informazioni sul corso

Prof: Luigi Palopoli, Fabio Fassetti

Ricevimento: su appuntamento

Libri: Artificial Intelligence and Modern Approach (anche versione italiana)

Modalità d'esame:

- **Orale** (obbligatorio, può essere sostenuto in qualsiasi momento)

- Il **progetto**, che consiste nella costruzione di un giocatore artificiale su un gioco concordato. Verrà quindi indetto un torneo fra i vari gruppi e stilata una classifica. In base alla posizione in classifica, si avrà un voto più alto. Il progetto va presentato entro la data prefissata, e vale fino alla sessione di settembre successiva al corso. In caso di posticipo, il voto verrà scalato di conseguenza, come penalità.

- **Scritto** (facoltativo, in alternativa al progetto):
  - 1° Esercizio (2/3 del voto): Searching
  - 2° Esercizio (1/3 del voto): Knolowedge Rappresentation

Composizione del corso

- **1° Parte:** Tecniche di costruzione di soluzioni (ricerca di conoscenza/ searching)

- **2° Parte:** Tecniche e formalismi di rappresentazione di conoscenza (ragionamenti di senso comune)

- **3° Parte:** Applicazioni:
  - Problemi di Planning
  - Problemi di Learning

Lezione 1 01/10/2018

Cos'è l'Intelligenza?

Cosa si intende per intelligenza? Vediamo qualche esempio noto.

Esempio 1 – DeepBlue

Nel 1997, il più grande campione di scacchi Garry Kasparov vene sconfitto da una macchina, in condizione di partita aperta (il giocatore era consapevole di giocare contro un'entità meccanica). Tale macchina, denominata DeepBlue, di produzione IBM, basava la propria "intelligenza" su una serie di calcoli, di conti, predefiniti. Parliamo di intelligenza?

Esempio 2 – Algoritmi di Face Recognition

Parliamo degli algoritmi di Face-recognition. Essi riescono ad estrapolare da una foto una serie di informazioni utili ad identificare il soggetto ritratto, scartando al contempo l'enorme mole di informazioni superflue (tipo di luce, paesaggio, vestiario, posa del soggetto, ecc.), tramite appositi filtri. Parliamo di intelligenza?

Ad entrambi i sistemi appena visti manca una importante caratteristica: la **flessibilità**, ossia la possibilità di impiegare le risorse disponibili in più settori, e non solo su uno (DeepBlue, ad esempio, era in grado unicamente di giocare a scacchi). Quindi, cos'è l'intelligenza?

John Ellman la definisce come il risultato di un comportamento emergente, sinergico. Ad esempio la si può ritrovare nelle formiche: un'intera colonia di formiche, grazie alla cooperazione e allo scambio di informazioni, è considerata "intelligente"; al contrario, una singola formica no.

Altri definiscono l'intelligenza come fenomeno quantistico.

Ai fini del nostro corso, definiremo l'intelligenza come una **funzione di risoluzione di un problema**, la quale deve fornire soluzioni ragionevolmente corrette (non per forza le migliori) in un tempo ragionevole (relativo alla complessità del problema). Con tale definizione, la macchina DeepBlue vista in precedenza è "intelligente", in quanto è in grado di fornire una soluzione corretta in tempi ragionevoli ad un determinato problema (nel suo caso, gli scacchi). Esiste, tuttavia, un criterio col quale stabilire se un'entità (di qualsiasi tipo) è intelligente?

Test di intelligenza: il test di Turing

Tale test fu ideato dall'omonimo matematico, Alan Turing, per stabilire se un dato "artefatto" può essere considerato "intelligente" o meno.

Esso consiste in una serie domande poste da un operatore (tester) all'entità sotto esame. Il tester non può interagire con l'entità direttamente, ma solo attraverso domande poste di volta in volta in forma scritta. Una volte poste tutte le domande previste (il test dura mezz'ora), il test si considera superato (cioè l'entità è intelligente) se il tester afferma di parlare con un essere umano. Notare che tale responso viene dato tenendo conto di vari fattori oltre alla correttezza in se delle domande: un umano, ad esempio, ha tempi di reazione nettamente inferiori ad una macchina, e il tempo di risposta potrebbe far nascere sospetti al tester sulla natura dell'entità. Lo stesso ragionamento vale per calcoli molto complicati, che una macchina è in grado di svolgere meglio di un essere umano.

Tale test non basta, però, come prova definitiva di intelligenza. Ad esempio, per superare il test, basterebbe dotare l'entità di una grande tabella, contenente tutte le possibili domande e le relative risposte. Inoltre, per ingannare il tester, basterebbe preimpostare un tempo random di risposta, in modo tale da non risultare eccessivamente veloci.

Noi faremo uso di agenti razionali, le cui decisioni sono dettate unicamente da una funzione di utilità (codificata matematicamente) che stabilisca, dati gli input, quanto una soluzione sia conveniente (vicina all'ottimo, oltre che in base al contesto, ecc.)

Lezione 2 05/10/2018

Tecniche di ricerca

Tradizionalmente in intelligenza artificiale abbiamo 3 categorie:

- **Tecniche di ricerca blind:** algoritmi di ricerca non informata (o algoritmi blind) se la scelta sul prossimo nodo da analizzare dipende solo dalla posizione del nodo all'interno dell'albero (e quindi all'interno dello spazio di ricerca)

- **Tecniche di ricerca informata (o euristica):** Gli algoritmi di ricerca informata (o algoritmi euristici) hanno invece una scelta che dipende da altre informazioni riguardanti il dominio applicativo del problema e sintetizzate in una funzione di costo detta **funzione euristica**.

- **Tecniche di ricerca con avversario:** sono tecniche di ricerca euristica, ma specializzate ad un insieme di problematiche specifiche che sono quelle che portano alla produzione di algoritmo per i giochi

Qualora sarà opportuno farlo, applicheremo a tali tecniche un'altra componente importante, che è data dalle **tecniche di randomizzazione**.

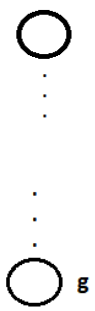
Dobbiamo immaginarci lo spazio in cui la ricerca ha luogo: in tutti i casi lo spazio di ricerca sia **piccolo**, oppure che sia **polinomialmente visitabile in maniera deterministica**, in realtà siamo davanti ad un problema che non è di nostra competenza.

Ad esempio, nell'ordinamento di un vettore lo spazio di ricerca è esponenziale; mentre ordinare un vettore ci costa un tempo polinomiale con un algoritmo deterministico a nostra scelta (heap-sort, insertion sort, ecc..). Il problema di ordinamento di un vettore non rientra tra i problemi di interesse di chi fa Intelligenza Artificiale.

Stiamo quindi parlando di uno **spazio esponenziale non deterministicamente visitabile**; allora che forma avrà tale spazio?

Qualsiasi sia il problema, sicuramente avremo sempre davanti un certo **stato iniziale** (nodo).

Consideriamo l'esempio del cubo di Rubik: lo stato iniziale è la configurazione del cubo per come ci si presenta:



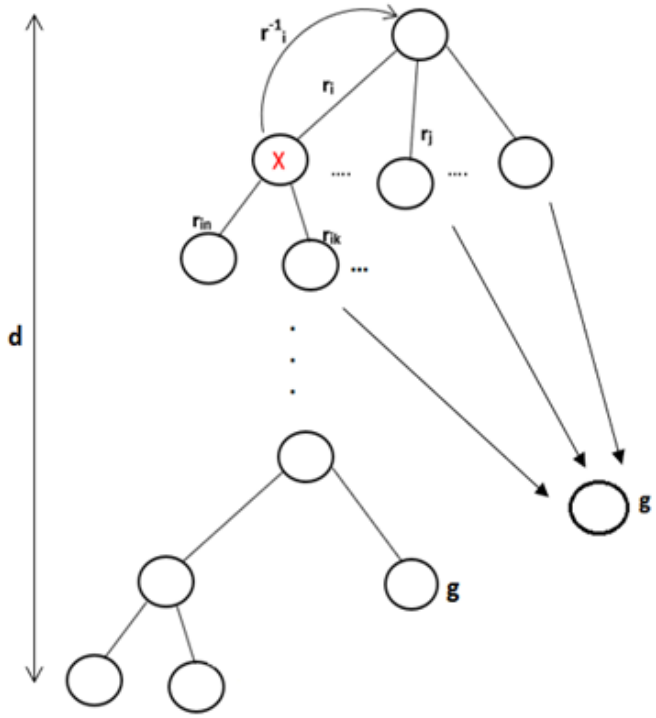
Possiamo riconoscere in questo caso anche lo stato finale, cioè la configurazione del cubo in cui tutte e sei le facce del cubo hanno colore omogeneo.

Definiremo lo stato finale come **stato goal g**.

Il cubo è in una certa configurazione iniziale, come possiamo immaginare la prossima configurazione?

Poiché nel cubo possiamo contare le facce, per ogni configurazione del cubo possiamo immaginare che le prossime configurazioni del cubo siano quelle che si riescono ad ottenere con una mossa elementare, ossia quella della rotazione di una faccia del cubo di 90°. Allora possiamo immaginare i prossimi stati come rotazione $\mathbf{r_i}$ di qualche tipo di una delle facce di 90°:

Immagine dello spazio di ricerca



Data una di queste configurazioni, essa può essere trasformata in qualcos'altro applicando una rotazione dello stesso modo.

Otteniamo in tal modo un **albero** (nei nostri spazi di ricerca immagineremo di avere sempre a che fare con alberi).

Questo oggetto può comunque essere compattato in un grafo: se ad esempio fossimo nella posizione **X** e considerassimo l'**anti-rotazione $\mathbf{r_i^{-1}}$** di r_i chiaramente si tornerebbe allo stato iniziale.

I modi di risolvere il cubo di Rubik sono tanti, ma la configurazione finale è una sola: si avranno quindi più path che confluiscono su un unico stato goal (quindi non si tratta di un albero ma di un **grafo**, poiché si hanno situazioni in cui le azioni sono **reversibili**).

Dal nostro punto di vista è molto più naturale **immaginare sempre e comunque lo spazio di ricerca come un albero**.

Questa scelta non è banale, perché, in realtà, la duplicazione che si può ritrovare nella forma ad albero dello spazio di ricerca potrebbe essere assai rilevante dal punto di vista dimensionale (potremmo avere un numero esponenziale di nodi che nel grafo corrispondono ad un solo nodo).

Tuttavia, le tecniche di visita sono molto meglio esposte se viste su un albero.

Quanto è grande un albero di questo tipo?

La dimensione dell'albero dipende da quanto è lontano il goal dalla radice, cioè quanto è profondo l'albero; quindi potremmo avere anche situazioni in cui ci siano dei **nodi più profondi rispetto al livello dei goa**l (come si nota in figura).

Proprio nell'esempio del cubo di Rubik, esso può essere risolto in diversi modi (diversi goal), ma la sequenza di azioni può essere più o meno lunga.

Quindi, dobbiamo in qualche modo fissare le idee sulla dimensione dell'albero: supporremo di avere a che fare (quasi) sempre con un albero avente numero di livelli (altezza) pari a **d**.

C'è poi un altro fattore importante che è il numero di figli che ogni nodo può avere: notiamo che, nelle situazioni usuali, questo valore può cambiare con la configurazione corrente (a seconda del nodo in cui ci troviamo, è possibile che il numero di figli vari) e questo porta in situazioni di difficoltà).

Allora, facendo una semplificazione, supporremo che, mediamente, il **numero di figli** di **ogni** nodo sia pari a "**b**" (**branching factor**) e che l'**altezza** dell'albero sia pari a "**d**" (**profondità**). Infine, assumendo che l'albero sia **completo**, otteniamo che :

b = branching factor **nodi dell'albero** = $\frac{b^{d+1}-1}{b-1}$ **d** = altezza

Poiché lo spazio è limitato, memorizzare di volta in volta ogni configurazione dell'albero può essere un problema. Per tale motivo, gli algoritmi di I.A. generano la configurazione attuale a partire da quella precedente, memorizzando di fatto solo l'ultima.

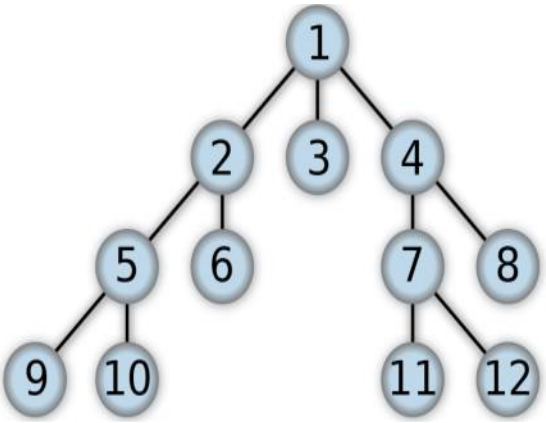
Con visita di un albero intendiamo la generazione di una sequenza di nodi (path), che parte dalla radice e arriva fino al nodo obiettivo (goal). Di seguito, vedremo alcuni algoritmi di ricerca di tipo **Blind** (anche detti "non informati"), ossia algoritmi che non richiedono nessuna conoscenza specifica relativa al problema (approccio brute-force).

