



APPUNTI INTELLIGENZA ARTIFICIALE

DANILO FORTUGNO

Index

*Da Visualizzatore PDF Per arrivare alla pagina effettuare un +2



01	Agenti e Ambienti	pag. 1
02	Search	pag. 2
03	Informed Search	pag. 12
04	Search in Complex Environments	pag. 17
05	Adversial Search	pag. 20
06	Uncertainty and Utilities	pag. 25
07	Constraints Satisfaction Problem	pag. 28
08	Planning	pag. 45
09	Answer Set Programming	pag. 52
10	Game Theory	pag. 64

Agenti e Ambienti



Un **agente** è qualsiasi cosa possa essere vista come un sistema che percepisce il suo ambiente attraverso dei sensori e agisce su di esso mediante attuatori.

Come gli uomini possiedono i 5 sensi, l'obiettivo degli agenti è simulare questo aspetto tramite elementi che lo permettono.

La differenza che si presenta tra i vari agenti è dettata da dove questi vengono o posti all'interno del mondo, poiché sulla base del loro dominio possono eseguire o meno delle azioni al fine di raggiungere un goal.

Dato che per un robot è impossibile gestire una quantità di dati pari a quella dell'uomo, l'**ambiente** in cui deve essere inserito ha come necessità l'essere *minimale*.

Esistono diverse rappresentazioni di un *ambiente*:

1. **ATOMICA:** ogni stato del mondo è indivisibile e non ha struttura interna.
2. **FATTORIZZATA:** si suddivide ogni stato in un insieme fissato di *variabili* e *attributi* ognuno dei quali può avere un *valore*.
3. **STRUTTURATA:** Si può descrivere in modo esplicito sia gli oggetti che le relazioni che intercorrono tra essi.

Queste rappresentazioni sono poste in un *crescendo* di *espressività*, maggiore è il potere espressivo, maggiore è il livello di dettaglio che quell'ambiente possiede.

Search

Quando l'agente deve effettuare una decisione ma non riesce a capire nell'immediato quale è la mossa corretta da effettuare, può decidere di *analizzare* i possibili stati successivi che si potrebbero creare considerando una determinata mossa e poi effettuare quella che sembra effettivamente la più vantaggiosa.

Un agente di questo tipo prende il nome di **agente risolutore di problemi** in questo caso sfruttano una rappresentazione del mondo *atomica*.

Gli agenti che utilizzano una rappresentazione *fattorizzata* prendono il nome di **agenti pianificatori**.

La ricerca a sua volta si suddivide in due tipi che: **Informata e Non-Informata**.

Il processo di risoluzione di un agente è suddiviso in 4 stati:

1. **Formulazione dell'obiettivo**
2. **Formulazione del Problema**
3. **Ricerca**
4. **Esecuzione.**

L'agente può trovarsi in ambienti in cui non si hanno eventi esterni che possono interferire con l'esecuzione (**Struttura ad Anello chiuso**) e dunque una volta identificato il piano può semplicemente portarlo a conclusione; oppure ambienti in cui, a causa di eventi esterni, il piano deve essere continuamente riadattato. (**Struttura ad Anello aperto**).

Formulazione di un Problema

Al fine di comprendere il ragionamento che esegue l'agente per trovare il piano è necessario definire formalmente un problema.



Un **problema di ricerca** può essere formalmente definito come una situazione in cui un agente deve trovare una sequenza di azioni che lo porti da uno stato iniziale a uno stato obiettivo all'interno di uno **spazio degli stati**.

In particolare è costituito dai seguenti Elementi:

- **Spazio degli Stati:** è un possibile insieme di stati in cui si può trovare l'agente. In ogni stato sono conservati i dettagli necessari alla pianificazione.
- **Stato Iniziale:** è il punto di partenza dell'agente.
- **Stati Obiettivo:** Sono uno o un insieme di stati che l'agente deve raggiungere.
- **Azioni:** Permettono all'agente di passare da uno stato s ad uno stato s' . L'insieme delle azioni è definito nel dominio iniziale.
- **Modello di Transizione:** Descrive ciò che fa ogni azione applicata ad un determinato stato.
- **Funzione di costo:** è definita come una tripla (s, a, s') e identifica quanto costa passare dallo stato s allo stato s' applicando l'azione a .



Una sequenza di *azioni* forma un **cammino**.

Una **soluzione** è un *cammino* che porta l'agente dallo **stato iniziale** ad uno **stato obiettivo**.

Una **soluzione è ottima** se tra tutte le possibili soluzioni è quella di *costo minore*.

Una formulazione di un problema è un **modello** del mondo reale tramite un processo di astrazione matematico.

Dunque, l'obiettivo è andare a generare un modello il maggiormente astratto possibile che però mantenga tutte le informazioni principali per permettere all'agente di giungere al *goal*.

Rappresentazione di un Problema

Al fine di raggiungere l'obiettivo è necessario comprendere come i problemi possono essere rappresentati; vengono utilizzate principalmente due strutture:

Search Trees

Ogni *nodo* dell'albero rappresenta uno dei possibili stati, mentre gli *archi* identificano le possibili azioni a partire dallo stato s.

In pratica l'arco e i nodi collegati rappresentano una ***funzione di successione***.

La caratteristica peculiare degli alberi è la possibilità di avere degli *stati ridondanti*, poichè non è possibile effettuare un rollback una volta considerata un'azione.

L'assenza di memoria del passato garantisce una struttura semplice e poco dispendiosa per la memoria.

La ricerca avviene tramite espansione dei nodi sulla *frontiera*.



La **Frontiera** è l'insieme dei nodi nello spazio degli stati che sono stati **generati** ma non ancora **espansi**.

State Space Graph

Il grafo dello stato dello spazio rappresenta un modello matematico per i search problem: ogni nodo rappresenta una possibile configurazione ed ogni arco identifica il risultato dell'azione presa in considerazione, l'obiettivo è identificare uno o più stati di gol che interessano.

La peculiarità è data dall'**impossibilità** di esplorare due volte lo stesso stato. Per permettere ciò si utilizzano strutture dati apposite che memorizzano gli stati precedentemente analizzati.



Lo State Space Graph ha un insieme di stati che è FINITO; invece, il Search tree non è necessariamente finito, in particolare se il grafo presenta un ciclo l'albero è di conseguenza infinito.

General Tree Search

L'algoritmo più semplice da applicare al fine di risolvere un problema è la *general tree Search*.

La ricerca, come suggerisce il nome, è generale: si espandono i nodi sulla frontiera fino a quando non si raggiunge una soluzione oppure non esistono più *candidati da espandere*.

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

La scelta dei candidati è dettato dal modello di ricerca che si utilizza, a seguire saranno analizzati i più comuni.

▼ Termini Tecnici

1. Complete (Completamento):

o Un algoritmo di ricerca è completo se è garantito trovare una soluzione se ne esiste una. In altre parole, se c'è una soluzione nel problema, l'algoritmo completa la sua esecuzione e la trova.

2. Optimal (Ottimale):

o Un algoritmo di ricerca è ottimale se è garantito trovare il percorso di costo minimo o la soluzione di costo minimo. L'obiettivo è trovare la soluzione più efficiente o il percorso più breve in base a una metrica specifica.

3. **Time Complexity (Complessità Temporale):**
o La complessità temporale di un algoritmo di ricerca rappresenta il tempo richiesto per completare l'esecuzione in termini di operazioni di base. Può essere espressa in notazione "O" (Grande O) e dipende dalla dimensione del problema.
4. **Space Complexity (Complessità Spaziale):**
o La complessità spaziale di un algoritmo di ricerca rappresenta lo spazio di memoria richiesto durante l'esecuzione. Anche questa può essere espressa in notazione "O" e dipende dalla dimensione del problema.
5. **Cartoon of Search Tree (Rappresentazione Grafica dell'Albero di Ricerca):**
o Un albero di ricerca è una rappresentazione grafica che mostra come l'algoritmo esplora lo spazio di ricerca. Nella rappresentazione, i nodi corrispondono agli stati e gli archi alle azioni. I nodi di soluzione sono evidenziati.
6. **Branching Factor (Fattore di Branca):**
o "b" rappresenta il fattore di branca, ovvero il numero massimo di successori generati da ogni stato. È il numero massimo di nodi figlio che ogni nodo può avere nell'albero di ricerca.
7. **Maximum Depth (Massima Profondità):**
o "m" rappresenta la massima profondità dell'albero, ovvero la distanza massima dalla radice al nodo più profondo.
8. **Number of Nodes in Entire Tree (Numero di Nodi nell'Albero Completo):**
o La formula $1 + b + b^2 + \dots + b^m$ rappresenta il numero totale di nodi nell'albero di ricerca, assumendo che il numero massimo di profondità sia m e il fattore di branca sia b. Questa somma può essere approssimata a $O(b^m)$ nell'analisi della complessità.

Depth First Search

Tramite questo processo si espande sempre **il nodo più in profondità nella frontiera**.

La ricerca in *profondità* non è **ottima** rispetto al costo in quanto restituisce sempre la prima soluzione che trova, la quale non è detto che sia la migliore.

	Completo	Efficiente	Notes
Spazi Stati Finiti	X	X	Dato che lo spazio è finito nel peggiore dei casi si analizzerà l'intero albero e se esiste una soluzione sicuramente la troverà
Spazi Stati Infiniti			Nel caso di Spazio infinito si ha la presenza di un ciclo, questo comporta che, previo controllo, l'algoritmo può ritrovarsi in un loop infinito rendendo l'algoritmo incompleto

Dati questi problemi, l'unico vantaggio è dato dal costo della ricerca che ha un *costo temporale spaziale* pari a $O(bm)$ ed una *complessità temporale* pari a $O(b^m)$

Esiste una variante della ricerca in profondità che fa utilizzo del *backtracking*, tramite questa scelta non si espandono tutti i possibili successori di un nodo, ma solo un nodo per volta che verrà esplorato; nel caso di fallimento si fa un roll-back e si esplora il nodo successivo.

In questo modo i *costi spaziali* scendono a $O(m)$

Breadth First Search

La visita in ampiezza è gestita con una *coda* con politica **FIFO** (first-in first-out), in questo modo i primi nodi inseriti sono i primi ad essere espansi.

Data l'espansione a *livelli uniformi* la ricerca in ampiezza ci fornirà sempre la **soluzione ottima**, in quanto se ha espanso tutti i nodi al livello *d-1* e non ha trovato alcuna soluzione, e la trova al livello *d* allora sicuramente è quella di costo minore.

Il problema di questa ricerca è dettata dal costo: supponendo di avere un fattore di *branching* pari a *b*, per ogni nodo verranno generati (ed inseriti nella coda) b nodi.

Lavorando per livelli notiamo come al primo livello ci sarà solo la radice (1); al secondo livello avremo tutti i suoi successori (b); al livello successivo tutti i successori dei precedenti b nodi generati; formalmente:

$$\rightarrow 1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$$

dove d è la profondità massima raggiunta.

Quindi si presenta un *costo esponenziale* che è preoccupante dato che in questi problemi il numero di stati generati è elevatissimo.



In generale, i problemi di ricerca con complessità esponenziale non possono essere risolti mediante ricerche ***non informate*** tranne per istanze piccole.

▼ Confronto tra BFS e DFS

La **ricerca in profondità** ha il vantaggio di richiedere una quantità molto limitata di memoria, in quanto mantiene solo il percorso attuale. Tuttavia, presenta il rischio di scendere all'infinito, specialmente se lo spazio degli stati è molto ampio o contiene cicli. È simile all'algoritmo di **backtracking**: oltre al percorso, conserva alcune informazioni aggiuntive (come la gestione dei rami da esplorare), ma il costo in termini di memoria non cresce esponenzialmente, bensì **linearmente** rispetto alla profondità massima raggiunta.

D'altra parte, la **ricerca in ampiezza** non corre il rischio di andare all'infinito, poiché esplora sistematicamente gli stati a partire dal più vicino. Se esiste una soluzione più vicina allo stato iniziale, questa sarà trovata per prima, garantendo l'ottimalità in termini di lunghezza del cammino. Tuttavia, man mano che i livelli aumentano, la **frontiera** della ricerca cresce **esponenzialmente**, diventando un problema significativo per la memoria, soprattutto nei casi di problemi reali con spazi di stati molto grandi.

Uniform Cost Search

Quando le azioni presentano costi differenti è possibile applicare una ricerca basandosi sul *costo minore*.

L'algoritmo può essere visto come una ricerca ***best-first*** la cui *funzione di valutazione* è dettata dal *costo del cammino minore*.

Identificando con C^* il costo della soluzione ottima e con ε un *lower bound* sul costo delle azioni, avremo come caso peggiore $O(b^{\frac{C^*}{\varepsilon}})$ che può essere anche molto maggiore b^d .

Questo avviene in quanto con la UCS è possibile che si espandano alberi molto grandi prima di andare a valutare azioni che hanno un costo maggiore ma portano più facilmente al *goal*.

Il vantaggio che si ha è che sicuramente, se c'è una soluzione la trova, dunque è *completo*.