Pseudocodice Algoritmo di ricerca generico

1. L={root}
2. If L=∅ stop with failure (FALLIMENTO)
3. Else x=**extract_node**(L)
   - 3.1. If goal(x) then return x and **path**(x) [SUCCESSO]
   - 3.2. Else **insert**(**sons**(x), L)
   - 3.3. Go to 2

Quella appena vista è la struttura generica di un algoritmo di ricerca. Il tipo di visita dell'albero viene determinato specializzando i vari metodi evidenziati.

Visita in ampiezza (breath-first search)



Questo algoritmo di ricerca visita i nodi dell'albero partendo dalla radice, e proseguendo, uno per volta, su ogni livello dell'albero, partendo da sinistra fino destra (LEFT Major Order).

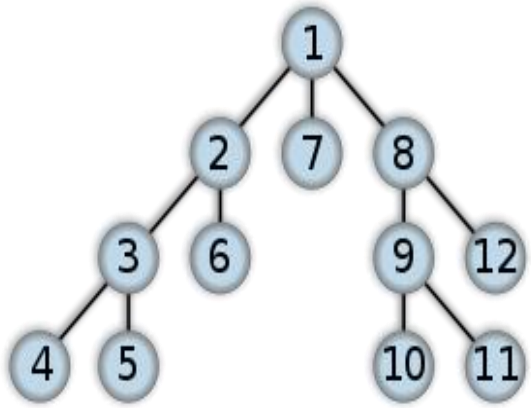
Nel disegno, i numeri in ogni nodo indicano l'ordine di visita di ognuno

Possiamo implementarlo tramite la struttura dell'algoritmo generico vista prima, specializzando, in particolare, due metodi:

- **extract_node** viene specializzato in extract_head, che estrae la testa della lista dei nodi
- **insert** viene specializzato in **queue** from LEFT to RIGHT, che scorre I nodi da sinistra a destra

1. L={root}
2. If L=∅ stop with failure (FALLIMENTO)
3. Else x=**extract_head**(L)
   - 3.1. If goal(x) then return x and **path**(x) [SUCCESSO]
   - 3.2. Else **queue** from LEFT to RIGHT (**sons**(x), L)
   - 3.3. Go to 2

Visita in profondità (depth-fisrst search)



Questo algoritmo di ricerca visita i nodi dell'albero partendo dalla radice, e proseguendo, un nodo per volta, sul lato sinistro dell'albero, scendendo di livello, per poi passare al lato destro.

Nel disegno, i numeri in ogni nodo indicano l'ordine di visita di ognuno

Possiamo implementarlo tramite la struttura dell'algoritmo generico vista prima, specializzando, in particolare, due metodi:

- **extract_node** viene specializzato in **extract_head**, che estrae la testa della lista dei nodi
- **insert** viene specializzato in **queue** **from RIGHT to LEFT**

1. L={root}
2. If L=∅ stop with failure (FALLIMENTO)
3. Else x=**extract_head**(L)
   - 3.1. If goal(x) then return x and **path**(x) [SUCCESSO]
   - 3.2. Else **queue** **from RIGHT to LEFT** (**sons**(x), L)
   - 3.3. Go to 2

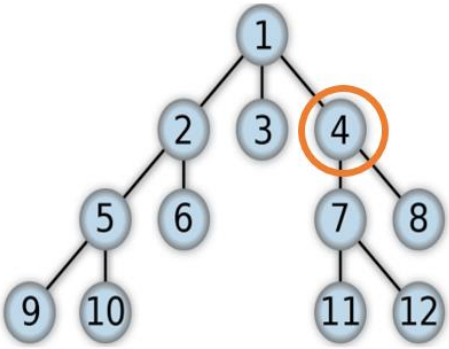
Analisi Complessità

Per i nostri scopi, analizzeremo due tipi di complessità per gli algoritmi appena visti:

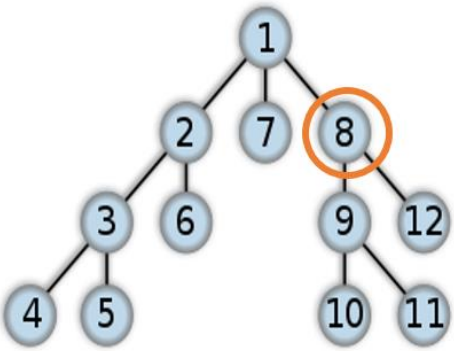
- **Complessità spaziale:** occupazione di memoria richiesta per l'esecuzione di un algoritmo
- **Complessità temporale:** tempo richiesto per l'esecuzione di un algoritmo

Algoritmo	Complessità spaziale	Complessità temporale
Breadth-first	b^d	b^d
Depth-first	b*d	b^d

Stando ai dati, il Depth-first è migliore del Breadth-first in termini di complessità computazionale. In realtà, quando ci troviamo a dover raggiungere un determinato nodo (goal), il Breadth-first si comporta meglio. Di seguito, ne vediamo un esempio:



Breadth - Search

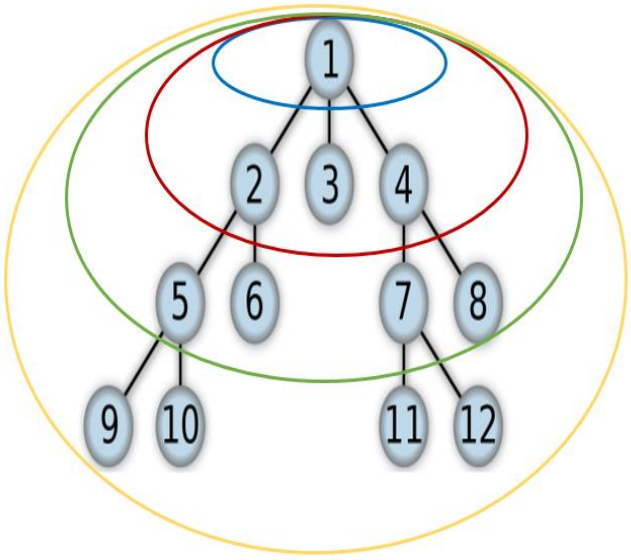


Depth - Search

Ipotizzando che il nodo goal sia quello evidenziato, si può notare come il Breath-Search lo raggiunga in meno tempo rispetto al Depth-Search. Questo poiché il Depth-Search scende per tutta la profondità dell'albero, partendo da sinistra, di fatto ignorando i nodi adiacenti. Se il goal si trova, come in questo caso, al primo livello, l'albero verrà comunque visitato (inutilmente) per tutta la sua profondità.

Esiste, tuttavia, un algoritmo che unisca i vantaggi di entrambi?

Iterative deepening



In questo algoritmo, l'albero viene visitato secondo la politica del Depth-Search, ma con alcune modifiche. Innanzitutto, alla prima iterazione, viene considerata la sola radice (Iterazione 1). Se il goal non è stato trovato, allora si scende di livello, aggiungendo i nodi sottostanti (Iterazione 2). Questo passaggio viene ripetuto di volta in volta finchè il goal non viene raggiunto. In questo modo, è possibile limitare la profondità dell'albero da esaminare, sfruttando i vantaggi di entrambi gli algoritmi visti in precedenza.

Parlando di complessità, ipotizziamo che il goal si trovi al livello K dell'albero. Avremo quindi:

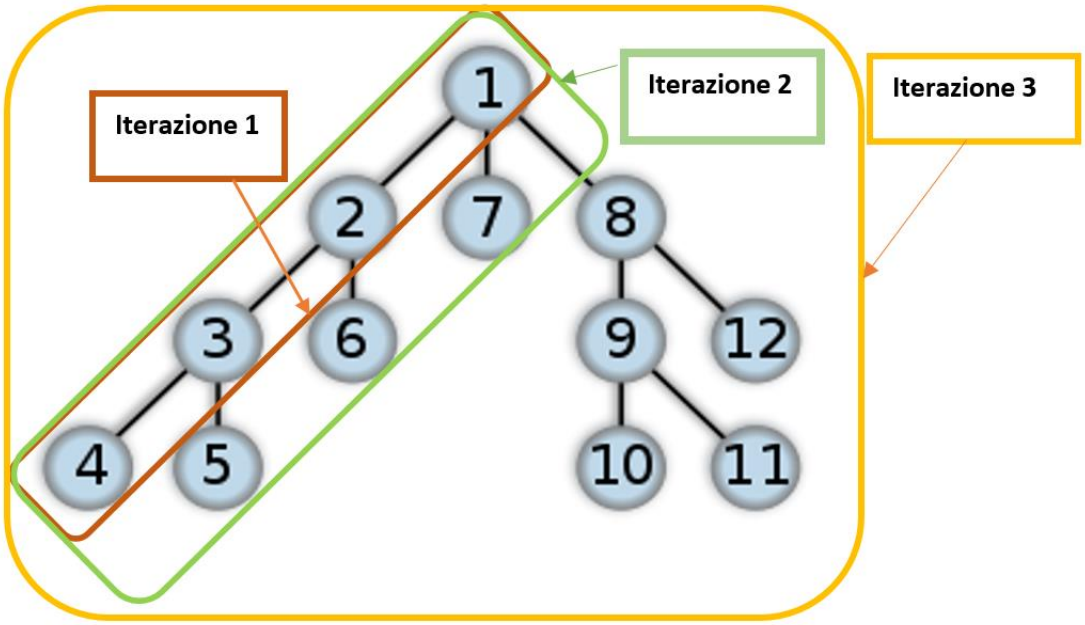
$$C_{D-F}^{k} = Complessità\ Depth - Search\ al\ livello\ k$$

$$Complessità\ Iterative\ Deepening = C_{D-F}^{k} + \sum_{j=1}^{k-1} \frac{b^{j+1} - 1}{b - 1}$$

Fattore di inefficienza

Il fattore di inefficienza ($\approx b^d$) indica il costo in più da sostenere per ottenere la soluzione ottimale. Tale algoritmo, di fatto, è il migliore fra gli algoritmi di tipo Blind.

Iterative broadening



In questo algoritmo viene seguita una procedura simile all'Iterative Deepening, con la differenza che non viene limitata la profondità dell'albero, bensì la sua ampiezza (in sintesi, il numero di figli per nodo considerati). Notiamo che alla prima iterazione, viene considerato un solo figlio per nodo, partendo da sinistra. Alla seconda iterazione, l'albero viene espanso considerando due figli per nodo, e così via anche per la terza iterazione.

A conti fatti, questo algoritmo si dimostra inefficiente, poiché condivide i difetti di Depth- search e di Breadth-search. Nonostante ciò, in alcuni casi si dimostra più efficiente persino di Iterative Deepening, ad esempio quando i nodi goal sono sparsi nello spazio di ricerca.

Questo ci dimostra, sostanzialmente, che è la struttura dello spazio di ricerca a suggerirci le azioni migliori da intraprendere per ottenere performance migliori.

Gli algoritmi visti finora possono essere applicati anche su grafi (in quanto gli alberi, di per sé, sono essi stessi grafi, orientati e non ciclici). In questo caso, vengono considerati i vari sottoalberi ottenuti a partire da un nodo del grafo (che ne diverrà la radice), escludendo il resto. Ovviamente, ad ogni iterazione bisogna evitare di scansionare tali sottoalberi nuovamente.

Confronto performance B-F e D-F

$$\frac{P_{B-F}}{P_{D-F}} \approx \frac{b+1}{b}$$

Il rapporto varia al variare del parametro b (branching factor).

Vediamo un esempio:

b	$\boldsymbol{\frac{P_{B-F}}{P_{D-F}}}$
2	1.5
3	1.3
5	1.2
10	1.1
25	1.04

Notiamo come al crescere di b, i due algoritmi tendano a somigliarsi

Confronto performance I-D e D-F

$$\frac{P_{I-D}}{P_{D-F}} \approx \frac{b+1}{b-1}$$

Stavolta, per un b piccolo, iterative deepening si dimostra più lento di D-F.

Nonostante ciò, I-D rimane un algoritmo adattabile a qualunque situazione.

Lezione 3 05/10/2018

Quando in un albero cerchiamo un nodo, scorrendo gli altri (senza altre informazioni), parliamo di **Ricerca a Livello** (Level Wise Search), ad esempio con gli algoritmi Blind.

Quando invece possediamo informazioni aggiuntive, parliamo di **Ricerca a Metalivello**.