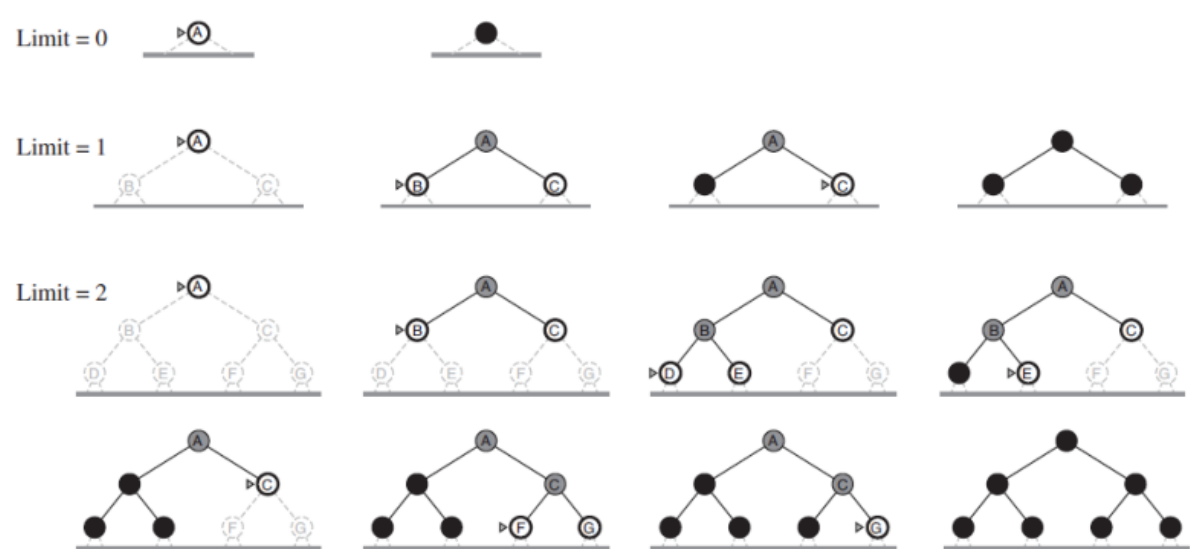
Iterative Deepening

È una strategia utilizzata in combinazione con la *ricerca in profondità*.

Il problema principale della *ricerca in profondità* è data dall'impossibilità di poter garantire l'ottimalità, per risolvere questo aspetto si fa un'analisi nella quale si incrementa man mano il livello di profondità a cui l'algoritmo deve arrivare fino a quando non si trova il goal.

Supponendo che l sia la profondità massima raggiunta, come nella ricerca in ampiezza, se fino alla profondità $l-1$ non è stata trovata una soluzione e alla profondità l invece sì, allora è la soluzione ottima.

Il *costo temporale* è $O(b^l)$, il *costo spaziale* è $O(bl)$.



Nel caso peggiore la **iterative deepening** ha asintoticamente lo stesso costo della **ricerca in ampiezza**.
In quel caso il valore di l è pari alla profondità massima dell'albero.

Bi-Directional Search

Se il nodo obiettivo è noto, è possibile eseguire una ricerca partendo non solo dallo stato iniziale, ma anche dal goal stesso.

Questo approccio sfrutta il fatto che, in un **albero**, ogni nodo ha un solo genitore, e non ci sono ambiguità nelle scelte dei cammini.

Anche se lo spazio degli stati non è propriamente un albero, può comunque avere una struttura simile. Per migliorare l'efficienza della ricerca, si può utilizzare una strategia bidirezionale: si avviano contemporaneamente due ricerche, una dallo stato iniziale e una dal goal, e si portano avanti entrambe finché è conveniente.

L'obiettivo è che le due ricerche si incontrino, ottenendo una soluzione completa combinando le soluzioni parziali delle due espansioni.

In questo approccio, si utilizzano due **code di priorità** (una per ciascuna direzione). La ricerca procede come segue:

1. Si prende un nodo dalla frontiera di una delle due direzioni (la scelta può essere basata su un criterio di priorità o euristica).
2. Si espandono i figli di quel nodo.
3. Se un figlio non è ancora stato raggiunto dalla ricerca, o se la nuova strada ha un costo inferiore rispetto a un cammino precedente, il figlio viene aggiunto alla frontiera e il costo associato viene aggiornato.
4. A questo punto, si controlla se il nodo espanso è già stato raggiunto dalla ricerca che procede nella direzione opposta (dall'altra frontiera).
 - Se il nodo è stato già esplorato, significa che le due ricerche si sono incontrate, e si può costruire una **soluzione completa** combinando i percorsi parziali ottenuti dalle due espansioni.

In questo modo, si riduce lo spazio di ricerca esplorato rispetto a una ricerca unidirezionale, potendo ottenere una soluzione più velocemente, specialmente quando lo spazio degli stati è grande.

Tabella Riassuntiva

	Breadth-First	Depth-First	Uniform Cost	Iterative Deepening	Bi-Directional
Completo	X		X	X	X
Ottimale	X		X	X	X

Search

Quando l'agente deve effettuare una decisione ma non riesce a capire nell'immediato quale è la mossa corretta da effettuare, può decidere di *analizzare* i possibili stati successivi che si potrebbero creare considerando una determinata mossa e poi effettuare quella che sembra effettivamente la più vantaggiosa.

Un agente di questo tipo prende il nome di **agente risolutore di problemi** in questo caso sfruttano una rappresentazione del mondo *atomica*.

Gli agenti che utilizzano una rappresentazione *fattorizzata* prendono il nome di **agenti pianificatori**.

La ricerca a sua volta si suddivide in due tipi che: **Informata e Non-Informata**.

Il processo di risoluzione di un agente è suddiviso in 4 stati:

1. **Formulazione dell'obiettivo**
2. **Formulazione del Problema**
3. **Ricerca**
4. **Esecuzione.**

L'agente può trovarsi in ambienti in cui non si hanno eventi esterni che possono interferire con l'esecuzione (**Struttura ad Anello chiuso**) e dunque una volta identificato il piano può semplicemente portarlo a conclusione; oppure ambienti in cui, a causa di eventi esterni, il piano deve essere continuamente riadattato. (**Struttura ad Anello aperto**).

Formulazione di un Problema

Al fine di comprendere il ragionamento che esegue l'agente per trovare il piano è necessario definire formalmente un problema.



Un **problema di ricerca** può essere formalmente definito come una situazione in cui un agente deve trovare una sequenza di azioni che lo porti da uno stato iniziale a uno stato obiettivo all'interno di uno **spazio degli stati**.

In particolare è costituito dai seguenti Elementi:

- **Spazio degli Stati:** è un possibile insieme di stati in cui si può trovare l'agente. In ogni stato sono conservati i dettagli necessari alla pianificazione.
- **Stato Iniziale:** è il punto di partenza dell'agente.
- **Stati Obiettivo:** Sono uno o un insieme di stati che l'agente deve raggiungere.
- **Azioni:** Permettono all'agente di passare da uno stato s ad uno stato s' . L'insieme delle azioni è definito nel dominio iniziale.
- **Modello di Transizione:** Descrive ciò che fa ogni azione applicata ad un determinato stato.
- **Funzione di costo:** è definita come una tripla (s, a, s') e identifica quanto costa passare dallo stato s allo stato s' applicando l'azione a .



Una sequenza di *azioni* forma un **cammino**.

Una **soluzione** è un *cammino* che porta l'agente dallo **stato iniziale** ad uno **stato obiettivo**.

Una **soluzione è ottima** se tra tutte le possibili soluzioni è quella di *costo minore*.

Una formulazione di un problema è un **modello** del mondo reale tramite un processo di astrazione matematico.

Dunque, l'obiettivo è andare a generare un modello il maggiormente astratto possibile che però mantenga tutte le informazioni principali per permettere all'agente di giungere al *goal*.

Rappresentazione di un Problema

Al fine di raggiungere l'obiettivo è necessario comprendere come i problemi possono essere rappresentati; vengono utilizzate principalmente due strutture:

Search Trees

Ogni *nodo* dell'albero rappresenta uno dei possibili stati, mentre gli *archi* identificano le possibili azioni a partire dallo stato *s*.

In pratica l'arco e i nodi collegati rappresentano una ***funzione di successione***.

La caratteristica peculiare degli alberi è la possibilità di avere degli *stati ridondanti*, poichè non è possibile effettuare un rollback una volta considerata un'azione.

L'assenza di memoria del passato garantisce una struttura semplice e poco dispendiosa per la memoria.

La ricerca avviene tramite espansione dei nodi sulla *frontiera*.



La **Frontiera** è l'insieme dei nodi nello spazio degli stati che sono stati **generati** ma non ancora **espansi**.

State Space Graph

Il grafo dello stato dello spazio rappresenta un modello matematico per i search problem: ogni nodo rappresenta una possibile configurazione ed ogni arco identifica il risultato dell'azione presa in considerazione, l'obiettivo è identificare uno o più stati di gol che interessano.

La peculiarità è data dall'**impossibilità** di esplorare due volte lo stesso stato. Per permettere ciò si utilizzano strutture dati apposite che memorizzano gli stati precedentemente analizzati.



Lo State Space Graph ha un insieme di stati che è FINITO; invece, il Search tree non è necessariamente finito, in particolare se il grafo presenta un ciclo l'albero è di conseguenza infinito.

General Tree Search

L'algoritmo più semplice da applicare al fine di risolvere un problema è la *general tree Search*.

La ricerca, come suggerisce il nome, è generale: si espandono i nodi sulla frontiera fino a quando non si raggiunge una soluzione oppure non esistono più *candidati da espandere*.

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

La scelta dei candidati è dettato dal modello di ricerca che si utilizza, a seguire saranno analizzati i più comuni.

▼ Termini Tecnici

1. Complete (Completamento):

o Un algoritmo di ricerca è completo se è garantito trovare una soluzione se ne esiste una. In altre parole, se c'è una soluzione nel problema, l'algoritmo completa la sua esecuzione e la trova.

2. Optimal (Ottimale):

o Un algoritmo di ricerca è ottimale se è garantito trovare il percorso di costo minimo o la soluzione di costo minimo. L'obiettivo è trovare la soluzione più efficiente o il percorso più breve in base a una metrica specifica.

3. **Time Complexity (Complessità Temporale):**
o La complessità temporale di un algoritmo di ricerca rappresenta il tempo richiesto per completare l'esecuzione in termini di operazioni di base. Può essere espressa in notazione "O" (Grande O) e dipende dalla dimensione del problema.
4. **Space Complexity (Complessità Spaziale):**
o La complessità spaziale di un algoritmo di ricerca rappresenta lo spazio di memoria richiesto durante l'esecuzione. Anche questa può essere espressa in notazione "O" e dipende dalla dimensione del problema.
5. **Cartoon of Search Tree (Rappresentazione Grafica dell'Albero di Ricerca):**
o Un albero di ricerca è una rappresentazione grafica che mostra come l'algoritmo esplora lo spazio di ricerca. Nella rappresentazione, i nodi corrispondono agli stati e gli archi alle azioni. I nodi di soluzione sono evidenziati.
6. **Branching Factor (Fattore di Branca):**
o "b" rappresenta il fattore di branca, ovvero il numero massimo di successori generati da ogni stato. È il numero massimo di nodi figlio che ogni nodo può avere nell'albero di ricerca.
7. **Maximum Depth (Massima Profondità):**
o "m" rappresenta la massima profondità dell'albero, ovvero la distanza massima dalla radice al nodo più profondo.
8. **Number of Nodes in Entire Tree (Numero di Nodi nell'Albero Completo):**
o La formula $1 + b + b^2 + \dots + b^m$ rappresenta il numero totale di nodi nell'albero di ricerca, assumendo che il numero massimo di profondità sia m e il fattore di branca sia b. Questa somma può essere approssimata a $O(b^m)$ nell'analisi della complessità.

Depth First Search

Tramite questo processo si espande sempre **il nodo più in profondità nella frontiera**.

La ricerca in *profondità* non è **ottima** rispetto al costo in quanto restituisce sempre la prima soluzione che trova, la quale non è detto che sia la migliore.

	Completo	Efficiente	Notes
Spazi Stati Finiti	X	X	Dato che lo spazio è finito nel peggiore dei casi si analizzerà l'intero albero e se esiste una soluzione sicuramente la troverà
Spazi Stati Infiniti			Nel caso di Spazio infinito si ha la presenza di un ciclo, questo comporta che, previo controllo, l'algoritmo può ritrovarsi in un loop infinito rendendo l'algoritmo incompleto

Dati questi problemi, l'unico vantaggio è dato dal costo della ricerca che ha un *costo temporale spaziale* pari a $O(bm)$ ed una *complessità temporale* pari a $O(b^m)$

Esiste una variante della ricerca in profondità che fa utilizzo del *backtracking*, tramite questa scelta non si espandono tutti i possibili successori di un nodo, ma solo un nodo per volta che verrà esplorato; nel caso di fallimento si fa un roll-back e si esplora il nodo successivo.

In questo modo i *costi spaziali* scendono a $O(m)$

Breadth First Search

La visita in ampiezza è gestita con una *coda* con politica **FIFO** (first-in first-out), in questo modo i primi nodi inseriti sono i primi ad essere espansi.

Data l'espansione a *livelli uniformi* la ricerca in ampiezza ci fornirà sempre la **soluzione ottima**, in quanto se ha espanso tutti i nodi al livello *d-1* e non ha trovato alcuna soluzione, e la trova al livello *d* allora sicuramente è quella di costo minore.

Il problema di questa ricerca è dettata dal costo: supponendo di avere un fattore di *branching* pari a *b*, per ogni nodo verranno generati (ed inseriti nella coda) b nodi.

Lavorando per livelli notiamo come al primo livello ci sarà solo la radice (1); al secondo livello avremo tutti i suoi successori (b); al livello successivo tutti i successori dei precedenti b nodi generati; formalmente:

$$\rightarrow 1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$$

dove d è la profondità massima raggiunta.

Quindi si presenta un *costo esponenziale* che è preoccupante dato che in questi problemi il numero di stati generati è elevatissimo.



In generale, i problemi di ricerca con complessità esponenziale non possono essere risolti mediante ricerche ***non informate*** tranne per istanze piccole.

▼ Confronto tra BFS e DFS

La **ricerca in profondità** ha il vantaggio di richiedere una quantità molto limitata di memoria, in quanto mantiene solo il percorso attuale. Tuttavia, presenta il rischio di scendere all'infinito, specialmente se lo spazio degli stati è molto ampio o contiene cicli. È simile all'algoritmo di **backtracking**: oltre al percorso, conserva alcune informazioni aggiuntive (come la gestione dei rami da esplorare), ma il costo in termini di memoria non cresce esponenzialmente, bensì **linearmente** rispetto alla profondità massima raggiunta.