Più informazioni sono in nostro possesso, più l'algoritmo sarà efficiente. Va però osservato che tale operazione costa di più, per cui, prima di aggiungere informazioni, bisogna valutare se il costo aggiuntivo è minore del vantaggio della ricerca a metalivello rispetto alla ricerca a livello.

Vediamo alcuni problemi tipici dell'IA:

- **Scacchi:** lo spazio di ricerca è immenso. I gran maestri operano maggiormente un approccio a metalivello: tramite l'esperienza, riescono a scegliere la mossa migliore rispetto al contesto, di fatto scegliendone una fra le innumerevoli configurazioni possibili, ed escludendo le altre. Una macchina, al contrario, sceglie la mossa migliore fra tutte le configurazioni possibili (milioni di milioni), e ciò è ovviamente impossibile per una mente umana. Ciò ci fa comprendere la principale differenza fra la mente umana e la macchina: la prima "ragiona" secondo una logica a metalivello, mentre la seconda opera secondo la logica a livello.

- **Backgammon:** in questo caso, è stato possibile codificare un'euristica abbastanza simile al ragionamento umano. Ciò è possibile in quanto lo spazio di ricerca è molto minore rispetto agli scacchi (il gioco è più semplice)

Block World

È una stilizzazione dei problemi di planning. Abbiamo un tavolo, abbastanza ampio, e un numero indefinito di blocchi, distinguibili l'uno dall'altro, disposti su di esso:

Azioni consentite:

- **Muovere un blocco (sul Tavolo):** move_to_t (B, p), dove p è la posizione di destinazione
- **Muovere un blocco su una posizione a partire da un'altra:** move (B, p1, p2)

Proviamo ad implementare una soluzione. Ipotizziamo di voler arrivare a questa configurazione:

Serie di mosse

Move_to_t (B, A)

Move (A, T, C)

Move (B, T, A)

Lezione 4 15/10/2018

Ricerca euristica

La ricerca esaustiva non è praticabile in problemi di complessità esponenziale. Noi usiamo conoscenza del problema e l'esperienza per riconoscere i cammini più promettenti.

La conoscenza euristica (dal greco "eureka") aiuta a fare scelte "oculate", in quanto scegliamo i nodi con un livello di "desiderabilità" maggiore. Ciò non evita la ricerca ma la riduce, e consente in genere di trovare una buona soluzione in tempi accettabili. Tale tecnica, sotto certe condizioni, garantisce completezza e ottimalità. Nella pratica, facciamo uso di una **funzione di valutazione ($f: n \rightarrow \mathbb{R}$)**, che assegna ad ogni nodo un valore, sulla base di quanto esso sia conveniente per i nostri scopi. Tale funzione si applica al nodo, ma dipende solo dallo stato. Vediamo ora alcuni algoritmi euristici famosi, per completezza.

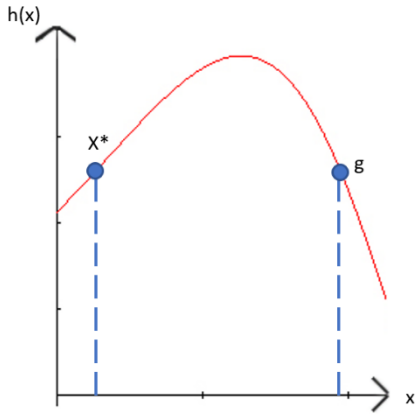
Hill Climbing

Il nome dell'algoritmo deriva dal fatto che ci spostiamo sulla zona di massima pendenza della funzione euristica (ci "arrampichiamo", per l'appunto). Di fatto è simile al D-F, ma presenta differenze in fase di inserimento dei nodi nella lista L. Nello specifico, questo algoritmo fa uso di una funzione euristica, f, che assegna ad ogni nodo un peso. Tale valore, associato ad ogni nodo, costituisce il parametro di inserimento: i nodi vengono aggiunti alla lista L in ordine, in base al loro peso calcolato tramite la funzione f:

$$insert(sons(x), L) \rightarrow insert\_ordered(sons(x), L, f)$$

Cosa succede se i figli di un nodo, chiamiamolo x*, hanno un valore (peso) peggiore del padre?

La situazione, schematizzando, appare come segue:



Come si può vedere, il nodo in esame, x*, ha un valore di funzione euristica (h(x)) più basso dei suoi figli, fino ad un certo punto. L'algoritmo, di fatto, si fermerebbe restituendo il nodo x, in quanto non trova valori migliori (è un minimo locale), ma non avremmo ottenuto il nodo cercato e ciò non è ovviamente ammissibile. Come fare, quindi, affinché l'algoritmo non si arresti quando si trova in questo tipo di problemi?

Possiamo tentare di andare oltre il valore minimo, peggiorando così la funzione euristica h(x). Ciò ha senso in questi casi, infatti si può notare come il nodo obiettivo (g, ossia **goal**) si trova esattamente dopo il picco della funzione euristica, dato dai figli di x*. Per cui, peggiorando inizialmente h(x), riusciremo a trovare, successivamente, il nodo obiettivo (è un valore minimo migliore di quello associato a x*).

Simulated Annealing

È una tecnica che va a migliorare il già analizzato algoritmo Hill Climbing. Essa fa uso della **randomizzazione:** anziché seguire i valori dati da h(x), preleviamo un nodo a caso dalla lista L:

1. Prendiamo una distribuzione di probabilità $p(t) = K * e^{-\lambda t}$, dove k è scelto in modo tale che inizialmente la probabilità di operare in modo random sia alta

2. Con probabilità $1 - p(t)$ →$x = estrai\_dalla\_testa(L)$

3. Con probabilità $p(t)$-> $x = estrai\_random(L)$

Il miglioramento rispetto a Hill Climbing Base è notevole, tuttavia esistono problemi dove tale aumento è minimo, ad esempio il SAT.

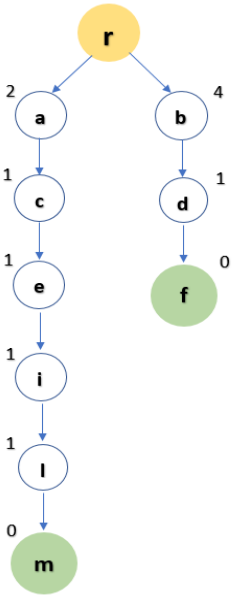
Best First

La ricerca best first è una strategia di ricerca informata in cui ad ogni passo viene espanso sempre il nodo migliore, quello più vicino all'obiettivo da raggiungere. Best first significa "prima il migliore". La ricerca best first è anche conosciuta come **ricerca greedy** o **ricerca golosa** (ricerca avida). Queste ultime due denominazioni hanno il pregio di mettere meglio in luce, come vedremo, anche alcuni handicap della ricerca best first. Come funziona un algoritmo di ricerca best first? Per valutare la qualità (best) dei nodi l'algoritmo si avvale di una funzione di conoscenza (funzione euristica). Ad esempio, può valutare la qualità dei nodi in base alla distanza aerea (in linea retta) dal nodo obiettivo senza tenere conto del cammino complessivo. La modifica all'algoritmo di ricerca è la seguente:

$$\boldsymbol{insert(sons(x), L)} \rightarrow \boldsymbol{insert\_sorted(sons(x), L, f)}$$

A*

L'algoritmo Best-first, appena visto, si dimostra ottimale in molti casi. Tuttavia, in certe situazioni, può avere comportamenti anomali. Vediamo un esempio:



Accanto ad ogni nodo c'è il valore associato dalla funzione euristica. Ipotizziamo di che i nodi f ed m siano i nodi obiettivo (goal). Applichiamo quindi l'algoritmo best-first:

Iterazione	Lista L
1	< r >
2	< a, b>
3	< c, b>
4	< e, b >
5	< i, b >
6	< l, b >
7	< m, b > → goal

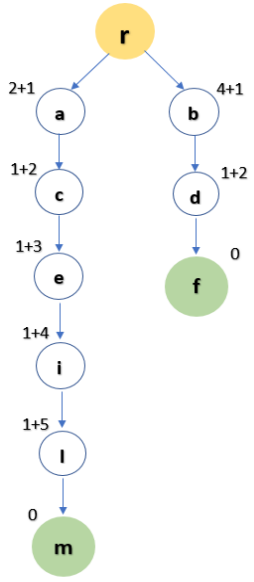
Notiamo come l'algoritmo sia rimasto "intrappolato" in un minimo locale. Il goal generato è un nodo obiettivo valido, ma non il migliore, in quanto il nodo m è più lontano dalla radice rispetto al secondo goal, f. Come risolvere?

Usiamo come funzione euristica la somma della funzione precedente e la profondità del nodo:

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

Otteniamo un best-first modificato. In particolare, tale variante del best-first prende il nome di A*.

Torniamo all'esempio precedente e applichiamo le modifiche:



Aggiungiamo ai valori di ogni nodo la sua profondità. Con questi nuovi pesi, otteniamo:

Iterazione	Lista L
1	< r >
2	< a, b>
3	< c, b>
4	< e, b >
5	< i, b >
6	< b, l >
7	< d, l >
8	< f, b > → goal ottimo

Stavolta l'algoritmo non è sceso troppo in profondità, trovando di fatto il **goal ottimo, ossia il goal più vicino alla radice.**

Normalmente, noi cercheremo i **goal ottimali, ossia goal validi ma non i migliori in assoluto.** Questo perché cercare i goal ottimi ha un costo elevato, il quale non è giustificato dal risultato ottenuto. La funzione h(x) ci da il numero di mosse di distanza dal nodo corrente al nodo ottimo. Notiamo che, nel caso del nodo b, tale valore sia sovrastimato rispetto agli altri. Tuttavia, anche modificandolo, l'algoritmo A* continua a funzionare.

Diamo una definizione preliminare. Sia h*(n) la distanza reale dal nodo n al goal ottimo, e sia h(n) una generica funzione euristica:

$$h \text{ è } ammissibile\ se\ \forall\ n, h(n) \leq h^*(n)$$

Da qui, possiamo affermare:

TEOREMA: Se h è ammissibile, allora A* restituirà un goal ottimo.

Facciamo qualche considerazione:

- Nell'esempio abbiamo supposto che la distanza fra i nodi sia unitaria, tuttavia ciò non è sempre valido (ad esempio se i nodi sono città). Se tale distanza non è unitaria, A* rimane valido? Se h(n) è ammissibile, allora A* restituirà un goal ottimo.

- Se $h(n) = 0, \forall\ n$ (ossia nulla per ogni nodo), allora A* si riduce ad un algoritmo Branch and Bound

- Il difetto di A* sta nel fatto che lo spazio di ricerca può diventare esponenziale, riducendo A* ad un B-F

È possibile strutturare un algoritmo che restituisca l'ottimo in uno spazio al più polinomiale?

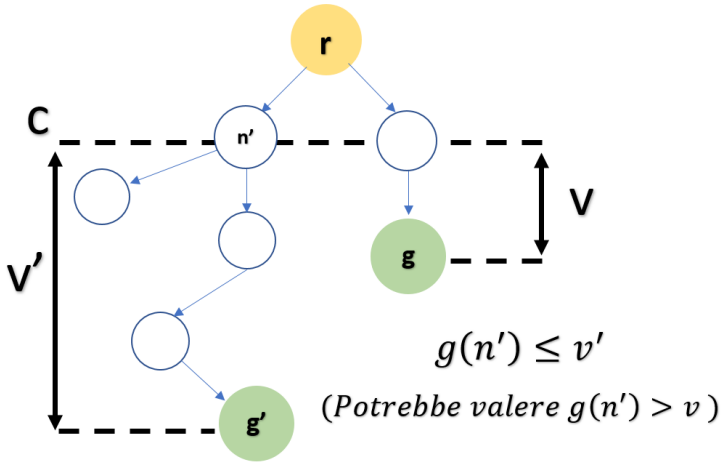
Possiamo usare una variante, Iterative-Deepening A*. In tale variante, lo spazio di visita dell'albero viene incrementato di un valore imposto dalla funzione g(n), e non di 1 come in I-D.

Iterative Deepening A*

1. $c = 1$
2. $L = < nodi\ iniziali >,\ c' = +\infty$ (normalmente L=< radice >, tuttavia in alcuni casi può essere popolata da più nodi; c' rappresenta il valore del prossimo cutting level)
3. **IF** $L =< vuoto >$ (L=∅) **AND** $c' = +\infty$ **THEN** ***STOP*** $(Fallimento)$
4. **IF** $L =< vuoto >$ ***AND*** $c \neq +\infty$ **THEN** $c = c'$, **GO TO 2**
5. **ELSE** $n = \boldsymbol{extract\_head}(L)$
6. **IF goal(**n**) THEN RETURN** n **AND path(**n**),** STOP (Successo) **ELSE** $n = \boldsymbol{extract\_head}(L)$
7. **ELSE** $n' = generico\ figlio\ del\ nodo\ n$
8. **IF** $g(n') \leq c$ **THEN** $\boldsymbol{insert}_{\boldsymbol{head}(n',L)}$
9. **ELSE** $c' = \mathbf{mi\,n}(c', g(n'))$
10. **GO TO 3**

Perché al passo 9 il cutting level c è preso come MIN?