D'altra parte, la **ricerca in ampiezza** non corre il rischio di andare all'infinito, poiché esplora sistematicamente gli stati a partire dal più vicino. Se esiste una soluzione più vicina allo stato iniziale, questa sarà trovata per prima, garantendo l'ottimalità in termini di lunghezza del cammino. Tuttavia, man mano che i livelli aumentano, la **frontiera** della ricerca cresce **esponenzialmente**, diventando un problema significativo per la memoria, soprattutto nei casi di problemi reali con spazi di stati molto grandi.

Uniform Cost Search

Quando le azioni presentano costi differenti è possibile applicare una ricerca basandosi sul *costo minore*.

L'algoritmo può essere visto come una ricerca ***best-first*** la cui *funzione di valutazione* è dettata dal *costo del cammino minore*.

Identificando con C^* il costo della soluzione ottima e con ϵ un *lower bound* sul costo delle azioni, avremo come caso peggiore $O(b^{\frac{C^*}{\epsilon}})$ che può essere anche molto maggiore b^d .

Questo avviene in quanto con la UCS è possibile che si espandano alberi molto grandi prima di andare a valutare azioni che hanno un costo maggiore ma portano più facilmente al *goal*.

Il vantaggio che si ha è che sicuramente, se c'è una soluzione la trova, dunque è *completo*.

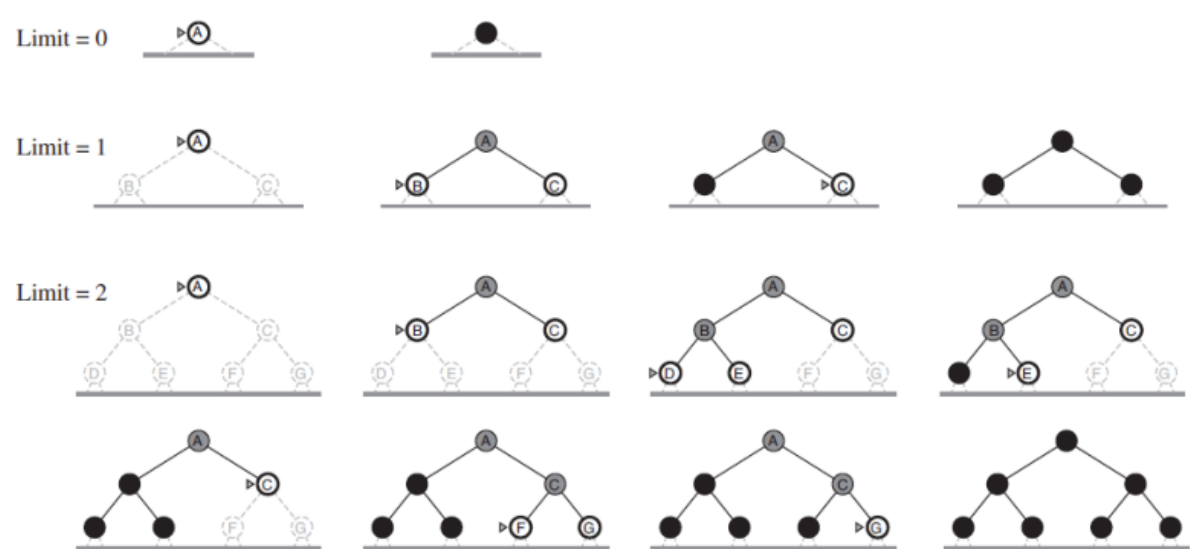
Iterative Deepening


È una strategia utilizzata in combinazione con la *ricerca in profondità*.

Il problema principale della *ricerca in profondità* è data dall'impossibilità di poter garantire l'ottimalità, per risolvere questo aspetto si fa un'analisi nella quale si incrementa man mano il livello di profondità a cui l'algoritmo deve arrivare fino a quando non si trova il goal.

Supponendo che l sia la profondità massima raggiunta, come nella ricerca in ampiezza, se fino alla profondità $l-1$ non è stata trovata una soluzione e alla profondità l invece sì, allora è la soluzione ottima.

Il *costo temporale* è $O(b^l)$, il *costo spaziale* è $O(bl)$.



 Nel caso peggiore la **iterative deepening** ha asintoticamente lo stesso costo della **ricerca in ampiezza**.
In quel caso il valore di l è pari alla profondità massima dell'albero.

Bi-Directional Search

Se il nodo obiettivo è noto, è possibile eseguire una ricerca partendo non solo dallo stato iniziale, ma anche dal goal stesso.

Questo approccio sfrutta il fatto che, in un **albero**, ogni nodo ha un solo genitore, e non ci sono ambiguità nelle scelte dei cammini.

Anche se lo spazio degli stati non è propriamente un albero, può comunque avere una struttura simile. Per migliorare l'efficienza della ricerca, si può utilizzare una strategia bidirezionale: si avviano contemporaneamente due ricerche, una dallo stato iniziale e una dal goal, e si portano avanti entrambe finché è conveniente.

L'obiettivo è che le due ricerche si incontrino, ottenendo una soluzione completa combinando le soluzioni parziali delle due espansioni.

In questo approccio, si utilizzano due **code di priorità** (una per ciascuna direzione). La ricerca procede come segue:

1. Si prende un nodo dalla frontiera di una delle due direzioni (la scelta può essere basata su un criterio di priorità o euristica).
2. Si espandono i figli di quel nodo.
3. Se un figlio non è ancora stato raggiunto dalla ricerca, o se la nuova strada ha un costo inferiore rispetto a un cammino precedente, il figlio viene aggiunto alla frontiera e il costo associato viene aggiornato.
4. A questo punto, si controlla se il nodo espanso è già stato raggiunto dalla ricerca che procede nella direzione opposta (dall'altra frontiera).
 - Se il nodo è stato già esplorato, significa che le due ricerche si sono incontrate, e si può costruire una **soluzione completa** combinando i percorsi parziali ottenuti dalle due espansioni.

In questo modo, si riduce lo spazio di ricerca esplorato rispetto a una ricerca unidirezionale, potendo ottenere una soluzione più velocemente, specialmente quando lo spazio degli stati è grande.

Tabella Riassuntiva

	Breadth-First	Depth-First	Uniform Cost	Iterative Deepening	Bi-Directional
Completo	X		X	X	X
Ottimale	X		X	X	X

Informed Search

Gli algoritmi di ricerca informata fanno uso di una funzione **euristica** denotata come $h(n)$.



La *funzione euristica* $h(n)$ rappresenta una stima del costo del cammino meno costoso dal *nodo* n ad uno *stato obiettivo*.

NOTA: La funzione euristica non è necessariamente precisa, può fornire anche stime non corrette.

Ricerca Greedy

La ricerca si basa totalmente sul valore della *funzione euristica*.

Il nodo espando è sempre quello che ha il valore di $h(n)$ più basso.

Dunque la funzione di valutazione $f(n)$ coincide con $h(n)$.

Risulta essere *completo* in stati degli spazi finiti, ma in stati infiniti non è garantito trovare una soluzione.

Il costo nel caso peggiore è pari a $O(|V|)$, ma con una buona euristica può scendere a $O(bm)$.

A^* e Tree Search

Rappresenta la forma più nota di *algoritmo di ricerca informata*.

La funzione di valutazione è così definita:

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

- $h(n)$ rappresenta la stima del costo che si dovrebbe avere dal nodo n fino al goal.
- $g(n)$ rappresenta il costo accumulato a partire dal nodo radice fino al nodo n .

La ricerca con A^* tramite *tree search* è sempre *completa*, ma non è detto che sia *ottimale*.

Per far sì che sia ottimale è necessario che l'euristica rispetti due caratteristiche: deve essere **ammissibile**.

Euristica Ammissibile



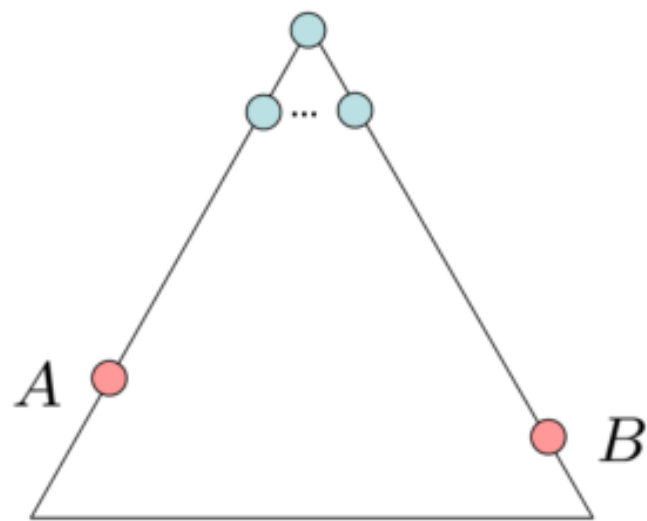
Un'euristica si definisce ammissibile quando: $h(n) \leq h^*(n)$

Dunque la stima data dall'euristica non deve mai *sovrastimare* il valore vero per raggiungere il *goal*.

Deve essere *ottimistica*.

Non è una condizione necessaria però vale $EuristicaNonAmmissibile \not\Rightarrow Ottimalità$.

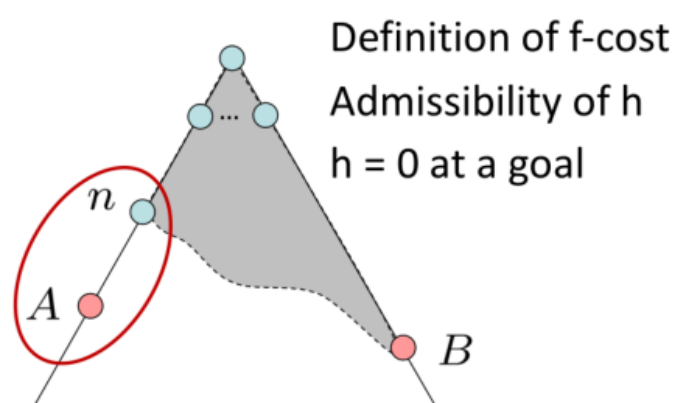
Ottimalità A^* in Tree Search



Stato iniziale

Si suppone di avere un albero in cui il *nodo A* rappresenta l'**ottimo**, invece il *nodo B* rappresenta un *sub-ottimo*, perciò vale: $g(A) \leq g(B)$ [1]

Per ipotesi l'euristica è **ammissibile**.



Al generico passo N il valore del *nodo B* è sulla *frontiera* e lo è anche un *nodo n* che è **antenato** di **A**.

Essendo un antenato sicuramente farà parte del *path* per raggiungere l'ottimo.

Da queste considerazioni è possibile ricavare:

- $f(n) = g^*(n) + h(n) \leq g^*(n) + h^*(n) = f(A)$ (Data l'ammissibilità) [2]

Per l'ipotesi di euristica ammissibile il valore di $f(n) \leq f(A)$

Per quanto riguarda il *nodo B* avremo:

- $f(B) = g^*(B) + h(B)$

siamo sicuri che il valore della funzione g fino al *nodo B* sia quella ottima perchè è nella *frontiera* assieme al *nodo n*.

Però nel [punto \[2\]](#) è stato visto come $f(n) \leq f(A)$, ma nel [punto \[1\]](#) è stato visto che

$$f(A) \leq f(B).$$

Unendo questi due risultati si ricava: $f(n) \leq f(A) < f(B)$

Di conseguenza non è possibile che sia stato espanso prima il *nodo B* in quanto ha un costo maggiore del *nodo n*.

▼ Dominanza Tra Euristiche

Un metodo comunemente usato per ottenere euristiche più affidabili è quello di combinarle scegliendo il valore più alto tra di esse.

In particolare, se si hanno due euristiche ammissibili, la loro combinazione tramite il massimo è ancora ammissibile.

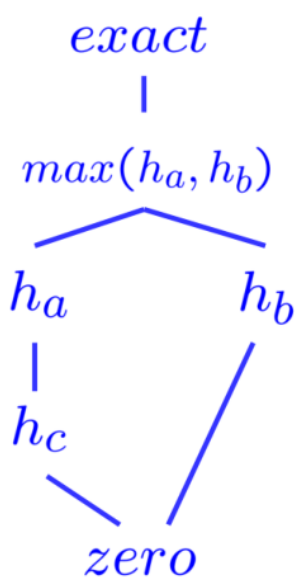
Questo approccio sfrutta il fatto che, tra più stime, la più alta rappresenta un vincolo più stretto mantenendo comunque la proprietà di ammissibilità.

Per questa metodologia $h(n)$ si sviluppa così: $h(n) = \max(h_a(n), h_b(n))$

Se tra due euristiche una è **sempre maggiore o uguale** rispetto all'altra, si dice che la prima **domina** la seconda.

In questo caso, non ha senso considerare la seconda euristica, perché quella dominante fornisce una stima migliore (più vicina al costo reale).

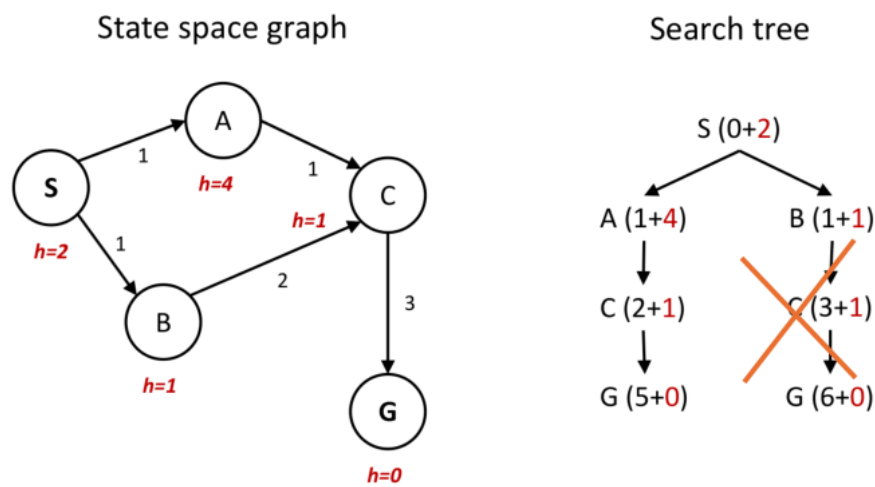
Si dice che l'**euristica A** domina l'**euristica B** se: $\forall n : h_a(n) \geq h_b(n)$



A* e Graph Search

Come analizzato in precedenza, il problema principale della *tree search* è l'incapacità di valutare se un nodo è stato esplorato più volte o meno, generando così un consumo di risorse molto elevato.

Utilizzando **graph search** è possibile risolvere questo problema, però è necessario sottolineare un aspetto: Come si fa ad essere sicuri che un nodo già visitato non avrebbe portato al goal in altre condizioni?



Esempio

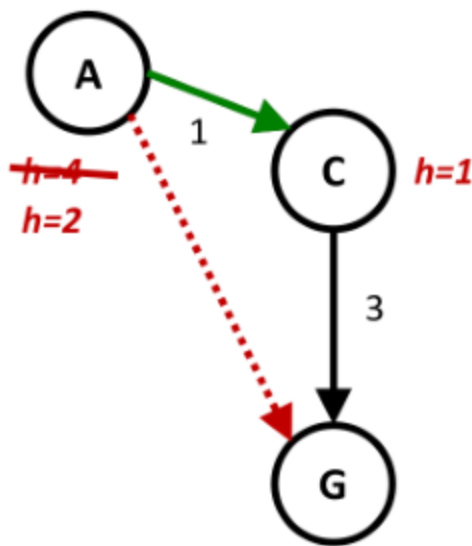
Per rispondere a questa domanda è necessario definire la **consistenza** per l'euristica.

Euristica Consistente



Un'euristica è **consistente** se, per ogni nodo n , e ogni successore n' generato da un'azione a vale che:

$$h(n) \leq c(n, a, n') + h(n')$$



In pratica stiamo affermando che vale la *disuguaglianza triangolare*.

Costo non Decrescente di $f(n)$ lungo un Path

La dimostrazione deriva dalla definizione di consistenza.

Se due nodi sono adiacenti avremo che:

$$cost(n \rightarrow n') = g(n') - g(n)$$

data la consistenza deve valere che:

$$h(n) - h(n') \leq g(n') - g(n)$$

Spostiamo i valori

$$g(n) + h(n) \leq g(n') + h(n')$$

Ambo i membri rappresentano, rispettivamente, i costi di $f(n)$ e di $f(n')$.

Dunque varrà:

$$f(n) \leq f(n')$$

Rapporto tra Ammissibilità e Consistenza

Una funzione **Ammissibile** non è detto che sia anche **Consistente**, ma è possibile dimostrare come valga *Consistenza* \implies *Ammissibilità*.

→ Caso Base

Supponiamo di avere due nodi n la cui distanza dal *goal* è minima. (Siamo dunque a distanza 1 dal goal).

Per la *consistenza* si avrà:

$$h(n) \leq c(n, a, G) + h(G)$$

Ma dato che G è il goal allora $h(G) = 0$, mentre $c(n, a, G)$ è il costo reale per arrivare al *goal*, di conseguenza $h(n)$ sta sottostimando il valore reale per raggiungere il *goal*.

$$h(n) \leq h^*(n)$$

→ Induzione

Per ipotesi induttiva questa affermazione dovrà essere vera per ogni stato n' con costo $h(n') \leq k$.

Si deve dimostrare che vale per successivo valore di costo identificato con k' .

Dunque si dovrà avere uno stato con $h(n) = k'$.

Per ipotesi di **consistenza** si avrà:

$$h(n) \leq c(n, a, n') + h(n')$$

In particolare ciò varrà anche per la **migliore azione possibile per arrivare da n ad n'** .

Dunque varrà:

$$h(n) \leq c(n, a^*, n') + h(n')$$

Per ipotesi induttiva vale che:

$$h(n') \leq h^*(n')$$

Ottenendo dunque:

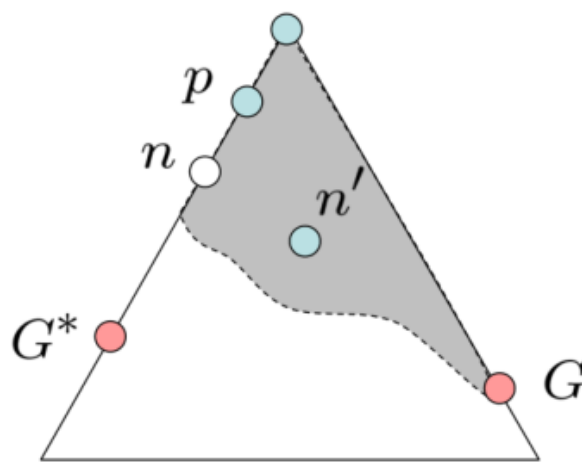
$$h(n) \leq c(n, a^*, n') + h^*(n')$$

Dunque ricaviamo ciò che si voleva dimostrare:

$$h(n') \leq h^*(n')$$

Questo varrà per ogni nodo che dista 1 da n e dunque varrà per ogni nodo del grafo.

Ottimalità di A^* Graph Search



G rappresenta un nodo *sub-ottimale*, invece G^* rappresenta l'*ottimo*.

Supponendo che n ed n' siano lo stesso stato.

Supponendo per assurdo che sia stato esplorato prima il *nodo* n' che porta al sub-ottimo, idealmente (essendo in un grafo) non potremmo andare ad esplorare il *nodo* n che invece ci avrebbe portato all'ottimo.

Sia p l'antenato di n che si trovava sulla frontiera allo stesso momento di n' essendo però scartato.

→ Dallo schema mostrato è stato preferito il nodo n' rispetto al *nodo* p . Dunque deve valere che: $f(n') \leq f(p)$

per la consistenza sappiamo che man mano che si scende nell'albero il costo è non decrescente quindi $f(p) \leq f(n)$.

Da queste disuguaglianze si ottiene:

$$f(n') \leq f(n)$$

Tuttavia, sappiamo che n è un nodo che conduce alla soluzione **ottimale** G^* , mentre n' conduce a una soluzione **sub-ottimale** G .

Per definizione, il costo totale del percorso che passa per n deve essere inferiore (o al massimo uguale) rispetto a qualsiasi altro percorso sub-ottimale.

Dunque, il valore di $f(n)$, che rappresenta il costo stimato di raggiungere la soluzione ottima partendo da n , dovrebbe essere **minore** rispetto a $f(n')$, che conduce a una soluzione sub-ottimale.

Search in Complex Environments

Molto spesso capita di essere interessati solo al ***risultato finale*** e non al percorso per raggiungerlo.

Per risolvere questa tipologia di problemi si utilizzano gli *algoritmi di ricerca locale*. Questi operano a partire da uno stato iniziale e si spostano negli stati adiacenti senza però ricordare quelli in cui sono già stati e neanche del cammino effettuato.

Con questa metodologia può capitare che non si vada ad analizzare una porzione di albero che contiene la soluzione.

Il vantaggio molto forte di questi algoritmi è dettato dall'utilizzo irrisorio di memoria e spesso riescono a trovare soluzioni anche in spazi degli stati infiniti.

Hill Climbing

L'algoritmo di ricerca Hill Climbing; tiene traccia di un solo stato corrente e a ogni iterazione passa allo stato vicino con valore più alto, cioè punta nella direzione che presenta l'ascesa più ripida (steepest ascent), senza guardare oltre gli stati immediatamente vicini a quello corrente.

```
function HILL-CLIMBING(problema) returns uno stato che è un massimo locale
  corrente ← problema.STATOINIZIALE
  while true do
    vicino ← lo stato successore di corrente di valore più alto
    if VALORE(vicino) ≤ VALORE(corrente) then return corrente
    corrente ← vicino
```

Versione Greedy

L'algoritmo **hill climbing** è talvolta chiamato **ricerca locale greedy** perché sceglie uno stato vicino "buono" senza considerare come andrà avanti.

L'Hill Climbing può procedere rapidamente verso la soluzione, poiché è generalmente abbastanza facile migliorare uno stato sfavorevole.

Sfortunatamente, l'Hill Climbing spesso rimane bloccato per le seguenti ragioni:

- **Massimi locali:** un massimo locale è un picco più alto degli stati vicini, ma inferiore al massimo globale. Gli algoritmi Hill Climbing che raggiungono la vicinanza di un massimo locale saranno attratti verso il picco, ma rimarranno bloccati lì senza poter andare altrove.
- **Creste:** una cresta (ridge) dà origine a una sequenza di massimi locali molto difficili da esplorare da parte degli algoritmi greedy.
- **Plateau:** un plateau è un'area piatta del panorama dello spazio degli stati. Può essere un massimo locale piatto, da cui non è possibile fare ulteriori progressi, oppure una spalla (shoulder), da cui si potrà salire ulteriormente. Una ricerca Hill Climbing potrebbe perdersi sul plateau.

In ognuno di questi casi, l'algoritmo raggiunge un punto dal quale non riesce a compiere ulteriori progressi.

Simulated Annealing

Un algoritmo **hill climbing** che non scende mai "a valle" verso stati con valore più basso (o costo più alto) è sempre vulnerabile alla possibilità di rimanere bloccato in corrispondenza di un massimo locale.

D'altro canto, un'esplorazione del tutto casuale che si muove verso uno stato successore senza preoccuparsi del valore finirà per incontrare il massimo globale, ma sarà estremamente inefficiente. Sembra ragionevole, quindi, cercare di combinare in qualche modo l'Hill Climbing con un'esplorazione casuale, al fine di ottenere sia l'efficienza che la completezza.

Un algoritmo che fa proprio questo è il

simulated annealing (letteralmente, "tempra simulata").

In metallurgia, l'annealing (tempra) è il processo usato per indurire i metalli o il vetro riscaldandoli ad altissime temperature e raffreddandoli gradualmente, permettendo così

al materiale di cristallizzare in uno stato a bassa energia. La soluzione proposta dal simulated annealing è di cominciare con un'alta temperatura e poi ridurre gradualmente l'intensità della temperatura.

La struttura complessiva dell'algoritmo simulated annealing è simile a quella dell'Hill Climbing; stavolta però, invece della mossa migliore, viene scelta una mossa casuale.

Se la mossa migliora la situazione, viene sempre accettata; in caso contrario, l'algoritmo la accetta con una probabilità inferiore a 1.

La probabilità decresce esponenzialmente con la "cattiva qualità" della mossa, misurata dal peggioramento ΔE della valutazione. La probabilità decresce anche con la "temperatura" T , che scende costantemente: le mosse "cattive" saranno accettate più facilmente all'inizio, in condizioni di alta T , e diventeranno sempre meno probabili man mano che T si abbassa.

Se la velocità di raffreddamento fa decrescere la temperatura da T a 0 abbastanza lentamente, per una proprietà della distribuzione di Boltzmann, $e^{\Delta E/T}$, tutta la probabilità è concentrata sui massimi globali, che l'algoritmo troverà con probabilità tendente a 1.

Ricerca con Azioni non Deterministiche

Quando l'ambiente è parzialmente osservabile, l'agente non sa con certezza in quale stato si trova; e quando l'ambiente è non deterministico, l'agente non sa in quale stato arriverà dopo l'esecuzione di un'azione.

Questo significa che, anziché pensare "Sono nello stato $s1$ e se eseguo l'azione a andrò nello stato $s2$ ", l'agente penserà "Sono nello stato $s1$ o $s3$, e se eseguo l'azione a passerò nello stato $s2$, $s4$ o $s5$ ".



Chiamiamo **stato-credenza** un insieme di stati fisici che l'agente ritiene siano possibili.

In ambienti parzialmente osservabili e non deterministici, la soluzione di un problema non è una sequenza ma un **piano condizionale** (piano di contingenza o strategia) che specifica cosa fare in base alle percezioni ricevute dall'agente durante l'esecuzione del piano.

Alberi di ricerca AND-OR

In un ambiente deterministico, l'unica ramificazione è introdotta dalle scelte dell'agente in ogni stato: "Posso fare questa azione o quella"; parliamo in questo caso di **nodi OR**.

In un ambiente non deterministico, la ramificazione è anche legata alla scelta del risultato per ogni azione, effettuata dall'ambiente. In questo caso parliamo di **nodi AND**.

Questi due tipi di nodi si alternano, generando un **albero AND-OR**.

Una soluzione per un problema di ricerca AND-OR è un sottoalbero dell'albero di ricerca completo che

1. Ha un nodo obiettivo in ogni foglia,
2. Specifica una sola azione in ognuno dei suoi nodi OR
3. Include ogni ramo uscente da ognuno dei suoi nodi AND.

Un aspetto chiave dell'algoritmo è il modo in cui gestisce i cicli, che spesso si presentano in problemi non deterministici: se lo stato corrente è identico a uno stato sul cammino dalla radice, allora l'algoritmo termina con un fallimento.

Questo non significa che non esista alcuna soluzione dallo stato corrente, ma soltanto che, se esiste una soluzione non ciclica, deve essere raggiungibile dalla precedente incarnazione dello stato corrente, perciò la nuova incarnazione può essere scartata.

Con questo controllo ci assicuriamo che l'algoritmo termini in ogni spazio degli stati finito, perché ogni cammino deve raggiungere un obiettivo, un vicolo cieco o uno stato ripetuto.

Notate che l'algoritmo non controlla se lo stato corrente è una ripetizione di uno stato in qualche altro cammino dalla radice, cosa importante per l'efficienza.

I grafi AND-OR possono essere esplorati dai metodi in ampiezza o best-first.

Il concetto di funzione euristica deve essere modificato per stimare il costo di una soluzione contingente anziché di una sequenza, ma la nozione di ammissibilità viene mantenuta ed esiste un analogo dell'algoritmo A* per trovare soluzioni ottime.

Adversial Search

Esistono diverse tipologie di giochi:

- Deterministici o Casuali
- Con due o più giocatori
- A **Somma Zero**: questa tipologia indica quei giochi a **turni** in cui uno dei giocatori vince e l'altro perde ed ogni mossa che un giocatore sceglie mira a danneggiare l'avversario.
Inoltre, sono anche i giochi ad *informazione perfetta* in cui in ogni momento è noto lo stato del gioco in quanto *completamente osservabile*.