Ipotizziamo di avere 2 goal, uno a distanza v, ed uno a distanza v':



Se non modificassimo c', l'algoritmo A* andrebbe ad esaminare l'albero troppo in profondità, restituendo un goal non ottimo (g è più vicino alla radice di g').

In questo modo, il valore del cutting level risulta essere bilanciato (né troppo grande, né troppo piccolo).

Lezione 5 19/10/2018

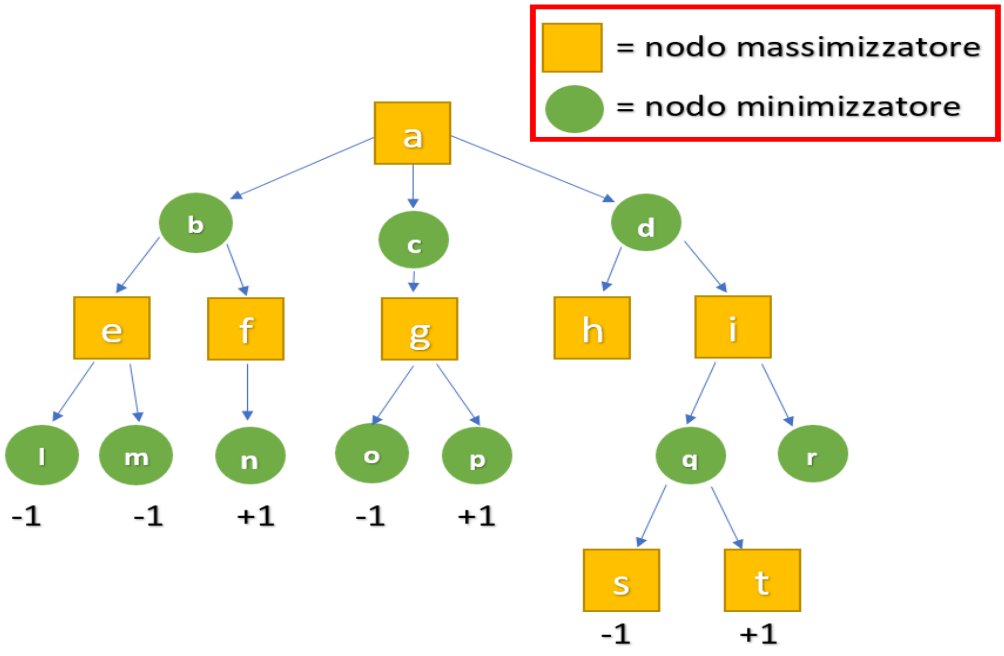
Ricerca con avversario

Finora, gli algoritmi di ricerca visti si limitavano a passare da una configurazione valida del problema ad un'altra. In questi casi, è presente un solo attore.

Aggiungiamo ora un secondo attore, un avversario. La configurazione del problema cambierà, quindi, in relazione alle mosse eseguite da 2 attori. Dobbiamo quindi distinguere sull'albero di ricerca le mosse del nostro agente e dell'avversario.

Useremo un modello che trasforma lo spazio di ricerca di un gioco con avversario in un albero, ossia il Minimizzatore-Massimizzatore (Min-Max)

Min-Max



Questo non rappresenta l'albero completo di gioco. È una configurazione di partenza (non per forza quella iniziale), dalla quale iniziare a decidere. I due giocatori sono il **massimizzatore** (nodi quadrati) e il **minimizzatore** (nodi rotondi). All'inizio, il problema del massimizzatore è la scelta di uno dei nodi figli (mossa), scegliendo fra i path m1, m2, m3. Il turno passa quindi al minimizzatore (nodi rotondi) che a sua volta farà la sua mossa partendo dal nodo scelto dal massimizzatore nel turno precedente. Le mosse operano in sequenza finché non ci troviamo in un nodo foglia, ad ognuno dei quali è associato un valore (positivo o negativo):

- **Se il valore è positivo, vince il massimizzatore**
- **Se il valore è negativo, vince il minimizzatore**

Notiamo che scegliere il path m1 non è una buona mossa, in quanto con alta probabilità finiremmo su una configurazione del gioco con nodo foglia negativo (essendo noi il massimizzatore, avremmo perso).
Facciamo alcune considerazioni:

- Se avessimo la possibilità di espandere l'albero di ricerca, per il nostro giocatore il gioco sarebbe vinto (se esiste una serie di mosse he, per qualsiasi mossa avversaria, portano alla vittoria; tale serie di mosse è detta **strategia vincente**).

- Noi ipotizziamo di avere a che fare con un avversario **perfettamente razionale**, ossia che sceglie sempre le mosse migliori, ma non sempre è così (comportamento inatteso)

Per giochi non banali non è possibile espandere completamente l'albero di ricerca. Ciò è evidente soprattutto per giochi con un tempo limitato per ogni mossa (es. scacchi). Inoltre, molti giochi non forniscono informazioni complete (es. tresette), ossia non ci è subito chiaro lo schema (negli scacchi ad esempio la scacchiera è visibile fin da subito, nel tresette le informazioni a disposizione sono incomplete). In questo caso parliamo di **Incomplete** (o **Complete**) **Information Games.**

Va poi detto che molti giochi sono **strategicamente caotici**, ossia lo schema di gioco varia parecchio cambiando gioco (es. tresette e tresette a perdere). Vi sono poi dei giochi nei quali è possibile fare delle mosse disastro (**crash moves**), ossia mosse che fanno salire parecchio il valore della funzione euristica (portano chi la fa in situazione di immediato svantaggio). La struttura adattata per 2 giocatori può essere usata (con opportune modifiche non banali) per giochi a più giocatori.

Tornando al Min-Max, supponiamo di avere un gioco a 2, ad informazione perfetta, ed una funzione che valuta le configurazioni finali (ossia senza altre mosse possibili) che possono essere vincenti per il massimizzatore, per il minimizzatore o che portano ad un pareggio (se previsto).

Diamo quindi un valore numerico ad ogni nodo:

- **Configurazione vincente per il massimizzatore:** 1
- **Configurazione vincente per il minimizzatore:** -1
- **Pareggio:** 0

Da notare che non tutte le configurazioni finali sono allo stesso livello.

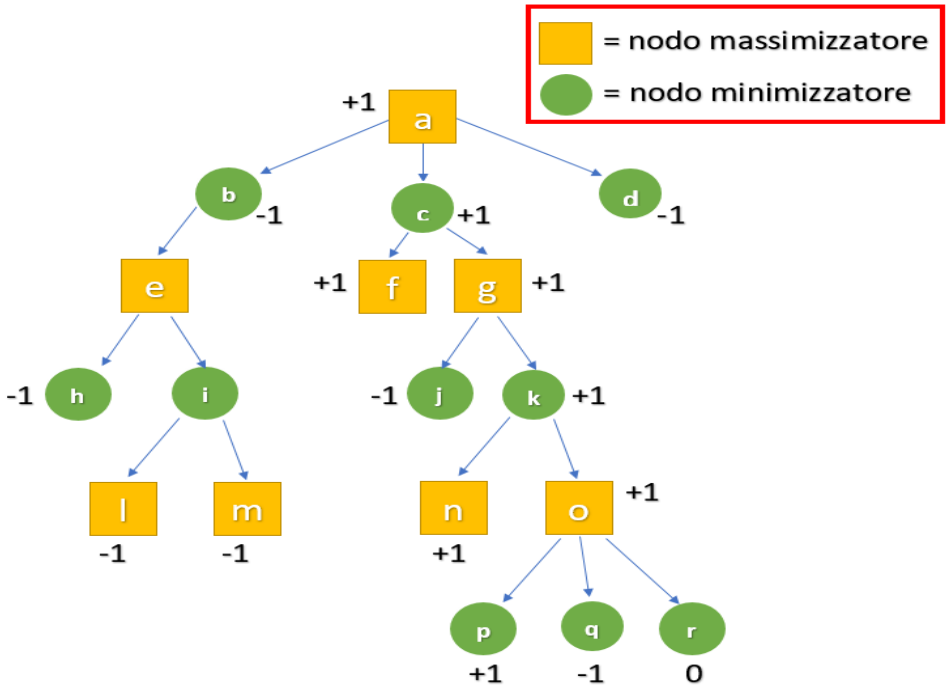
L'algoritmo consiste in due fasi:

- **Turno del massimizzatore**: sceglie il figlio con valore massimo
- **Turno del minimizzatore**: sceglie il figlio con valore minimo

Se ci troviamo in una configurazione (non finale, quindi un nodo non foglia) con valore positivo, esiste una serie di mosse che portano il massimizzatore a vincere (configurazione vincente). In modo analogo, se negativo esiste una configurazione vincente per il minimizzatore.

In sintesi, se su un nodo abbiamo:

- **Valore positivo**: esiste una serie di mosse che porta il massimizzatore a vincere
- **Valore negativo**: esiste una serie di mosse che porta il minimizzatore a vincere



Introduciamo ora una funzione di valutazione della configurazione corrente, $e : N \rightarrow [-1, 1]$.

Tale funzione, dato un nodo, associa 1 se quella configurazione è vantaggiosa per il massimizzatore, -1 se è vantaggiosa per il minimizzatore.

A questo punto, supponendo di trovarci in una configurazione intermedia (nodo non foglia) e di poter "vedere" oltre essa, l'algoritmo è chiaro:

- Espandere l'albero il più possibile
- Assegnare ad ogni nodo il valore di e

Se usiamo il D-F, e siamo nella configurazione migliore (nodi con *) allora possiamo evitare di espandere ulteriormente l'albero (ciò è importante specie se si pensa che nella realtà il branching factor può essere parecchio elevato). Quanto deve essere difficile da calcolare la funzione e?

Non esiste una risposta specifica. Ad esempio, nel gioco degli scacchi diamo un peso diverso ad ogni tipo di pezzo, ottenendo:

$$w = peso\ dei\ bianchi \qquad e(n) = \frac{w-b}{w+b}$$
$$b = peso\ dei\ neri$$

Se e(n) si avvicina di più ad 1, siamo in una configurazione migliore per il massimizzatore, mentre se si avvicina ad -1 è una configurazione migliore per il minimizzatore. È una buona funzione? Dipende da molti fattori, come il tipo di gioco. Riassumiamo il tutto in un algoritmo, di seguito.

Min-Max Depth-First

$L = Lista\ dei\ nodi$

$V_X = peso\ associato\ dalla\ funzione\ e\ al\ nodo\ x$

1. $L = \{n\}$
2. $x = \boldsymbol{pop}(L)$

   $\boldsymbol{if\ has\_value}(x)\ \boldsymbol{then}$
3. $\boldsymbol{if\ has\_value}(x) == V_x\ \boldsymbol{then}$

   $\boldsymbol{let}\ p = \boldsymbol{be\_parent}(x)$ and $\boldsymbol{has\_value}(p) == V_p$

   $\boldsymbol{if\ minimizer}(p)\ \boldsymbol{then}$ *//p è un nodo del minimizzatore*

   $V_p = \mathbf{min}(V_p, V_x)$

   $\boldsymbol{else}$ *//p è un nodo del massimizzatore*

   $V_p = \mathbf{max}(V_p, V_x)$

   $\boldsymbol{go\ to}\ 2$
4. $\boldsymbol{if}(\boldsymbol{not}(\ \boldsymbol{has\_value}(x))$ **and** $(\boldsymbol{terminal}(x)\ \boldsymbol{or}\ \boldsymbol{not}(\boldsymbol{expandable}(x))\ \boldsymbol{then}$

   $V_x = e(x)$ *//x è una foglia senza valore associato*

   $\boldsymbol{go\ to}\ 2$
5. $\boldsymbol{if}(\boldsymbol{not}(\ \boldsymbol{has\_value}(x))$ **and** $\boldsymbol{not}(\boldsymbol{terminal}(x))\ \boldsymbol{and}\ \boldsymbol{expandable}(x))\ \boldsymbol{then}$