Inoltre per questi giochi le azioni prendono il nome di *mosse* e gli stati prendono il nome di *posizione*.

L'albero generato dalla combinazione di **azioni** e **stati** prende il nome di **albero di gioco**.

Ricerca Min-Max

Nel caso dei giochi a *Somma Zero* è possibile identificare due giocatori che partecipano: Il giocatore **Min** e il giocatore **Max**.

Il primo desidera *minimizzare* il punteggio delle sue azioni, invece il secondo desidera *massimizzare* il suo punteggio.

Come detto precedentemente nei giochi a somma zero si hanno dei turni, dunque Min e Max si alternano costantemente all'interno dell'albero.

Dunque le strategie adottate da entrambi i giocatori sono relative ai comportamenti dei propri avversari.

Dato un albero di gioco è possibile determinare la *strategia ottima* calcolando il valore di *minmax* di ogni stato.



Il valore *minmax* è l'utilità ottenuta nel trovarsi in quel determinato stato.

MinMax funziona correttamente di fronte a due avversari che giocano in *maniera ottimale* e dunque non sbagliano mai alcuna mossa.

Questo aspetto è fondamentale in quanto la scelta della mossa si baserà unicamente sull'*utilità* che può portare questa, dunque non c'è un margine di errore.

Nel caso in cui l'avversario, MIN, ***non giocasse in modo ottimale*** la situazione cambierebbe.

Infatti, in quel caso sarebbe possibile per MAX *"rischiare"* delle mosse che potrebbero portarlo più facilmente alla vittoria.



Se l'avversario gioca in maniera *sub-ottimale* non è detto che effettuare sempre la mossa ottima sia una strategia vincente.

Ovviamente questo aspetto è totalmente nelle mani di MAX che deve valutare se l'avversario non ha sufficiente potenza di calcolo per identificare la mossa corretta nel tempo prestabilito e dunque scegliere una mossa più azzardata.

In generale MinMax trova la soluzione ottima rispetto al comportamento ottimo dell'avversario.

Se l'avversario non gioca in maniera ottimale allora si avrà un punteggio

\geq al punteggio che si otterrebbe contro un giocatore ottimo.

L'algoritmo di ricerca utilizzato è relativamente molto semplice.

Si percorre l'albero e tramite un'operazione di ***back-up*** si riportano i valori trovati alle foglie verso la radice.

Dunque MinMax esegue un'esplorazione completa in profondità dell'*albero di gioco*.

Supponendo che la *profondità massima* sia m e che per ogni punto ci sono b mosse legali la complessità temporale dell'algoritmo sarà $O(b^m)$, quella spaziale invece $O(b * m)$.


```
function RICERCA-MINIMAX(gioco, stato) returns un'azione
  giocatore  $\leftarrow$  gioco.DEVE-MUOVERE(stato)
  valore, mossa  $\leftarrow$  VALORE-MAX(gioco, stato)
  return mossa

function VALORE-MAX(gioco, stato) returns una coppia (utilità, mossa)
  if gioco.È-TERMINALE(stato) then return gioco.UTILITÀ(stato, giocatore), null
  v  $\leftarrow -\infty$ 
  for each a in gioco.AZIONI(stato) do
    v2, a2  $\leftarrow$  VALORE-MIN(gioco, gioco.RISULTATO(stato, a))
    if v2 > v then
      v, mossa  $\leftarrow$  v2, a
  return v, mossa

function VALORE-MIN(gioco, stato) returns una coppia (utilità, mossa)
  if gioco.È-TERMINALE(stato) then return gioco.UTILITÀ(stato, giocatore), null
  v  $\leftarrow +\infty$ 
  for each a in gioco.AZIONI(stato) do
    v2, a2  $\leftarrow$  VALORE-MAX(gioco, gioco.RISULTATO(stato, a))
    if v2 < v then
      v, mossa  $\leftarrow$  v2, a
  return v, mossa
```



In giochi complessi MinMax non è un algoritmo utilizzabile in quanto non sarebbe in grado di esplorare per intero l'albero.

Potatura Alfa-Beta

Data la crescita esponenziale man mano che si scende nell'albero, anche per giochi molto semplici la ricerca potrebbe generare delle *difficoltà* nel calcolo.

Per questo motivo si utilizza la potatura, tramite la quale si eliminano molti spazi di ricerca diminuendo drasticamente il numero di nodi da analizzare.

Si utilizzano due valori:

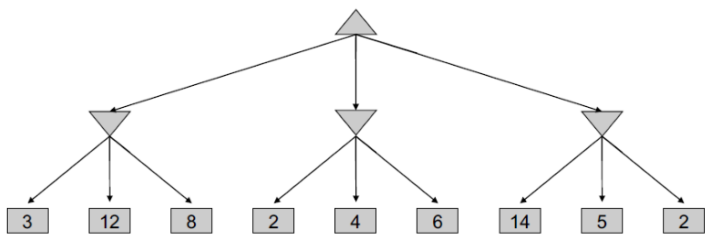
- α : il valore della scelta migliore per *MAX* trovata fino a quel punto durante l'esplorazione
- β : il valore della scelta migliore per *MIN* trovata fino a quel punto durante l'esplorazione.

Questi valori vengono aggiornati man mano durante l'esplorazione.

Un sottoalbero viene potato quando si trova un valore in quel sottospazio che è $\leq \alpha$.

Ciò avviene perchè nel caso in cui si sia trovato un valore più basso di alpha nel sottoalbero esplorato, non ha senso procedere in quanto se scegliessimo quel ramo poi

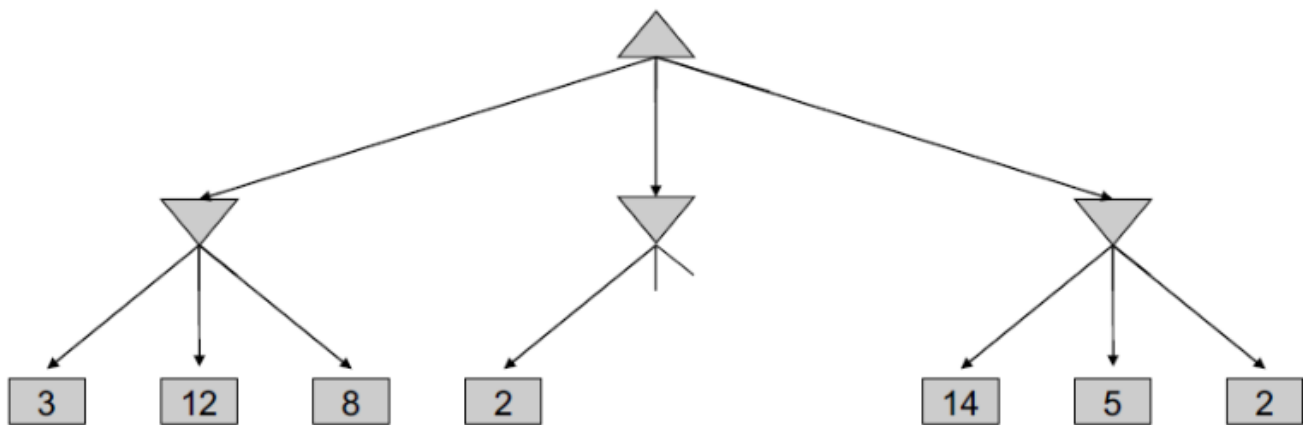
MIN sceglierebbe quel valore portandoci ad una situazione di svantaggio.



In questo esempio dal primo sottoalbero il valore migliore trovato è 3. (Il valore è 3 perchè la mossa successiva sarebbe di *MIN*).

Nel sotto-albero successivo, tenendo conto di 3 come valore migliore, si andrà ad esplorare e il primo valore ottenuto sarà 2.

Dato che $2 \leq 3$ non ha senso continuare ad esplorare e si portano evitare di analizzare i nodi vicini.



Nel terzo sotto-albero invece non ci saranno miglioramenti in quanto dall'esplorazione si otterranno:

- Primo nodo 14, e dato che $14 \geq 3$ si procede.
- Secondo nodo 5, e dato che $5 \geq 3$ si procede.
- Solo all'ultimo nodo si avrà il valore 2, ma nonostante $2 \leq 3$ non si apportano miglorie in quanto era già l'ultimo nodo.

Tramite la potatura nel caso migliore, cioè quando i nodi sono ordinati, è possibile ottenere un miglioramento temporale pari a $O(b^{m/2})$.

NOTA: è possibile ordinare i nodi tramite un'euristica ed ottenere dunque anche in quel caso un miglioramento notevole.

Funzione di Valutazione

Tramite una *funzione di valutazione* si restituisce una **stima euristica** sull'utilità attesa dello stato s per il giocatore p .

Si indica con $Eval(s, p)$.

Negli stati finali deve valere che $Eval(s, p) = Utility(s, p)$.

Solitamente le funzioni di valutazione si basano su alcuni *criteri* che prendono il nome di **caratteristiche**.

Queste possono essere ad esempio: *La posizione delle pedine, il numero di pedine a favore, ecc...*

Le caratteristiche unite prendono il nome di *categorie*.

Gran parte delle funzioni di valutazione effettuano calcoli considerato le caratteristiche come indipendenti per poi combinarle assieme.

$$Eval(s) = w_1 f_1(s) + \dots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s)$$

Dove:

- Ogni f_i è una caratteristica.
- Ogni w_i è un peso ed indica l'importanza della caratteristica.

Considerarle indipendenti è un comportamento errato, infatti negli ultimi tempi si stanno utilizzando *combinazioni non lineari* di caratteristiche.

Dunque, si può affermare che le *funzioni di valutazione* siano *imperfette*.

L'**effetto orizzonte** è particolarmente difficile da eliminare. Questo problema si verifica quando il programma deve valutare una mossa dell'avversario che causa un grave danno inevitabile, ma che può essere temporaneamente posticipata usando tattiche dilatorie.

Una strategia per attenuare l'effetto orizzonte consiste nell'utilizzare le estensioni singole: mosse che sono chiaramente migliori rispetto a tutte le altre in una determinata posizione. Anche se normalmente la ricerca si fermerebbe a quel punto, queste estensioni consentono di esplorare l'albero più a fondo. Poiché solitamente ci sono poche estensioni singole, questa tecnica non aggiunge troppi nodi all'albero e si è dimostrata efficace in pratica.

▼ Effetto Orizzonte

L'effetto orizzonte si verifica in sistemi di ricerca e valutazione, come quelli utilizzati negli scacchi o in altri giochi a turni, quando un motore di intelligenza artificiale non è in grado di vedere abbastanza in profondità

nell'albero delle mosse future per valutare correttamente una situazione negativa imminente. In pratica, il motore può riconoscere che si sta avvicinando una minaccia, come la perdita di un pezzo o una mossa vincente dell'avversario, ma non può "vedere oltre l'orizzonte" della profondità di ricerca attuale.

A causa di questo limite, il motore potrebbe scegliere di ritardare l'inevitabile con mosse che posticipano l'esito negativo, anche se queste mosse non cambiano davvero la situazione. Questo porta a valutazioni ottimistiche e temporanee, dove sembra che il programma stia "evitando" il problema, quando in realtà sta solo rimandando il danno.

Per mitigare questo problema, si possono usare tecniche come le **estensioni singole**, che consentono di esplorare alcune mosse chiave più a fondo rispetto alla profondità di ricerca standard, aumentando la precisione delle valutazioni del motore senza incrementare significativamente il numero totale di nodi esplorati.

Ricerca Monte-Carlo

La strategia di Monte-Carlo non si basa su una funzione di valutazione euristica, ma utilizza dei valori di utilità ottenuti tramite **simulazioni**.



Una **Simulazione** prende in considerazione le mosse proprie e dell'avversario e le esegue fino a quando non si ottiene un risultato terminale.

Quindi tramite le simulazioni non è l'euristica a scegliere il vincitore, bensì le regole del gioco stesso.

L'aspetto fondamentale di MonteCarlo è la scelta della *politica di simulazione*.

Una volta selezionata è sufficiente determinare quale e quante simulazioni eseguire al fine di definire la mossa corretta.

Si ha:

Monte Carlo Puro: in cui si eseguono N simulazioni a partire dallo stato corrente del gioco.

Il punto critico sta nel decidere se favorire l'**esplorazione** oppure la **simulazione**.

Gli step seguiti sono i seguenti:

1. **Selezione** del nodo.
2. **Espansione** del nodo con la seguente generazione dei figli.
3. **Simulazione** sul nodo selezionato si eseguono diverse simulazioni per stabilire se è conveniente o meno la scelta dell'azione.
4. **Retropropagazione** dei risultati ottenuti dalla simulazione.

Upper Confidence Bounds Applied to Trees (UCT)

Una **politica di selezione** molto utilizzata è la *UCT*.

Questa classifica le mosse basandosi sulla seguente formula:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \sqrt{\frac{\log N(PADRE(n))}{N(n)}}$$

Dove

- $U(n)$ identifica il numero di vittorie.
- $N(n)$ identifica il numero di partite giocate
- $\frac{\log N(PADRE(n))}{N(n)}$ identifica il numero di volte in cui è stato scelto il padre diviso il numero di partite giocate.

Con questa politica si deve tenere conto di un *trade-off* tra esplorazione e simulazione.

Il termine sotto la radice rappresenta il termine di *esplorazione*.

Se un nodo n è stato selezionato frequentemente, questo valore diminuirà, favorendo l'esplorazione di nodi meno visitati.

Il primo termine rappresenta invece la *simulazione*. Più alto è il numero di vittorie rispetto alle simulazioni, maggiore sarà il valore di questo termine, spingendo la politica a scegliere mosse che hanno storicamente portato a vittorie..

Il termine C rappresenta un valore di bilanciamento che solitamente è identificato con $\sqrt{2}$ ma nella pratica si utilizzano diversi termini.

Monte Carlo vs Alpha-Beta

Caratteristica	Monte Carlo	Alpha-Beta
Ambito di utilizzo ideale	Giochi con alto fattore di ramificazione (es. Go); giochi nuovi o difficili da valutare.	Giochi con fattore di ramificazione inferiore e funzione di valutazione accurata.
Dipendenza dalla funzione di valutazione	Non richiede una funzione di valutazione precisa.	Fortemente dipendente dalla funzione di valutazione. Un errore può portare a decisioni sbagliate.
Metodo di selezione delle mosse	Basata su simulazioni multiple, aggregando i risultati per scegliere le mosse migliori.	Sceglie il cammino con il punteggio più alto, assumendo che l'avversario cerchi di minimizzare il punteggio.
Robustezza contro errori	Meno vulnerabile agli errori, poiché basata su molte simulazioni e non su singoli nodi.	Vulnerabile agli errori di valutazione su singoli nodi, che possono compromettere l'intera ricerca.
Approccio all'esplorazione vs sfruttamento	Bilancia esplorazione e sfruttamento tramite politiche come UCB1 (esplora nuovi nodi e sfrutta quelli già noti).	Maggiore enfasi sullo sfruttamento, cercando di massimizzare il punteggio basato sulla funzione di valutazione.
Termine anticipato delle simulazioni	Possibile interrompere simulazioni lunghe e applicare una funzione euristica o dichiarare la parità.	Si basa su una profondità di ricerca predefinita, senza possibilità di interruzione anticipata basata sul numero di mosse.
Applicazione a nuovi giochi	Può essere applicata senza esperienza precedente, basta conoscere le regole del gioco.	Richiede una funzione di valutazione ben definita, quindi meno adatta a giochi nuovi.
Svantaggi principali	Può trascurare mosse critiche a causa della sua natura stocastica.	Sensibile a errori nella funzione di valutazione, specialmente in giochi con molta incertezza.
Combinazione con altre tecniche	Può essere combinata con funzioni di valutazione per simulazioni più brevi, o con reti neurali.	Possibile combinazione con tecniche euristiche, ma meno flessibile nell'integrazione con approcci stocastici.

Uncertainty and Utilities

I giochi trattati fino ad ora sono contor avversari e con mosse definite da regole precise.

Ma cosa succede se si aggiunge una componente di *incertezza e probabilità*?

Quindi le mosse dettate dagli avversari non saranno più definite sulla base di “miro a far perdere l'avversario” ma si metterà in gioco anche una certa distribuzione di probabilità.

Esistono due possibili approcci a queste tipologie di giochi:

- **Worst-Case:** Si assume sempre che avvenga il caso peggiore.
- **Average-Case:** La scelta è legata alla probabilità e dunque si assume del rischio.

Dato che non si hanno più dei valori ben precisi della tipologia *minmax* si farà uso del **valore atteso**, cioè la media su tutti i possibili risultati dei nodi di casualità.

ExpectiMax Search

Non è altro che una modifica dell'algoritmo *minmax* dove si considera il **valore atteso**.

La procedura di ricerca non si differenzia particolarmente, ma bisogna tenere conto che invece dei valori di *min* e *max* precedentemente definiti, si ha un valore ottenuto considerando le probabilità che prende il nome di *utilità attesa*.

Il problema principale è dovuto all'impossibilità di effettuare un *pruning* in quanto non si lavora con valori certi ma con **possibili valori**.

Se ci fosse un upper bound o un lower bound però sarebbe possibile applicarlo.



Dato che si considerano oltre alle possibili mosse anche **tutte** le possibili combinazioni di mosse allora si dovrà tenere conto di un costo pari a:

$$O(b^m n^m)$$

La probabilità utilizzata dipende molto dal sistema che utilizziamo, infatti, in un caso con i dati ad esempio si potrebbe utilizzare una **distribuzione uniforme**, in altri casi si potrebbe utilizzare una distribuzione più complessa con molti più calcoli.

Teoria dell'Utilità e della Probabilità

Per ottenere un agente che è in grado di prendere decisioni tenendo conto dell'eventuali probabilità presenti deve basarsi sulla *teoria delle decisioni* e dunque saper scegliere in situazioni in cui non esiste uno *stato assoluto* migliore rispetto ad altri.



Il principio della *Massima utilità attesa (MEU)* afferma che un agente razionale dovrebbe scegliere l'azione che *massimizza l'utilità attesa*.

Però selezionare le *utilità* corrette non è un'operazione facile per l'agente, in quanto ci potrebbe essere diverse situazioni che potrebbero avere *pro* e *contro* che si bilanciano e dunque sarebbe difficile scegliere.

Per questo motivo si utilizzano le **Preferenze**.

Basi Teoria dell'Utilità

Ci si basa su due simbolismi:

- $A > B$ l'agente preferisce *A* a *B*.
- $A \sim B$ l'agente è indifferente tra *A* e *B*.

Si possono pensare all'insieme degli esiti per ogni azione come una **lotteria** in cui ogni azione è un biglietto.

Una *Lotteria* L che ha i vari esiti possibili S con le associate *probabilità* p .

$$L = [p_1, S_1; \dots; p_n, S_n]$$

In particolare ogni Esito S potrebbe essere o uno Stato o un'altra lotteria.

Proprietà

1. Ordinalità

date due lotterie, un agente razionale deve preferire l'uno o l'altra o considerarle ugualmente preferibili. In altre parole, l'agente non può rifiutare di decidere, in quanto rifiutare è come cercare di impedire lo scorrere del tempo.

$$(A > B) \vee (A < B) \vee (A \sim B)$$

2. Transitività

$$(A > B) \wedge (B > C) \implies (A > C)$$

3. Continuità

La **continuità** afferma che, se l'agente preferisce A a B e B a C , allora esiste una probabilità p tale che l'agente sarà **indifferente** tra:

1. **Ottenere con certezza B** (cioè, il risultato B senza incertezze).
2. **Partecipare a una lotteria** dove c'è una probabilità p di ottenere A e una probabilità $1-p$ di ottenere C .

In altre parole, se B si trova "a metà strada" nelle preferenze tra A e C , possiamo sempre trovare una probabilità p per cui l'agente considererà equivalenti:

- il ricevere direttamente B ,
- oppure affrontare una situazione di incertezza dove c'è una chance p di ottenere il risultato migliore (A) e una chance $1-p$ di ottenere il risultato peggiore (C).

$$A > B > C \implies \exists p [p, A; 1 - p, C] \sim B$$

4. Sostituibilità

Se un agente è indifferente tra due lotterie A e B , allora l'agente sarà indifferente anche tra due lotterie più complesse che sono identiche se non per il fatto che in una di essere B è sostituito da A .

$$A \sim B \implies [p, A; 1 - p, C] \sim [p, B; 1 - p, C]$$

5. Monotonicità

Supponiamo di avere due lotterie con gli stessi esiti: A e B .

Se un agente preferisce

A a B , allora l'agente deve preferire la lotteria che ha una probabilità più alta per A .

$$A > B \wedge p > q \implies [p, A; 1 - p, B] > [q, A; 1 - q, B]$$

6. Scomponibilità

Lotterie complesse possono essere ridotte a lotterie più semplici.

$$[p, A; 1 - p, [q, B; 1 - q, C]] \sim [p, A; (1 - p)q, B]; (1 - p)(1 - q), C]$$

Se un agente obbedisce agli assiomi dell'utilità allora esiste una funzione U tale che

- $U(A) > U(B)$ se e solo se A è preferito a B .
 - $U(A) = U(B)$ se e solo se l'agente è indifferente tra A e B .
-

L'**utilità attesa di una lotteria** è la somma delle probabilità di ogni esito moltiplicata per l'utilità di tale esito:

$$U([p_1, S_1; \dots; p_n, S_n]) = \sum_i p_i U(S_i)$$

Per il principio di **MEU** si scelgono le azioni che massimizzano l'utilità attesa.

Scale di Utilità

Molto spesso però gli umani non si comportano in maniera totalmente razionale, per questo motivo si utilizzano le *scale di utilità*.

Se si usa la funzione di utilità:

$$U'(x) = k_1 U(x) + x_2$$

Questa si comporterà in maniera sempre uguale se due agenti sono inseriti nello stesso identico ambiente con le stesse convinzioni.

Per ovviare questo problema si possono utilizzare delle **funzioni di utilità normalizzate**:

- Si indica con u_{\perp} l'utilità dello stato *peggiore possibile*.

$$u_{\perp} = U(S_{\perp})$$

- Si indica con u_{\top} l'utilità dello stato *migliore possibile*.

$$u_{\top} = U(S_{\top})$$

Per la funzione di utilità normalizzata si avrà: $U'(S_{\perp}) = 0 \wedge U'(S_{\top}) = 1$.

Le **utilità** degli stati intermedi $U'(S)$ sono determinate dalla probabilità p , tale per cui per l'agente non vi è differenza tra lo stato S e una **lotteria standard**.

$$U'(S) = p \text{ quando } S \sim [p, S_{\top}; 1 - p, S_{\perp}]$$



In pratica le **Utility Scales** servono per rappresentare le preferenze di un individuo su un insieme di premi e lotterie.

In particolare si avranno:

- **Scale cardinali**: che assegnano un valore numerico assoluto a ciascun premio.
- **Scale ordinali**: assegnano un valore numerico a ciascun premio, ma i valori numeri non hanno un significato assoluto.

▼ Significato in assoluto

In breve, un significato assoluto vuol dire che i valori numerici rappresentano grandezze misurabili e confrontabili in modo oggettivo. Nelle scale cardinali, i numeri hanno un significato intrinseco: ad esempio, il premio 10 vale esattamente il doppio del premio 5.

Invece, nelle scale ordinali, i numeri rappresentano solo l'ordine o la posizione relativa tra i premi (es. primo, secondo, terzo), ma non danno informazioni sulla distanza o sulla proporzione tra i premi.

Constraints Satisfaction Problem

A differenza dei problemi analizzati sino ad ora, in questa tipologia si presentano dei **vincoli** nelle variabili.

Dunque l'obiettivo di questi problemi è quello di andare ad associare ad ognuna delle N *variabili* presenti un valore garantendo il rispetto di ognuno dei vincoli.

Si possono avere due tipologie di problemi CSP:

- **Weak Constraints:** In questa tipologia non è necessario che tutti i vincoli vengano rispettati, in questo modo è possibile *rilassare* il problema è renderlo un semplice problema di PL (ottimizzazione).
- **Strong Constraints:** Tutti i vincoli devono essere necessariamente rispettati.

In generale un CSP possiede le seguenti caratteristiche:



- X è un insieme di variabili $\{X_1, \dots, X_n\}$
- D è un insieme di Domini associato uno ad ogni variabile $\{D_1, \dots, D_n\}$
- C un insieme di vincoli che specificano le compatibilità sui valori ammessi

In particolare i vincoli possono essere di diverso tipo:

1. **Unari:** coinvolgono un'unica variabile, ad esempio $A = green$
2. **Binari:** Coinvolgono due variabili differenti, ad esempio $A \neq B$
3. **Ordine Superiore:** Coinvolgono da tre a più variabili.

Associare un *valore* ad una *variabile* prende il nome di **assegnamento**.

Quando un **assegnamento** non viola alcun vincolo allora questo è definito come **legale** o **consistente**.

Un assegnamento risulta essere *completo* quando a tutte le variabili è stato assegnato un valore.

Quando un assegnamento è **completo** e **consistente** allora è una soluzione per il CSP.

Se invece ciò non accade e si riescono ad assegnare solo alcune variabili allora l'assegnamento è **parziale**.

In generale risolvere un problema CSP è **NP-Completo**.

Istanza di un Constraint Satisfaction Problem

Un problema di tipo CSP è definito dalla coppia

$$I = (\overline{C}, DB)$$

in particolare:

- $\overline{C} = \{r_1(\overline{x_1}), \dots, r_m(\overline{x_m})\}$ è l'insieme dei vincoli.
- $DB =$ relations r_1, \dots, r_m che è l'insieme delle relazioni con le rispettive arità.

Inoltre si hanno:

- V come set di variabili
- D insieme di valori ammissibili ottenuti dal DB .

Il contenuto del DB è pressoché identico a quello di un comune database relazionale: all'interno sono presenti le *signature* delle relazioni.

La **signature** è una struttura formale che definisce lo schema di un database o di un sistema di vincoli, descrivendo le relazioni disponibili e le loro caratteristiche. Ogni relazione nella signature è identificata da un nome e da un'arità, che indica il numero di elementi coinvolti nella relazione.

Nel database, ogni relazione associata alla signature contiene i dati compatibili con questa struttura, ovvero insiemi di tuple che rispettano le specifiche di arità.

Soluzione di un Constraint Satisfaction Problem

Formalmente una soluzione è un **Mapping** che va dall'insieme delle variabili al dominio delle variabili in modo da soddisfare i vincoli.

$$\sigma : V \implies D, \forall c_i \in \overline{C} \sigma(c_i) \in r_{c_i}$$

La scrittura $\sigma(c_i)$ significa applicare il mapping ad ogni variabile appartenete a $c_i \rightarrow$ si prova a trovare una soluzione facendo la scansione all'interno dell'insieme dei vincoli per vedere se è possibile effettuare un assegnamento.

In maniera più formale, considerando un generico vincolo c_i che è descritto come una relazione r_i applicata ad un insieme di variabili $\overline{x} = (x_1, \dots, x_m)$:

$$c_i = r_i(\overline{x}) \text{ dove } \overline{x} = x_1, \dots, x_m$$

Si avrà dunque:

$$\sigma : V \implies D, \forall c_i \in \overline{C} \sigma(r_i(\overline{x})) \in r_i$$

dove

$$\sigma(r_i(\overline{x})) \in r_i = r_i(\sigma(x_1), \dots, \sigma(x_m))$$

Quando applichiamo il mapping σ , si stanno assegnando dei valori dal Dominio D alle variabili.

La verifica di un vincolo c_i consiste nel controllare che i valori assegnati a (x_1, \dots, x_m) rispettino la relazione r_i .

Dunque l'**immagine** delle variabili attraverso la mappatura deve appartenere alla generica relazione r_i .