   //x è un nodo intermedio senza valore associato

   $\boldsymbol{if\ maximizer}(x)\ \boldsymbol{then}$

   $V_X = -\infty$

   $\boldsymbol{else}$

   $V_X = +\infty$

   $\boldsymbol{insert\_top}(\boldsymbol{sons}(x), L)$

   $\boldsymbol{go\ to}\ 2$

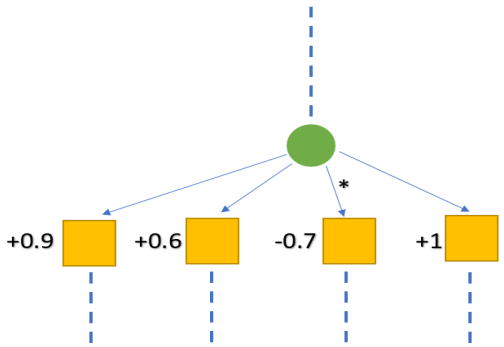
Introduciamo ora una classificazione per i nodi:

- **Nodo quieto**: è un nodo con un valore di funzione euristica simile (vicino) al valore dei suoi predecessori. In caso di giochi non caotici, trovandoci in questo tipo di nodi possiamo evitare di espandere ulteriormente l'albero (ovviamente, se siamo in vantaggio)

- **Nodo tattico:** è un nodo con un valore di funzione euristica che varia molto dai suoi predecessori. Stavolta, in caso di situazione vantaggiosa in giochi non caotici, conviene espandere l'albero

Estensione singolare

Supponiamo di essere in una configurazione in cui muove l'avversario (il suo turno), e di giocare col massimizzatore:

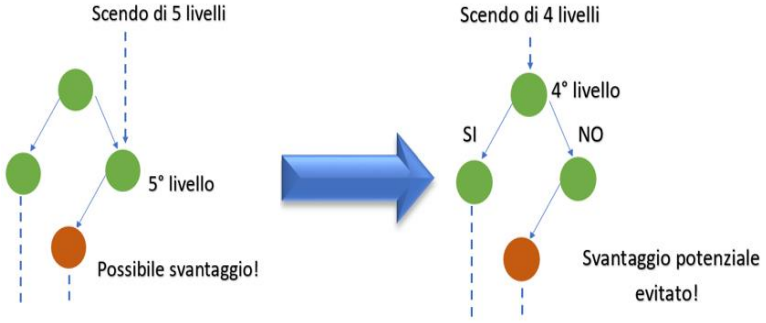


In questo caso il minimizzatore sceglierà sicuramente il path contrassegnato con *, poiché gli darebbe molto vantaggio. Questa è l'**estensione singolare,** poiché qualunque euristica usi l'avversario, egli sceglierà sicuramente il path col migliore vantaggio (evitando di espandere gli altri).

Tutto ciò, supponendo che l'avversario scelga sempre la mossa migliore

Effetto orizzonte

Supponiamo di espandere l'albero per un numero prestabilito di livelli. L'avversario muove, e scendiamo di n livelli. Supponiamo di trovarci, dopo tale mossa, in una situazione di svantaggio. Come mai è successo?



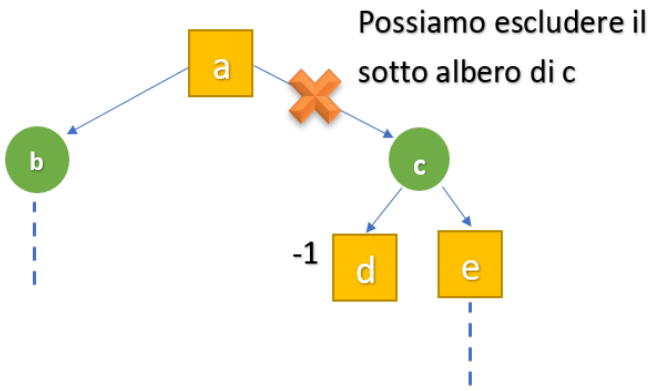
Ciò accade poiché non abbiamo "visto" il vantaggio potenziale dell'avversario ai livelli successivi, poiché il nostro "orizzonte" era di soli n livelli, e non abbiamo visto la "minaccia" al livello n+1. Come procedere? Supponendo di potersi muovere massimo di 5 livelli, mi muovo invece di 4 livelli ed adotto un'altra euristica per espandere l'albero. Questo ci permette di "tenere di riserva" qualche risorsa (in questo caso un livello) per evitare di cadere in situazioni svantaggiose. Tale tecnica è detta **ricerca secondaria.** Un'alternativa è l'uso di una **pseudo-randomizzazione**, scegliendo fra le mosse che danno maggior vantaggio, secondo l'euristica secondaria.

Lezione 6 22/10/2018

Pruning Alfa-Beta

Il **pruning** ("potatura", poiché tagliamo rami dell'albero) **alfa-beta** è un algoritmo di ricerca che riduce drasticamente il numero di nodi da valutare nell'albero di ricerca dell'algoritmo Min-Max. Viene comunemente usata nei programmi di gioco automatico per computer, per giochi a turni a due o più giocatori (Tris, Go, Scacchi ecc.), e consiste nel terminare la valutazione di una possibile mossa non appena viene dimostrato che è comunque peggiore di una già valutata in precedenza: è una ottimizzazione sicura, che non modifica il risultato finale dell'algoritmo a cui viene applicata.

Supponiamo di giocare col massimizzatore, e di trovarci in questa situazione:



Supponendo di trovarci nel nodo a, quale nodo scegliere per la prossima mossa? Ovviamente il nodo b, in quanto scegliendo c il minimizzatore sceglierebbe il nodo d e vincerebbe (vale -1). Tale scelta viene effettuata indipendentemente dal valore di c e dei suoi figli, ergo non c'è alcun bisogno di espandere il sotto albero del nodo c, e lo possiamo escludere. Il taglio del sotto-albero si può fare sotto l'ipotesi di una buona euristica, e se la sua eliminazione non influenza la scelta delle mosse (in questo caso, eliminare il sotto-albero di c non influenza in alcun modo la scelta del massimizzatore).

Vediamo una regola generale:

Sia n un nodo maximizer, e sia s fratello di n. Sia V_s il valore di s. Valga inoltre il seguente path:

$$P_0, \dots\dots\dots\dots, P_k$$

I nodi di ordine pari sono massimizzatori, quelli di ordine dispari sono minimizzatori.

Prendiamo dalla sequenza determinati nodi:

$$P_0, \dots\dots\dots., P_k\ (nodo\ n) \rightarrow \{P_i\}\ \ i = 2j + 1, \qquad con\ \ 0 \leq j \leq \frac{k}{2} - 1$$

Sia n' un fratello di un P_i, tale che $V_{n'} > V_S$. Se esiste n'→prune(n')

Il Pruning alfa-beta si basa su due valori, detti appunto alfa e beta che rappresentano, in ogni punto dell'albero, la posizione migliore e peggiore che è possibile raggiungere.

α è il punteggio minimo che A può raggiungere, a partire dalla posizione in esame; all'inizio dell'algoritmo viene posto a -∞. Durante il calcolo, α coincide con il valore della peggiore mossa possibile attualmente calcolata per A.

β è il punteggio massimo che B può raggiungere a partire dalla stessa posizione; all'inizio dell'algoritmo viene posto a +∞. Durante il calcolo, β coincide con il valore della miglior mossa possibile attualmente calcolata per B.

La ricerca procede come una normale ricerca min-max, in cui però i valori di α e β per ogni nodo vengono aggiornati man mano che la ricerca si approfondisce. Se durante la ricerca, per un dato nodo α diventa maggiore di β, la ricerca al di sotto di quel nodo cessa e il programma passa ad un altro sottoalbero, perché la posizione di quel nodo non può essere raggiunta durante il gioco normale (cioè da quella posizione in poi, A perderebbe inevitabilmente anche se giocasse per vincere, il che in un gioco competitivo è assurdo, e certamente non è il risultato che vogliamo). Si dice che il sottoalbero corrispondente al nodo con α e β "invertiti" viene potato, da cui il nome dell'algoritmo stesso. Vediamo l'algoritmo:

1. $L = \{n\}$
2. $\boldsymbol{Let}\ x = \boldsymbol{first}(L)$

   $\boldsymbol{if}\ x = n\ \boldsymbol{and}\ \boldsymbol{has\_value}(x)\ \boldsymbol{then}\ \ \boldsymbol{return}\ x$
3. $\boldsymbol{if}\ \boldsymbol{has\_value}(x) = V_x\ \boldsymbol{then}\quad \boldsymbol{let}\ \ p = parent(x)$

   $\boldsymbol{else\ go\ to}\ 5$

   $\boldsymbol{if}\ \boldsymbol{minimizer}(p)\ \boldsymbol{then}$

   $\boldsymbol{let}\ \alpha = \mathbf{max}(valori\ dei\ fratelli\ di\ p\ e\ dei\ nodi\ minimizzatori\ antenati\ \ di\ p)$

   $\boldsymbol{if}\ V_X \leq \alpha\ \ elimina\ p\ ed\ i\ suoi\ discendenti\ da\ L\ (prune)$

   $\boldsymbol{else\ if}\ \boldsymbol{maximizer}(\mathbf{p})\ \boldsymbol{then}$

   $\boldsymbol{let}\ \beta = \min(valori\ dei\ fratelli\ di\ p\ e\ dei\ nodi\ massimizzatori\ antenati\ di\ p)$

   $\boldsymbol{if}\ V_x \geq \beta\ elimina\ p\ ed\ i\ suoi\ discendenti\ (prune)$
4. $\boldsymbol{if}\ p\ \boldsymbol{is\ not\ pruned\ then}$

   $\boldsymbol{if}\ \boldsymbol{minimizer}(p)\ \boldsymbol{then}\ V_p = \mathbf{min}(V_p, V_x)$

   $\boldsymbol{else}\ V_p = \mathbf{max}(V_p, V_x)$

   $\boldsymbol{go\ to}\ \mathbf{2}$
5. $\boldsymbol{if}\ \boldsymbol{not}(\boldsymbol{has\_value}(x))\ ed$ è $un\ terminale\ o\ non\ possiamo\ espanderlo$

   $V_x = e(x)$

   $\boldsymbol{go\ to}\ \mathbf{2}$
6. $\boldsymbol{else\ if}\ \boldsymbol{maximizer}(p)\ \boldsymbol{then}\ V_x = -\infty$

   $\boldsymbol{if}\ \boldsymbol{minimizer}(p)\ \boldsymbol{then}\ V_x = +\infty$

Quanto costa l'algoritmo?

- **Caso peggiore**: non possiamo applicarlo e quindi non c'è risparmio
- **Caso migliore**:
  - **complessità senza alfa-beta:** $o(b^d)$
  - **complessità con alfa-beta:** $\approx 2 * b^{\frac{d}{2}}$

Come adattare queste tecniche a giochi con più di 2 giocatori? Abbiamo due casi:

- **Tutti sono avversari**: supponiamo di essere il massimizzatore. Allora il minimizzatore sarà la combinazione delle mosse degli avversari
- **Non tutti sono avversari**: in questo caso c'è un giocatore (o più giocatori) "aiutante", cioè che aiuta un altro giocatore a vincere tramite le sue mosse. In questo caso, i giocatori della stessa "squadra" useranno entrambi il minimizzatore o il massimizzatore.

Lezione 7 26/10/2018

Teoria dei giochi

La teoria dei giochi è stata sviluppata alla fine dell'800, ma ebbe successo con un libro, "Game Theory and Economic Behaviour", al quale partecipò Von Neumann. Tale disciplina è una branca della matematica che si occupa dei sistemi multi-agente.

Ogni agente è:

- **self-interested**: ha interesse solo della propria utilità
- **rational**: ogni sua decisione è presa razionalmente (senza alcuna influenza esterna)

I **giochi** si suddividono in 4 tipi:

- **Giochi Strategici:** 2 giocatori (o più) che possono fare una sola mossa. Il gioco è ad informazione completa.
    - **Esempio:** dilemma del prigioniero

- **Giochi Estensivi:** Simili ai giochi strategici, ma stavolta i giocatori hanno a disposizione più mosse
    - **Esempio:** scacchi

- **Giochi Ripetuti:** Può avere base strategica o estensiva, indifferentemente. Un gioco di questo tipo viene ripetuto più volte, incrementando così man mano la conoscenza dei giocatori sulle tattiche avversarie.

- **Giochi Cooperativi:** in questo tipo di giochi è possibile la cooperazione fra più giocatori, che formano coalizioni (squadre, gruppi) al fine di raggiungere un obiettivo

Noi in particolare ci soffermeremo su quest'ultima tipologia di giochi, analizzata nel seguito.

Giochi Cooperativi

Due o più agenti (sia n il loro numero) possono collaborare, al fine di vincere. In questo modello la somma del "valore" singolo di ogni giocatore è minore rispetto al valore ottenuto in squadra (ossia la somma dei singoli valori è minore del valore di una eventuale squadra formata dagli stessi giocatori). Il problema consiste nel suddividere tale surplus di valore fra ogni giocatore.