Si può notare come il *mapping* stia avvenendo tra due strutture relazionali separate: ciò prende il nome di **Omomorfismo**.

Le strutture interessate sono: I vincoli (visti come struttura relazionale) e il DB .

Constraints Graph

Ad ogni problema di tipo CSP è possibile associare naturalmente un grafo o un iper-grafo.

In particolare si avrà $G(\overline{C}, DB) = (VAR, E)$.

L'insieme dei nodi corrisponde all'insieme delle variabili, mentre l'insieme degli **archi** è così definito: $E = \{\{x_i, x_j\} \subseteq var(c) | c \in \overline{C}\}$.

Dunque, si prendono in considerazione tutte le coppie di variabili che sono all'interno dei vincoli.

Questo prende il nome di **Grafo Primale**.

Il *grafo duale* è invece il grafo in cui si ha un vertice per ogni vincolo. Due vertici sono collegati se hanno variabili in comune.

Il **Grafo** è fondamentale per la risoluzione dei problemi, in quanto, non raramente, li si riesce a semplificare.

Più in generale, in presenza di arità non vincolate, si sfruttano gli *iper-grafi* che permettono di collegare più nodi con più archi.

In generale un iper-grafo è così definito:

$$HG = (\overline{C}, DB) = (VAR, H) \text{ con } H = \{var(c) | c \in \overline{C}\}$$

CSP come un Homomorphism Problem

Dal punto di vista matematico trovare un *omomorfismo* equivale a trovare un CSP.

In questi tipi di problemi si hanno due strutture relazionali: **A** e **B**.

- $A = (U, R_1, \dots, R_k)$
- $B = (V, S_1, \dots, S_K)$

L'obiettivo è verificare se esiste un **mapping** tale che per ogni elemento di **A** è possibile trovare l'immagine ottenuta con l'*omomorfismo* nella struttura **B**.

$$\begin{aligned}
 &h : U \implies V \\
 &\text{tale che } \forall x, \forall i \\
 &x \in R_i \implies h(x) \in S_i
 \end{aligned}
 \tag{1}$$

Standard Search Formula

Si hanno:

- **Stati:** che sarebbero le rappresentazioni delle assegnazioni di valori delle variabili finora effettuate.
- **Stato Iniziale:** L'assegnazione vuota {}.
- **Funzione Successore:** Assegna un valore ad una variabile non ancora assegnata, assicurandosi che tale valore non violi nessun vincolo rispetto alle assegnazioni già effettuate. Si fallisce se non ci sono assegnazioni legali possibili.
- **Test obiettivo:** Si controlla se l'assegnazione corrente completa e soddisfa tutti i vincoli.

Backtracking Search

È l'algoritmo più semplice per la risoluzione dei problemi CSP.

Il ragionamento è semplice: si prende una variabile dall'insieme e gli si assegna un valore, se il valore non viola alcun vincolo si procede, altrimenti si torna indietro di un passaggio e si cambia nuovamente il valore.

```

function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given CONSTRAINTS[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment, csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure

```

La tecnica è *top-down* orientata al Goal.

Il problema principale è che si ha il rischio di espandere l'intero albero di ricerca avendo dunque un costo esponenziale.

Per poter decrementare il costo è possibile applicare delle migliorie.

La più intuitiva è eliminare dalle possibili assegnazioni le associazioni incompatibili ed evitare di fare *assegnamenti* che a priori sono errati.

La seconda miglioria è il **filtering** il cui obiettivo è quello di anticipare una possibile condizione di fallimento.

Filtering: Forward Checking

Il ragionamento è piuttosto semplice, una volta assegnata ad una variabile un valore del proprio dominio, a tutti le altre variabili che hanno una relazione vincolante con questa, si rimuove il valore assegnato.

In questo modo non si possono avere incompatibilità negli assegnamenti.

Il procedimento continua o fino a quando non si trova una soluzione oppure fino a quando un dominio delle variabili si è svuotato.

Se accade la seconda possibilità significa che l'assegnazione effettuata precedentemente è scorretta.

Dunque, il *filtering* non si accorge a priori che una determinata scelta può portare ad errore, ma con il procedere dell'algoritmo ciò avverrà sicuramente.

Per controllare questo aspetto si fa riferimento alla **arc consistency**.

Arc Consistency

Togliere qualcosa da una variabile va irrimediabilmente ad intaccare i vincoli che sono contigui a quest'ultimo.

Con l'arc consistency, quando un valore viene eliminato da un vincolo, si controllano anche gli altri vincoli connessi eliminando anche in quel caso quei valori. Ciò porta ad una catena di eliminazioni che propagano le scelte effettuate.

Se una tupla si **svuota totalmente** significa che **non esiste soluzione**.



Un arco $X \implies Y$ si definisce consistente se e soltanto se $\forall x_i \in X$ esiste un qualche valore $y \in Y$ che può essere assegnato senza violare un *vincolo*.

In altro modo si può definire come:



una variabile X_i è **arc-consistent** rispetto ad un'altra variabile X_j se per ogni valore nel dominio D_i c'è qualche valore nel dominio D_j che soddisfa il vincolo binario dell'arco.

Algoritmo Migliorato

```
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
local variables queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
   $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
  if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
    for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
      add  $(X_k, X_i)$  to queue

function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  removed  $\leftarrow$  false
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x, y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow$  true
  return removed
```

Questo tipo di approccio è applicabile nel caso di **istanze binarie**.

Come funziona l'Algoritmo?

Inizialmente si ha una *queue* che contiene tutti gli archi del problema.

Finchè la queue non è vuota si estrae la prima coppia di archi X_i, X_j e si chiama la funzione *remove-incosistent-values*

La funzione verifica la consistenza: per fare ciò prende in considerazione il dominio di X_i e verifica se per ogni elemento del dominio esiste almeno un elemento y nel dominio di X_j che soddisfa il vincolo.

Dunque verifica se la coppia (x, y) che soddisfa il vincolo esiste.

Se non esiste almeno un valore che soddisfa questa relazione allora il valore di x viene rimosso dal dominio di X_i .

Costo dell'algoritmo

Considerando:

- d : è la **dimensione del dominio**.
- n : è il **numero di variabili**.
- m : è il **numero di vincoli**.

Nel caso peggiore, per ogni arco presente all'interno della lista si rimuovono tutti i possibili valori: si effettuano d iterazioni sul dominio di X_i che portano ad altrettante d iterazioni sul dominio X_j .

Il costo per effettuare questo passaggio di rimozioni è pari a $O(d^2)$.

Il tutto deve essere calcolato per tutti gli m *vincoli* presenti.

Il costo è pari a $O(d^3 * m)$.

Il calcolo così effettuato è però molto impreciso e rappresenta un **upperbound**, in quanto stiamo dando per scontato che ogni variabile abbia una dimensione del dominio pari a d .

Per essere più precisi è possibile indicare con $|r_{max}|$ la dimensione della relazione più grande.

In questo modo il costo può esser scritto come: $|r_{max}| \times |r_{max}| \times m$

Utilizzando questa relazione non solo il costo è più preciso, ma si può applicare anche nei casi di relazioni NON binarie.

- **Possibili esiti dell'arc-consistency:**

- **Si ha una soluzione:** se tutti i vincoli sono soddisfatti e i domini delle variabili non sono vuoti, potrebbe esserci una soluzione, ma non è garantito.
- **Si hanno più soluzioni:** se dopo il filtraggio rimangono valori multipli nei domini, il problema può avere più soluzioni.
- **Non si hanno soluzioni:** se uno dei domini si svuota, significa che non esiste alcuna soluzione compatibile con i vincoli.

- **Rilevamento di inconsistenze:**

- Se durante l'applicazione dell'arc-consistency (ad esempio al passo iniziale, prima che siano fissati valori) un dominio si svuota, si è certi che il problema non ha soluzione.
- In questo caso, l'algoritmo può terminare immediatamente, evitando ulteriori calcoli.

- **Arc-consistency nel backtracking:**

- Quando l'arc-consistency è usata all'interno di un algoritmo di backtracking:
 - Se un dominio si svuota durante l'elaborazione, significa che la strada scelta è sbagliata e bisogna tornare indietro (backtrack) per cambiare decisione.
 - Se i domini vengono ridotti ma non svuotati, il problema viene semplificato senza ridurre le possibilità di trovare una soluzione.

- **Proprietà e vantaggi dell'arc-consistency:**

- L'arc-consistency forza ricorsivamente i vincoli tra le variabili, cercando di individuare inconsistenze.
- È una buona pratica applicarla preliminarmente, poiché se uno dei domini si svuota in questa fase, si può concludere che il problema non ha soluzione.
- Tuttavia, anche se i domini rimangono non vuoti, l'arc-consistency non garantisce che ci sia effettivamente una soluzione: ciò dipende dalle proprietà specifiche del problema.

- **Miglioramenti possibili:**

- Se un dominio viene ridotto a un solo valore, si può procedere con l'assegnazione di quel valore.
- In fase di scelta, si possono preferire variabili o valori che facilitano la risoluzione del problema (ad esempio, applicando euristiche come la "variabile più vincolata").

Ordering

Un altro modo per ottimizzare la ricerca di soluzioni dei problemi CSP è applicare l'**ordering**.

Si dà per scontato il fatto che il *filtering* sia già stato applicato e dunque i valori rimasti siano assegnabili.

Ma come si può scegliere la variabili da cui iniziare per dare gli assegnamenti?

Minimum Remaining Values



Si prende in considerazione la variabile che ha meno *valori* rimasti nel *dominio*.

Il motivo per cui si effettua questo ragionamento è dettato dal **Fail-Fast**, cioè si assegnano i valori alla variabile che è maggiormente vincolata, in questo modo se si trova una combinazione senza andare in fallimento allora è

molto più probabile che, per *variabili* che hanno più scelta, si trovi un valore da assegnare.

Least Constraining Value

Una volta identificata la variabile, quale *valore* si deve assegnare per primo?

Una scelta ponderata è quella di assegnare il valore che risulta essere **meno vincolante** e che dunque potrebbe garantire con maggiore probabilità di trovare una soluzione.

Si sceglie un valore che lascia più aperte le vie future.

Nonostante queste tecniche il *worst-case* rimane comunque esponenziale.

Iterative Improvement



Si parte da una scelta casuale (o pseudocasuale) e solo successivamente si prova ad aggiustare la soluzione rendendola *ammissibile*.

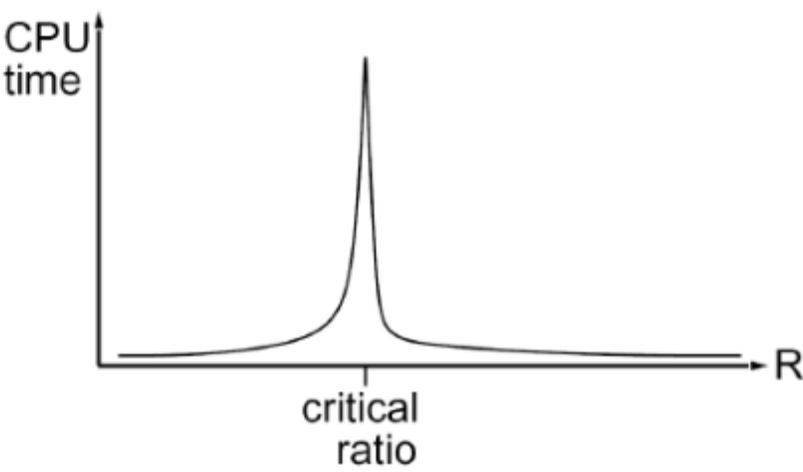
Performance

In generale si può ricadere in due tipologie di problemi:

- Problemi che presentano **pochi vincoli** e sono facilmente risolvibili con un *backtracking*.
- Problemi che sono **over-constrained** e dunque è impossibile trovare soluzioni.

In generale esiste un rapporto tra *numero di costanti* e *numero di vincoli* che definisce un **punto critico**.

$$R = \frac{\text{numero di vincoli}}{\text{numero di variabili}} \tag{2}$$



Struttura

L'obiettivo adesso è quello di sfruttare la struttura di un problema a nostro favore.

Nei problemi CSP non è raro avere strutture che suggeriscono esse stesse la soluzione.

Le caratteristiche riconosciute che possono essere sfruttate prendono il nome di **caratteristiche strutturali**.

Dunque, si possono avere sia problemi che è possibile risolvere per intero con qualche accorgimento nella struttura, sia problemi che invece richiedono la divisione in *sottoproblemi* per poterne trovare una soluzione.

Supponendo di avere un problema con:

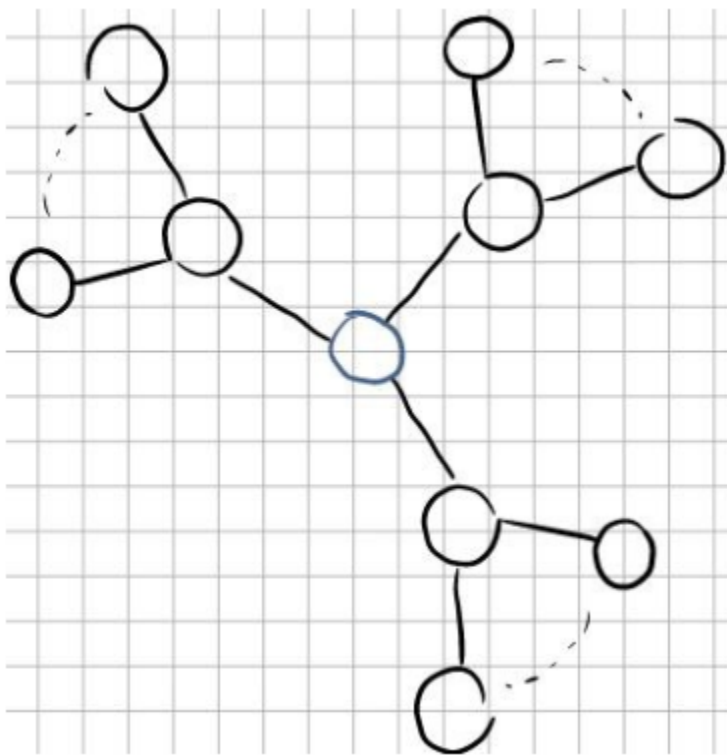
- ***n* variabili**.
- Diviso in sottoproblemi con ***c* variabili**.
- ***d*** come insieme di valori possibili.