In particolare, noi ci occuperemo di **giochi cooperativi ad utilità trasferibile** (**TU, Transferable Unit**), dove il valore totale di un giocatore può essere trasferito nel valore di un altro giocatore (o di più giocatori). La versione duale dei giochi cooperativi TU è la tipologia di giochi **NTU** (**Non Transferable Unit**), dove sono imposti specifici vincoli.

Vediamo il modello dei giochi TU:

$G = (N, v)$

$N = \{1, 2, 3, \dots\dots\dots, n\}$ $(insieme\ degli\ agenti)$

$v = 2^N \rightarrow \mathbb{R}$ $(dato\ un\ sottoinsieme\ non\ vuoto\ di\ agenti\ restituisce\ un\ valore\ reale$

L'insieme N viene detto **Grand Coalition**. Supporremo che la Grand Coalition si formi sempre inizialmente. Dobbiamo capire le condizioni per le quali essa possa sciogliersi.

Consideriamo pertanto giochi $G = (N, v)$ superadditivi, e vale:

$$\forall\ c.c' \subseteq N: c \cap c' \neq 0, v(c) + v(c') \leq v(c + c')$$

Sotto queste condizioni, la Grand Coalition si forma (in quanto conviene).

Affinché la Grand Coalition non si sciolga, è necessario che la ricchezza vada distribuita in maniera adeguata fra i membri della coalizione. Come rappresentiamo questa distribuzione di ricchezza?

$$x \in \mathbb{R}^N \rightarrow \begin{bmatrix} x_0 \rightarrow & Ricchezza\ agente\ 0 \\ \vdots & \vdots \\ x_n \rightarrow & Ricchezza\ agente\ n \end{bmatrix} \qquad v(N) = Ricchezza\ disponibile$$

Dovendo distribuire tutta la richiesta si impone: $\sum_{i=0}^{n} x_i = v(N)$. Tale proprietà è detta **efficienza.**

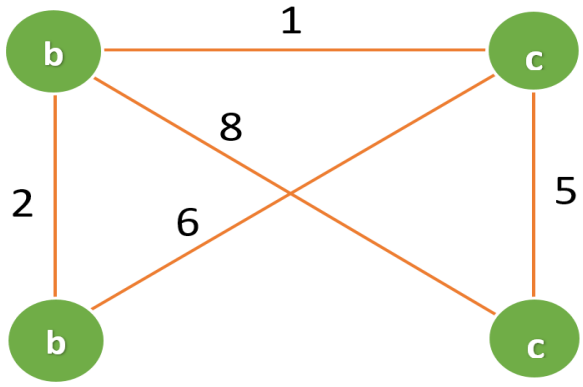
Inoltre, la ricchezza assegnata ad un giocatore deve essere almeno quanto la ricchezza che possederebbe stando da solo, per cui imponiamo: $\forall\ i,\ x_i \geq v(\{i\})$. Tale proprietà è detta **razionalità individuale.** Una distribuzione che è sia **efficiente** che **razionalmente individuale,** è detta **imputazione.**

Giochi Compatti

Un gioco è polinomialmente rappresetabile, o **compatto,** se può essere rappresentato mediante un modulo (ad es. una funzione), che, dato S, restituisce v(S) in tempo polinomiale.

Esempio (giochi a grafo)

I nodi del grafo sono i giocatori. Alcuni giocatori interagiscono fra loro, generando valore.



- $G = \{1, 2, 3, 4\}$
- $v(S) = 0\ se\ |S| = 1$
  $(un\ giocatore\ da\ solo\ vale\ 0)$
- $v(S), per\ |S| > 1$, è la somma dei pesi del sottoinsieme indotto da S
- Otteniamo $v(1,3) = 2,\ v(1,2,3) = 9$

Sappiamo che determinare la emptyness del core è un problema NP-Hard. Più formalmente:

$$C_g = \{x \in X_g \mid \forall\, S \subseteq N, x(S) \geq v(s)\}$$

Andiamo ad esplicitarne le equazioni:

$$\begin{cases} x_1 \geq v(1) \\ x_2 \geq v(2) \\ \vdots \\ x_n \geq v(n) \\ x_1 + x_2 \geq v(1,2) \\ x_1 + x_3 \geq v(1,3) \\ \vdots \\ x_1 + x_n \geq v(1,n) \\ \vdots \\ x_1 + x_2 + x_3 \geq v(1,2,3) \\ \vdots \\ \sum_{i=0}^{n} x_i \geq v(N) \end{cases}$$

Risolvere il problema di emptyness del core equivale alla risoluzione di un numero esponenziale di equazioni lineari, e ciò si può fare in tempo polinomiale, tramite vari algoritmi di PL (ad es. elissoide). Possiamo dire di poter risolvere questo problema in EXP_TIME.

Esiste un modo per renderlo polinomiale?

Vediamo un utile teorema.

Teorema di Elllis

Date k disequazioni lineari in n variabili reali $(in\ \mathbb{R}^N)$, *il sistema* è *inammssibile* $(soluzione\ vuota)$ *se e solo se esiste un sottoinsieme di* $n + 1$ *di quelle disequazioni anch'esso inammissibile*

Possiamo applicare il teorema appena visto al caso precedente:

- Il certificato per il problema di emptyness del core è dato dalle n+1 equazioni definite dal teorema di Ellis
- Basta quindi fare un guess di tali disequazioni (tramite backtracking) per dimostrarne l'inammissibilità
- Risolvere un sistema di n+1 disequazioni è polinomiale
- Il problema di verificare se una data distribuzione di ricchezza si trovi nel core segue un ragionamento simile, infatti tale problema è CO-NP Complete.

Bargaining Set

Tale gioco si basa sul concetto di **minaccia** e **contro-minaccia** (**obiezione** e **contro obiezione**).
Il gioco è definito come $G = (N, v)$, mentre x è la distribuzione di ricchezza proposta.

Diremo che (Y, S) è un'**obiezione** di i rispetto a j nel contesto x, utilizzando S se:

a) $i \in S,\ j \notin S$

b) $\forall\, k \in S,\ y_k \geq x_k$

In altre parole, ipotizzando che la distribuzione x non vada bene per il giocatore i, egli può minacciare il giocatore j di:

a) Andarsene con altri giocatori in una nuova coalizione S, della quale j non farà parte **($\boldsymbol{j \notin S}$)**

b) Di creare una nuova coalizione con un valore migliore della precedente **($\boldsymbol{y_k \geq x_k}$)**

A questo punto, può esserci una contro-minaccia. Diremo che (Z, T) è una contro-minaccia all'obiezione (Y, S) se:

a) $j \in T,\ i \notin T$

b) $\forall\, k \in S \cap T,\ z_k \geq y_k, \sum_{w \in T} z_w = v(T)$
$\forall\, k \in S - T,\ z_k \geq x_k$

In altri termini, il giocatore j controbatte alla minaccia di i:

a) j se ne va in una sua coalizione T, senza i ($i \notin T$)

b) C'è la possibilità che alcuni membri di S vadano in T: ad essi è garantita (almeno) la stessa ricchezza precedente ($z_k \geq y_k$). Per tutti gli altri, vale un discorso simile ($z_k \geq x_k$).

Riassumendo, definiamo il **Bargaining Set** come:

$$B_G = \{x \in X_g \mid \forall\ obiezione\ di\ i\ contro\ j\ esiste\ una\ contro - obiezione\}$$

Il core rappresenta un concetto di stabilità più forte, infatti è contenuto nel Bargaining Set.

Core

È un altro concetto soluzione che realizza la stabilità, ma è più forte rispetto alla stabilità fornita dallo Stable set. Il Core si definisce come un sottoinsieme di imputazioni $\mathrm{C_G} \subseteq \mathrm{X_G}$ dove C_g è detto **Core** di G:

$$\vec{x} \in C_G \Leftrightarrow \nexists S \subseteq N)\ (\vec{x}(S) < v(\vec{S}))$$

In ogni coalizione, un suo sottogruppo guadagna almeno quanto guadagnerebbe se uscisse da S, deve essere vero per ogni sottogruppo di giocatori, quindi si ragiona secondo i gruppi e non secondo i singoli giocatori. Dato un gioco, si vuole stabilire se il suo Core è vuoto o meno, oppure dato un gioco e una sua imputazione, si vuole stabilire se questa appartiene al Core del gioco o meno. Quindi dato G = (N,v) si vuole stabilire se $\mathrm{C_G} = \emptyset$, e data $\vec{x} \in \mathrm{X_G}$, stabilire se $\vec{x} \in \mathrm{C_G}$. Il core è un politopo con 2^n disequazioni in uno spazio di $\mathbb{R}^{\mathrm{N}}$, quindi in $\mathbb{R}^2$ ogni disequazione è un semipiano delimitato da una retta.

Lezione 8 29/10/2018

Introduciamo alcuni concetti chiave. Supponiamo di avere:

$$G = (N, r) \qquad x, y \in X_g$$

Diremo che:

- **x domina y via S** $(S \subseteq N)$ se $\forall\, k \in S \;\; vale\; x_k \geq y_k$, e si scrive come $\boldsymbol{X \geq_s Y}$
- **x domina y** se $\exists\, S \subseteq N : x \geq_s y$

Definiamo ora X come sottoinsieme di X_g $(X \subseteq X_g)$. Definiamo i **dominati di X** come:

$$D(X) = \{\, y \in X_g \mid \exists\, x \in X, x \geq y\}$$

Ossia D(X) rappresenta l'insieme delle imputazioni che sono dominate da almeno un membro di X.

Stable set

Introduciamo ora, per ragioni storiche, un concetto di soluzione, ossia lo **Stable Set.**

Un insieme $\boldsymbol{X \in X_g}$ di imputazioni è uno Stable-Set se $\boldsymbol{X = X_g - D(X)}$.

La condizione prescrive che:

a) $X \subseteq X_g - D(X)$
b) $X \supseteq X_g - D(X)$

Uno Stable-Set possiede due tipi di **stabilità:**

- **Stabilità interna:** siano A e B due imputazioni dello Stable-Set. A non domina B, e B non domina A. Ossia le imputazioni dello Stable-Set non si dominano a vicenda.
- **Stabilità esterna:** presa un'imputazione fuori dallo Stable-Set, esiste sempre un'imputazione contenuta nello Stable-Set che la domina.

Si può dimostrare che ogni gioco, fino a 7 giocatori, ammette uno Stable-Set.

A questo punto, viene a crearsi un problema: **dato $\boldsymbol{G = (N, v)}$, G ammette uno Stable Set?**
Tale problema rimane tutt'ora aperto, in quanto non si può stabilire se è indecidibile o no.

Finora abbiamo visto concetti di **soluzione multi-punto**, ossia soluzioni che ammettono più punti (sono presenti più soluzioni). Introduciamo ora dei concetti di soluzione con soluzione unica, ossia a **singolo punto**.

Shapley Value

Consideriamo il seguente gioco:

$$G = (N, v) \qquad\qquad S \subseteq N, \qquad i \notin S$$

Poiché parliamo di giochi super-additivi, vale che $v(S \cup \{i\}) \geq v(S)$.

Consideriamo quindi il **guadagno indotto da i su S,** ossia il guadagno introdotto da i entrando in S (il suo "contributo"):

$$d_i^S = v(S \cup \{i\}) - v(S)$$

Il guadagno indotto d_i^S non è fisso, ma dipende dalla coalizione e dall'ordine di ingresso del giocatore i nella coalizione.

Definiamo quindi lo **Shapley Value** come:

$$\varphi_i = \frac{1}{n!}\left(\sum_{S \in P(N)} d_i^S\right.$$

Definiamo inoltre il **vettore di Shapley Value**, ossia un vettore formato dagli Shapley Value di ogni giocatore i-esimo:

$$\varphi_g = < \varphi_1, \varphi_2, \ldots\ldots, \varphi_n >$$

Lo Shapley value verifica le seguenti proprietà:

- **Efficienza:** Il guadagno ottenuto viene distribuito completamente, ossia $\sum_{Xi \in N} \varphi_i = v(N)$;

- **Simmetria:** il giocatore i e il giocatore j sono equivalenti, nel senso che forniscono lo stesso contributo nella coalizione, per ogni sottoinsieme S di N che non contiene né i né j, vale $\varphi_i(v) = \varphi_j(v)$

- **Linearità:** Se due giochi cooperativi descritti da una funzione di guadagno v e w vengono combinati, allora la distribuzione di guadagno corrispondente alla ricchezza distribuita da v e quella distribuita da w è $\varphi_i(v + w) = \varphi i\,(v) + \varphi i\,(w)\ \forall i \in N$. Inoltre è vero che per ogni scalare a vale $\varphi\, i\,(a * v) = a * \varphi\, i\,(v) \quad \forall i \in N$

- **Giocatore zero**: Lo Shapley value $\varphi_i(v)$ di un giocatore nullo è zero. Un giocatore i viene definito nullo in un gioco v se vale $v(S \cup \{i\}) = v(S)$ per ogni coalizione S, cioè non da contributo.

Infine, lo Shapley Value aderisce al principio di **fairness individuale.**

Nucleolo

Questo concetto di soluzione adotta un principio di **fairness sociale, ossia "chi ci perde, ci perde il meno possibile".** E se tutti guadagnano? Allora diventa **"chi è più sfortunato (vince di meno), lo è il meno possibile".** Prendiamo una situazione tipica:

$$G = (N, v) \qquad x \in X_g, \qquad S \subseteq N$$

Definiamo l'eccesso di S su x come:

$$e(S, x) = v(s) - x(s)$$

Da qui, possiamo calcolare il **vettore degli eccessi** del giocatore x ($\theta(x)$), ossia il vettore contenente tutti gli eccessi (che sono in totale $2^n - 1$) di un determinato giocatore, calcolati su ogni coalizione:

$$\theta(x) =< e(S_1, x), e(S_2, x), \dots\dots, e(S_{2^n-1}, x)$$

Il vettore è ordinato $\forall\ j$ (in modo tale che la coalizione più sfortunata stia a sinistra).

Diremo che:

$$\theta_a < \theta_b \ se \begin{cases} \exists\ j: a_j < b_j \\ \forall\ k \leq j-1, a_k = b_k \end{cases}$$

$$[\boldsymbol{Ordine\ lessicografico}]$$

Definiamo il **nucleolo** come:

$$N_g = \{x \in X_g \mid \nexists\ y\ \in X_g, \theta(y) < \theta(x)\}$$

Valgono inoltre i seguenti teoremi:

- **Teorema:** In caso di giochi TU: $\forall\ G = (N, v), |N_g| = 1$ (ossia il nucleolo ha un solo punto)
- **Teorema:** $\forall\ G = (N, v), (C_g \neq 0 \rightarrow N_g \subseteq C_g$ (se il core non è vuoto, contiene il nucleolo; in altre parole, in condizioni di perfetta stabilità, il nucleolo è stabile)

G-Sat (Greedy Sat)

Dato un problema P di SAT, e dato $x = assegnamento\ di\ verità$:

$$V(x) = \{y\ assegnamento \mid y\ si\ ottiene\ cambiando\ di\ valore\ una\ sola\ variabile\}$$

$$Se\ P\ ha\ n\ variabili \rightarrow \forall\ x, |V(x)| = n$$

Introduciamo ora l'algoritmo:

- Sia **sodd**(x) il numero delle clausole della CNF soddisfatte da x
- Sia F una formula CNF
- Sia MAX_TRIES una costante

1. $x = \boldsymbol{random\_assign}$

   $\boldsymbol{for}\ i \coloneqq 1\ \boldsymbol{to}\ MAX\_TRIES$

   $\boldsymbol{if}\ \boldsymbol{SAT}(x, F)\ \boldsymbol{then} \quad \boldsymbol{return}\ x$

   $\boldsymbol{let}\ x' = \mathbf{max\_arg}\ (\boldsymbol{sodd}(x')), \qquad x' \in V_g$

   $\boldsymbol{if}\ \boldsymbol{sodd}(x') > \boldsymbol{sodd}(x)\ \boldsymbol{then} \qquad x = x'$

   $\boldsymbol{else} \quad \boldsymbol{go\ to}\ 1$

W-SAT (Walking-SAT)

Quest'algoritmo per la risoluzione del SAT nasce con l'introduzione di un nuovo livello di randomizzazione, ossia il **Random Walk** (da cui deriva il nome W-SAT). Vediamo il codice:

$F = Formula\ CNF, \quad P = probabilità\ di\ Random\ Walk\ (costante), \quad MAX\_TRIES = costante$

1. $x = \boldsymbol{random\_assign}$
2. $\boldsymbol{for}\ i \coloneqq 1\ \boldsymbol{to}\ MAX\_TRIES$

   $\boldsymbol{if}\ \boldsymbol{SAT}(x, F)\ \boldsymbol{then} \quad \boldsymbol{return}\ x$

   Con probabilità $(\mathbf{1} - \boldsymbol{p})$

   $\boldsymbol{do}\ \ G - SAT$

   Con probabilità p

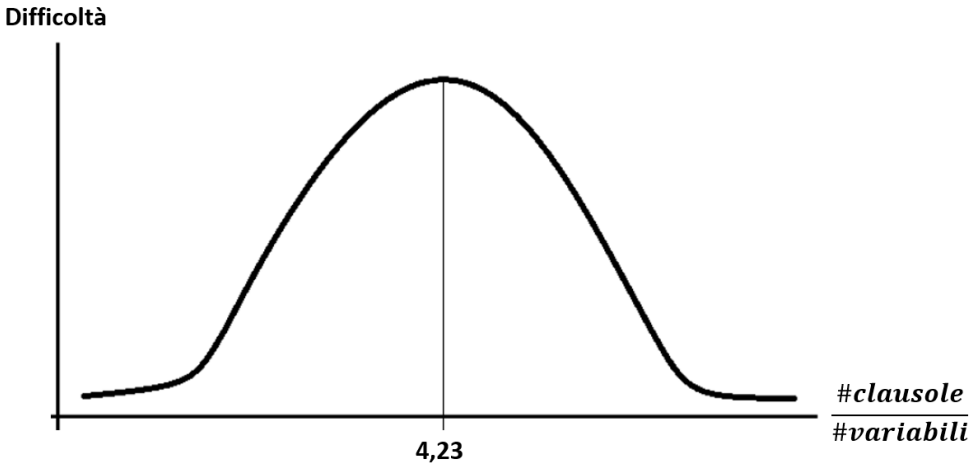
   1) Scegliamo una clausola C non soddisfatta da x
   2) Scegliamo una variabile v in C
      - **Flip**(v)

   Go to 2

Il W-SAT con un tasso di riuscita del 90% riesce a risolvere problemi con oltre 2500 variabili.

Gli autori dell'algoritmo hanno stabilito, dopo varie analisi, un parametro ($\frac{\#clausole}{\#variabili}$) che indichi quanto W-SAT è più in difficoltà (cioè quando tende a sbagliare più spesso).

Possiamo osservare sul grafico che il picco si raggiunge intorno ad un rapporto di 4,23:



Il fenomeno per il quale la difficoltà cresce fino ad un picco per poi decrescere è detto **Phase Transitioning.**

Lezione 9 05/11/2018

Logica Proposizionale

È la forma più semplice di linguaggio, non ambiguo, di rappresentazione della conoscenza.

Abbiamo a disposizione:

- **Lettere**: sono i nostri **predicati** (esprimibili tramite lettere, ad esempio a, b, c, ecc.) con valore true o false
- **Connettivi:** $\wedge, \vee, \neg, \rightarrow$ ecc. servono a concatenare le informazioni espresse dalle lettere, o a modificarle (ad esempio, $mortale(Socrate) \wedge \neg mortale(Zeus)$)
- **Formula:** composizione di una o più lettere tramite i connettivi

La **semantica** è espressa dall'insieme F delle formule proposizionali. Un sottoinsieme di F, $I \subseteq F$, è detto **interpretazione.**

Semantica a 2 valori

- $F \vDash a \quad se \quad a \in I$
- $F \vDash \neg a \quad se \quad F \nvDash a$
- $F \vDash a \wedge b \quad se \quad F \vDash a \wedge F \vDash b$
- $F \vDash a \vee b \quad se \quad F \vDash a \vee F \vDash b$
- $F \vDash a \rightarrow b \quad se \quad F \nvDash a \vee F \vDash b$

I si dice **modello** di F ($\mathbf{I \vDash F}$) $se \; \forall f \in F, f$ è $vera \; in \; I$.

Esempio:

$F = \{a, a \rightarrow b\}$

$I' = \{a\}$ ***non*** è ***un modello di F***

$I'' = \{a, b\}$ è ***un modello di F***

Esempio:

$F = \{a \rightarrow b\}$

$I' = \{a, b\}$ è ***un modello di F***

$I'' = \emptyset$ è ***un modello di F***

Possiamo quindi affermare che, data una teoria T, esisteranno, in generale, più modelli di essa.

Un'**interpretazione** è l'insieme delle **lettere** alle quali assegniamo true. **Le lettere assenti nell'interpretazione sono assunte false**. Un'interpretazione con tutto true è un **modello.**

Data una formula φ e una teoria F, **come stabilire se** $\boldsymbol{F \models^{?} \varphi}$?

Tale problema è CO-NP completo, soprattutto con la semantica. Abbiamo due tipi di semantica:

- **Brave:** $T \models \varphi \ \ se \ \ \exists\, I, I \models T : I \models \varphi$
  $(T\ implica\ \varphi\ se\ esiste\ un\ modello\ di\ T\ che\ implica\ \varphi)$

- **Cautious:** $T \models \varphi \ \ se \ \ \forall\, I, I \models T, \ \ I \models \varphi$
  $(T\ implica\ \varphi\ se\ in\ ogni\ modello\ di\ T, \varphi$ è $true)$

Il problema di stabilire se $T \models^{?} \varphi$, secondo la semantica cautious, è detto **implicazione classica** oppure **entailment problem.** Facciamo riferimento alla semantica cautious.

Vogliamo capire se, data una teoria T e una formula φ, $T \models^{?} \varphi$. Applicando la definizione vista, dovremmo analizzare tutti i possibili modelli di T, il cui numero è però esponenziale (quindi non conviene). Possiamo usare, in alternativa, un metodo semplificativo, ossia le **regole di inferenza.**

Regole di Inferenza

Nella logica matematica una regola di inferenza è uno schema formale che si applica nell'eseguire un'inferenza. In altre parole, è una regola che permette di passare da un numero finito di **proposizioni** assunte come premesse a una proposizione che funge da **conclusione**.

Nel caso una regola di inferenza sia corretta, allora stabilisce quando un enunciato formalizzato (cioè una formula di un linguaggio proposizionale o del primo ordine) è conseguenza logica di un altro, soltanto sulla base della struttura sintattica degli enunciati. La struttura di una regola è:

$$\frac{condizioni}{conseguenza}$$

In logica proposizionale, l'unica regola necessaria è la **Modus Ponens (MP):**

$$\frac{x;\ x \to y}{y}$$

Un'altra regola utile è la **Risoluzione (RIS)**:

$$\frac{x \to y;\ y \to z}{x \to z}$$

Tramite queste regole è possibile ricavare informazioni tramite ciò che è già in nostro possesso. Per cui, anziché analizzare ogni singolo modello, applichiamo più volte le regole di inferenza a nostra disposizione: $F \models'_R F' \models''_R F'' \ldots\ldots \models_{R_n} F^n$

Dato un insieme $\bar{R}$ di regole di inferenza:

- $\bar{R}$ è $\boldsymbol{sound}$ $se \; \forall F, \forall \varphi, F \vDash_R \varphi \rightarrow F \vDash \varphi$
  (ossia ciò che derivo sintatticamente, è derivabile anche semanticamente)

- $\bar{R}$ è $\boldsymbol{complete}$ $se \; \forall F, \; \forall \varphi, \; F \vDash \varphi \rightarrow F \vDash_R \varphi$

Prendendo le regole viste prima, possiamo notare che:

- MP è sound ma non complete
- RIS è sia sound che complete per formule in 2CNF. Poiché ogni formula può essere espressa in 2CNF, allora RIS è sempre sia sound che complete, nella sua forma più generale.

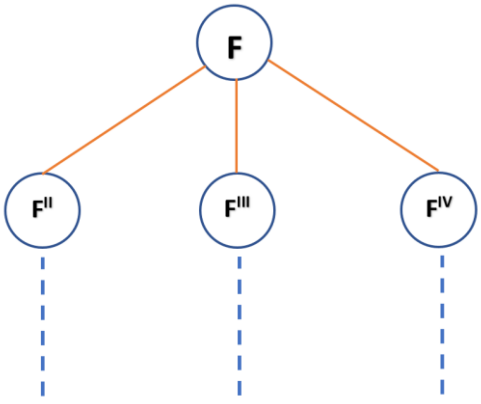
Esistono frammenti di formule che rendano MP sia sound che complete?

Formule di Horn

Sono formule che, scritte in CNF, presentano esclusivamente clausole (disgiunti) che contengono al più un letterale positivo.

Complessità del problema $F \vDash^{?} \varphi$

Parlando di complessità, il problema $F \vDash^{?} \varphi$ è CO-NP completo. Lo spazio di ricerca può essere ricondotto ad altri casi già visti:



Un formalismo di rappresentazione della conoscenza è un meccanismo che traduce un problema di ragionamento in un problema di ricerca. Grazie a questa caratteristica, possiamo "trasformare" lo spazio di ricerca in un albero, già visto in altri casi (possiamo ad esempio applicare algoritmi noti come D-F, I-D, ecc.)

Lezione 10 08/11/2018

Vediamo ora il problema inverso di $F \models^{?} \varphi$, ossia $\boldsymbol{F \not\models^{?} \varphi}$.

Prendiamo un modello M di una teoria T, che non implica φ:

$$T \models^{?} \varphi, ossia\ trovare\ M, M \models T, tale\ che\ \boldsymbol{M\ non}\ \text{è}\ \boldsymbol{un\ modello\ di\ \varphi}$$

Il certificato di questo problema è ottenibile in tempo CO-NP.

Per problemi difficili, non possiamo usare la Logica Proposizionale. In alternativa ad essa, è possibile utilizzare la **Logica del primo ordine (FOL, First Order Logic).**

Logica del primo ordine

Nella logica matematica il linguaggio del primo ordine è un linguaggio formale che serve per gestire meccanicamente enunciati e ragionamenti che coinvolgono i connettivi logici, le relazioni e i quantificatori "per ogni ..." (∀) ed "esiste..." (∃). Si dice del “primo ordine” poiché è presente un insieme di riferimento, e i quantificatori possono essere utilizzati solo sugli elementi di tale insieme e non dei suoi sottoinsiemi. In sintesi, abbiamo a disposizione:

- **Predicati**: (n-ari) ad esempio a, b, c ecc.
- **Funtori**: (n-ari) ossia funzioni lette ma non utilizzate
- **Quantificatori**: (utilizzati su variabili) ∀, ∃
- **Connettivi logici:** ∨, ∧, ¬, →, ecc.
- **Variabili:** ad esempio x, y, z, ecc.

Il **dominio** è dato da: $D = \{f \mid f \text{ è } un\ termine\ funtore\ 0-ario\}$.

Abbiamo poi i **termini:**

$$\boldsymbol{Termine = Funtore(Termine\ 1, Termine\ 2, \ldots\ldots, Termine\ n)}$$

Diamo quindi qualche definizione:

- Un **funtore 0-ario** (costante) è un **termine**
- Una **variabile** è un **termine**
- Un **funtore k-ario** applicato a k termini è un **termine**
- Un **predicato n-ario** applicato a n termini è un **atomo**
- Un **atomo** su cui applico o no un termine è un **letterale**
- Un **funtore** è, quindi, una funzione utilizzata per formare i termini (ossia un’espressione logica in grado di descrivere parte del dominio)

Qual è la differenza fra FOL e la macchina di Turing?

Nella macchina di Turing il nastro è infinito. **La vera differenza sta nei Funtori**, con i quali possiamo rappresentare il nastro. Ad esempio:

$$\boldsymbol{Nastro}(\,(),()),\boldsymbol{Nastro}(\boldsymbol{a},\boldsymbol{Nastro}(),\boldsymbol{Nastro}(\boldsymbol{a},\boldsymbol{Nastro}(\boldsymbol{b})\,\ldots\ldots.)$$

Abbiamo però alcuni problemi. Ad esempio, costruire una macchina (anche solo software) che usi come linguaggio macchina la FOL è molto complicato.

La complessità deriva dal fatto che la FOL differisce per molti aspetti dal nostro modo di ragionare:

- Contenuti numerici (probabilistici ecc.) sono complicati da esprimere in FOL
- Il nostro ragionare è **defeatable**, ossia può essere "sconfitto": ciò che in un momento è vero, può essere falso acquisendo conoscenza
- Nel nostro modo di ragionare non ci sono solo cose vere o solo cose false, ma anche informazioni che sono vere o false a seconda del contesto di utilizzo (i **beliefs**). Questo tipo di informazioni non è rappresentabile in FOL
- Nel ragionamento comune sono presenti i cosiddetti **default**, ossia fatti che vengono assunti veri, anche se non lo sono sempre; in sintesi, sono informazioni ritenute vere (o false) sempre che non si verifichi il contrario ("Tutti gli studenti hanno i capelli scuri, di solito").

Tutto ciò ci porta alla conclusione che FOL non basta, ma occorre introdurre dei meccanismi per esprimere i **beliefs.** Serve una logica che sappia rappresentare i default, cioè devono poter esistere informazioni assunte vere anche se non lo sono sempre (e se non lo sono, non provocano grandi cambiamenti). Proviamo ad esprimere più formalmente i concetti visti.

Consideriamo la seguente coppia:

$$\Delta = (w, D)$$

Dove:

$w = \text{conoscenza certa, informazioni sempre corrette}$

$D = conoscenza\ di\ Default, inforazioni\ vere\ o\ no, nonostante\ il\ contrario$

D sarà espresso come insieme di formule proposizionali: $D = \{f_1, f_2 \dots \dots, f_n\}$

Nello specifico:

$$f_i = \frac{\alpha_i : \psi_1^1, \dots \dots, \psi_i^k}{x_i}$$

$\alpha_i = premessa\ (pre - condizione)$

$x_i = conseguenza\ (dedotta)$

$\psi_i = condizione\ (giustificazione)$

Esempio:

$$f = \frac{student : \neg\ from\_Sweden \wedge\ blackhaired}{blackhaired}$$

Poiché è macchinoso usare la forma proposizionale per esprimere i default, faremo uso delle variabili.

Possiamo imporre la **closed world assumption** (l'ipotesi secondo cui ogni affermazione il cui valore di verità non è noto è considerata falsa) con **default logic.** Supponiamo che f sia un frammento di conoscenza:

$$f = \frac{true : \neg\varphi}{\neg\varphi}$$

Diamo alcune definizioni:

- Un default è **normale** se la sua giustificazione coincide con la sua conseguenza
- Una **teoria** è **normale** se formata solo da default normali
- Un **default** è **semi-normale** se la sua conseguenza è un sottoinsieme (anche proprio) della sua premessa

Lezione 11 12/11/2018

Prendiamo un esempio:

$$\frac{Sedia^T : \neg\, anormale\_sedia}{4\_piedi}$$

In logica proposizionale potremmo dire:

$$sedia^T \wedge \neg\, anormale\_sedia \rightarrow 4\_piedi$$

In realtà non sono equivalenti:

$\frac{Sedia^T : \neg\, anormale_{sedia}}{4_{piedi}}$ $\dashrightarrow$ **1 modello**: M = {sedia, 4_piedi}

$sedia^T \neg\, anormale_{sedia} \rightarrow 4_{piedi}$ $\dashrightarrow$ **2 modelli:**

M1 = {sedia, 4_piedi},

M2 = {sedia, anormale_sedia}

Come si può vedere, non abbiamo lo stesso numero di modelli in entrambi i casi.

Le due espressioni non sono, quindi, equivalenti.

Semantica

Definiamo ora la **semantica.** Diamo qualche definizione preliminare:

- δ è **applicabile** rispetto agli insiemi, **logicamente chiusi**, E ed F se:

  1) $Pre(\delta) \in E$, ossia la premessa di δ appartiene a E
  2) $\forall\, \psi \in just(\delta), \neg\, \psi \notin F$

- Sia $\tilde{E}$ un'estensione di E, allora per ogni default applicabile ad E e $\tilde{E}$, deve valere:

$$cons(\delta) \in E$$

Diamo inoltre un'altra definizione di **estensione.** Immaginiamo di avere:

$$\Delta = (w, D)$$

Diremo che E è un'estensione di Δ (E ⊨ Δ) se $E = \cup_i (E_i)$, dove:

- $E_0 = Th(w)$
- $E_{i+1} = Th(E_i \cup \{ cons(\delta) \mid \delta \in D, \delta \text{ è } applicabile\ a\ E_i, ed\ E \})$

Tale definizione, per come è posta, è **semi-induttiva**, **non costruttiva.** Questo perché per avere E_{i+1} serve sia E_i che il risultato, cioè E.

Non essendo costruttiva, è possibile verificare se una data espressione è un'estensione, ma non è possibile creare un'estensione da zero. Una soluzione può essere la chiusura transitiva, ossia applichiamo tutti i default possibili in tutti i modi possibili. Definiamo quindi il **processo:**

$$D = (w, D)$$

$M =< \delta_1, \delta_2, \dots\dots, \delta_k >$ con

- $\forall\ \delta_i \in M, \delta_i \in D$
- $\forall\ \delta_i, \delta_j \in M,\ \delta_i \neq \delta_j$

$\Pi[k] =< \delta_1, \delta_2, \dots\dots, \delta_k\}, \quad k \leq |\Pi|$ Π è una **sequenza** di informazioni

Definiamo 2 insiemi:

$$In(\Pi) = Th(\ w \cup \{cons(\delta) \mid \delta \in\ \Pi\})$$

$$Out(\Pi) = \{\neg\, \psi \mid \exists\ \delta \in \Pi, \psi \in just(\delta)\}$$

Non tutte le sequenze sono valide, definiamo quelle corrette.

Una sequenza è un **processo** (è una sequenza corretta) se:

$$\forall\ k, \delta_k \text{ è } applicabile\ a\ In(\Pi[k-1])\ e\ In(\Pi[k])$$

Ossia:

$$Pre(\delta_k) \in In(\Pi[k-1])$$

$$\forall\ \psi_i \in just(\delta_k), \neg\, \psi\ \notin In(\Pi[k-1])$$

Diremo che:

- Un processo è **successful** se: $In(\Pi) \cap Out(\Pi) = \emptyset$
- Un processo è **closed** se: $\forall\ \delta \in D, \delta \text{ è } applicabile\ a\ In(\Pi)\ e\ In(\Pi), ossia \rightarrow \delta \in \Pi$

Vediamo qualche esempio:

Esempio:

$$\Delta = (\{a\}, \{\underbrace{\frac{a : \neg b}{d}}_{\delta_1}; \underbrace{\frac{true : c}{b}}_{\delta_2}\})$$

$\pi' =< \delta_1 > \begin{cases} In(\pi') = Th(\{a, d\}) \\ Out(\pi') = \{b\} \end{cases} \rightarrow \boldsymbol{\pi'}$ è ***successfull ma non closed***

$\pi'' =< \delta_1,\ \delta_2 > \begin{cases} In(\pi'') = Th(\{a, d, b\}) \\ Out(\pi'') = \{b, \neg c\} \end{cases} \rightarrow \boldsymbol{\pi''}$ è ***closed ma non successful***

$\pi''' =< \delta_2 > \begin{cases} In(\pi''') = Th(\{a, b\}) \\ Out(\pi''') = \{\neg c\} \end{cases} \rightarrow \boldsymbol{\pi'''}$ è ***sia closed che successful***

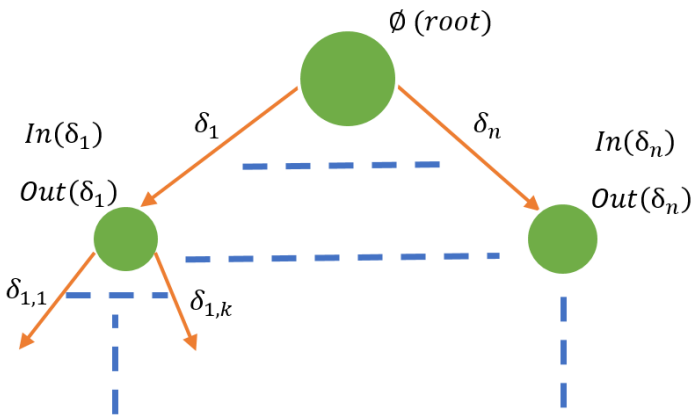
Notiamo come π''' rispetti entrambe le condizioni (successful e closed). Ciò corrisponde, inoltre, alla nostra definizione di **estensione**. Esiste una regola generale, espressa dal seguente teorema.

Teorema:

Sia $\Delta = (w, D)$. $E \vDash \Delta$ *se* $\exists\ \pi$ *processo di* Δ *closed e successful, tale che* $E = in(\pi)$

Avendo Δ = (w, D), posso generare un albero (ossia vengono generati tutti i possibili default):

- La radice (root) sarà: $In(root) = Th(w)\ \ e\ \ Out(root) = \emptyset$
- In alternativa a root: $In(\emptyset) = Th(w)$ *e* $\emptyset$



Le foglie che sono sia closed che succesfull saranno le **estensioni.**
I rami corrispondono ai default applicati. Ad ogni nodo corrisponderà quindi un diverso default, con i relativi In e Out. Diamo qualche definizione finale:

- Una **teoria** senza estensioni è detta **incoerente**

- Una **teoria** che consente di generare qualsiasi cosa è detta **inconsistente**

- Una **teoria di default** è inconsistente se e solo se è la sua componente certa (w) è inconsistente

Lezione 12

19/11/2018