Il costo per la risoluzione di uno dei singoli sottoproblemi, nel caso peggiore è $O(d^c)$.

Se si hanno $\frac{n}{c}$ sottoproblemi, allora il costo totale sarà $O(\frac{n}{c} * d^c)$.

Ovviamente può capitare che non è semplice dividere in sottoproblemi.

In quei casi spesso è possibile **“eliminare”** una variabile (per farlo è sufficiente dare un valore costante) che può rendere aciclico un grafo.



Se il valore assegnato precedentemente non porta ad una soluzione si può modificare il valore che prima era stato assegnato al nodo da eliminare e provare nuovamente l'algoritmo.

Non ha senso però eliminare troppe variabili perchè poi si potrebbe avere il problema opposto dato dal dover risolvere tutti i sottoproblemi modificando il valore di questa costante.

Ciò ha un peso pari a d^k dove k è proprio il numero di variabili da cancellare.

Tree Structured CSP

A seguire si analizzeranno i problemi CSP che presentano come struttura un albero e dunque, PER DEFINIZIONE, sono aciclici.



Ricorda, in un albero si ha

$$archi = nodi - 1 \text{ che asintoticamente diventa } archi = nodi \tag{3}$$

Se definiamo, come prima, d il numero di elementi in un dominio, avremo che il costo della risoluzione è pari a $O(n * d^2)$

In generale l'obiettivo è non avere cicli all'interno delle strutture perchè permettono una risoluzione molto più semplice e veloce.

CSP con istanze acicliche: Costo

Come detto in precedenza, l'*arc consistency* ha un costo pari a $|r_{max}|^2 * m$, dunque un costo cubico.

In realtà, il costo reale è pari a:

$$||A|| * ||B|| * \log ||B|| \tag{4}$$

Dove:

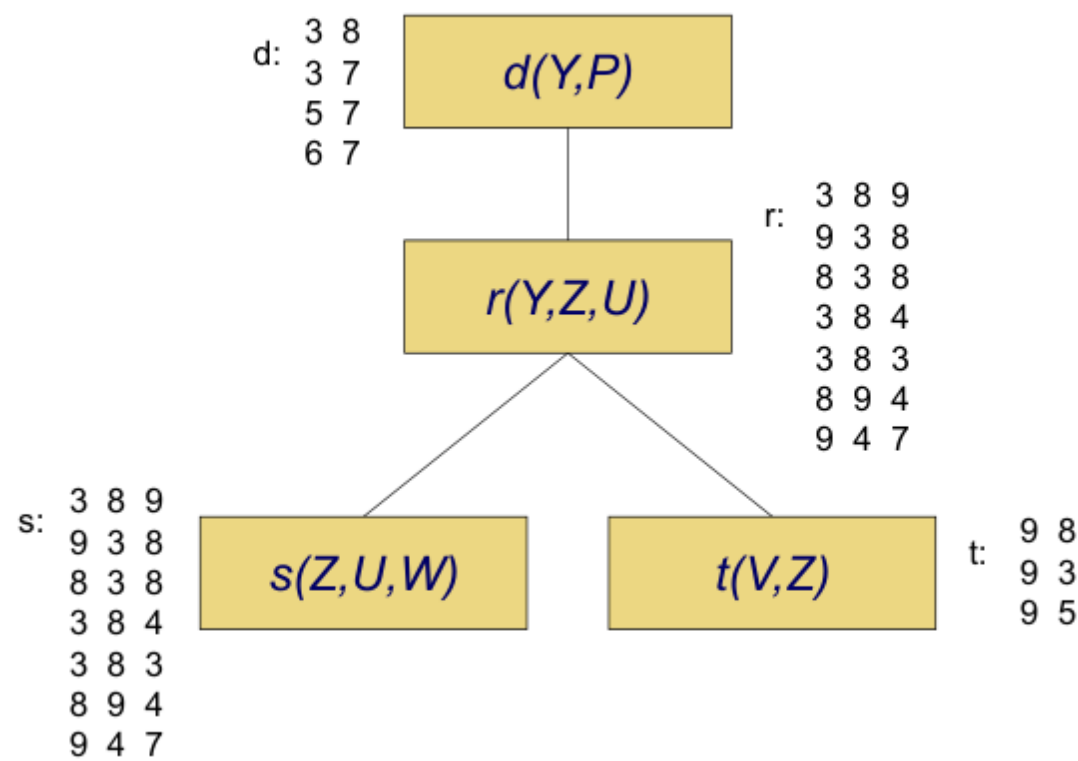
- $||A||$ è la dimensione dei vincoli ed è dunque pari ad m .
- $||B||$ è la dimensione del database e dunque si può associare a $|r_{max}|^2$.

Questo tipo di problemi appartiene alla classe *LOGCFL* dunque è parallelizzabile.

L'*arc consistency* può essere applicata anche in strutture cicliche, però, sapendo a priori che la struttura è aciclica, si può sfruttare questa proprietà.

Yannankakis's Alghoritm

Nel caso in cui si sappia a priori che la struttura è *aciclica* è possibile applicare un *ordinamento topologico bottom-up* e procedere dalle foglie verso la radice.

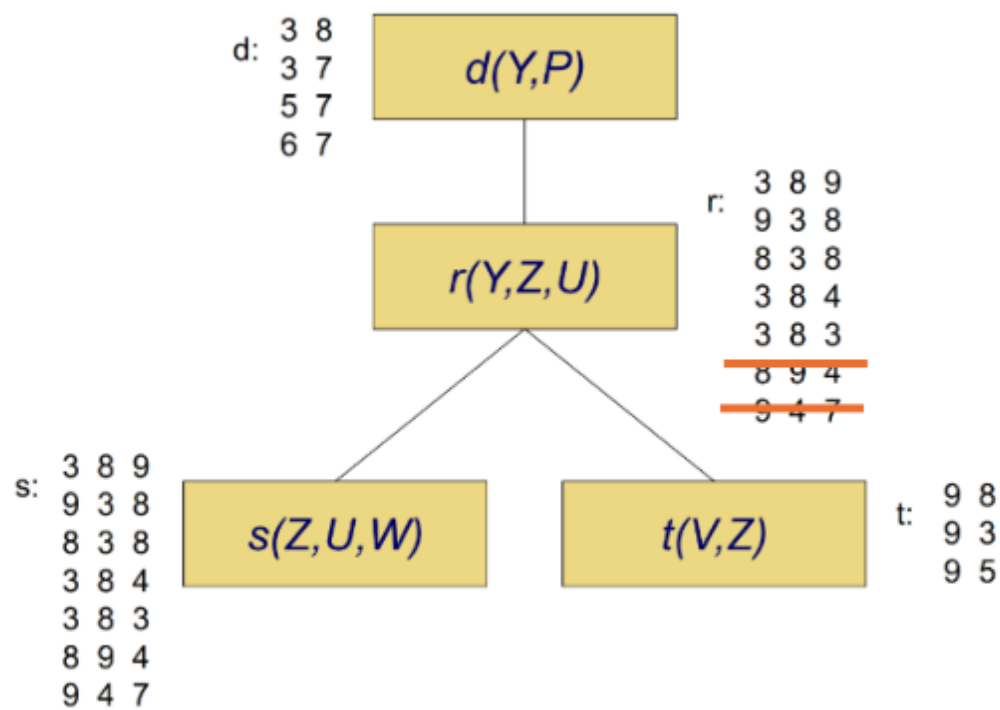


Supponiamo di avere questo problema dove i vincoli sono i nodi.

NOTA: Ciò che sta accanto ai nodi sono le relazioni.

Come anticipato, si parte dai nodi fogli e si propaga l'informazione ai nodi superiori.

Dunque si fanno prima dei confronti incrociati tra la relazione s e la relazione r , poi dei controlli incrociati tra t e r .

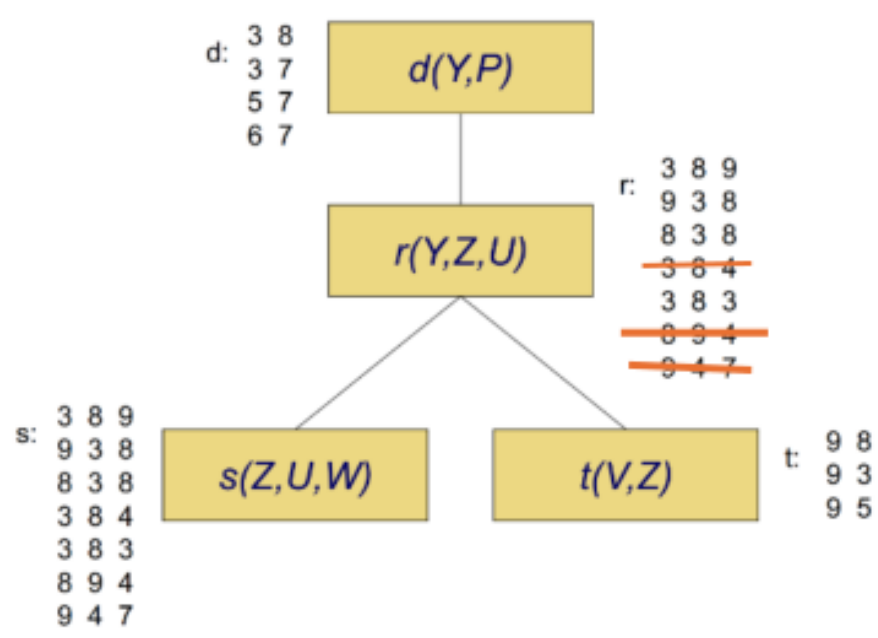


Le relazioni r ed t condividono entrambe il valore Z .

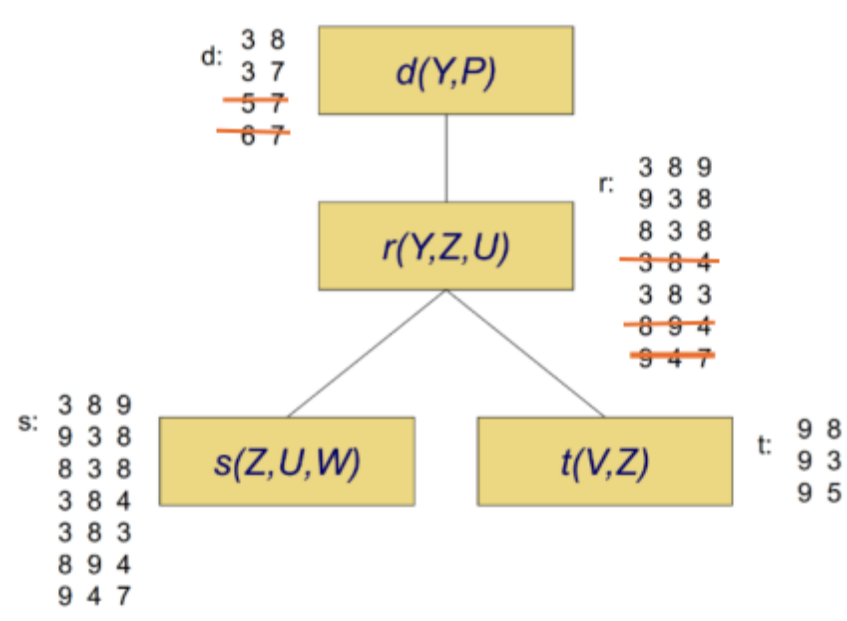
in t il valore Z può assumere i seguenti valori $Z = \{8, 3, 5\}$

in r il valore Z può assumere i seguenti valori $Z = \{8, 3, 9, 4\}$

Dato che i valori 9 e 4 non sono compatibili con t possono essere eliminati.



Lo stesso ragionamento è applicato con S.
 In questo caso si rimuove il valore 4 per U.
 Questo ragionamento è poi riportato a salire con la propagazione delle informazioni.



Dato che l'analisi è fatta dal basso verso l'alto, può capitare che una rimozione poi vada a modificare qualche parametro di altri vincoli, di conseguenza al termine si fa una passata *top-down* e si correggono eventuali problemi di consistenza.

Il costo dell'algoritmo è pari a:

$$|R_{max}| * \log |R_{max}| * (m - 1) \tag{5}$$

Dove:

- $m - 1$ è il numero di archi.
- $|R_{max}| * \log |R_{max}|$ è il costo della *join* in una tabella ordinata.

Iper-Grafo e Aciclicità

La nozione di ciclicità negli *iper-grafi* è differente e ne esistono di diverso tipo.
 La più utilizzata è l'*alpha* ciclicità.

Un **Iper-grafo** è **alpha-ciclico** se ammette un *join tree*.

Un **Join Tree** è un albero in cui ogni nodo è un *iper arco* ed è *aciclico* se i nodi costituiscono un albero.

La proprietà fondamentale di un **Join Tree** è la seguente:

Se una variabile compare in due vertici, deve comparire in tutto il percorso che li collega.

Questa prende il nome di proprietà di **connessione**.

La proprietà di connessione in pratica afferma che una variabile deve comparire al più nei suoi vicini, se ciò non accadesse implicherebbe la presenza di un *ciclo*.

La costruzione di un *join tree* si può fare in tempo *logaritmico*.

Definizione Join-Tree



Sia $HG = (V, H)$ un ipergrafo.

Un Albero $T = (H, E)$, i cui vertici corrispondono agli archi dell'ipergrafo, è un *join-tree* se:

$$\forall p, q \in V, p \cap q \neq \emptyset \implies \forall x \in p \cap q \quad (6)$$

$$x \text{ occorre nel path che collega } p \text{ e } q \text{ in } T \quad (7)$$

Global Consistency



Sia $c_i(\bar{x}_i)$ un vincolo.

$$\forall t = c_i(v_1, \dots, v_a) \in DB \text{ esiste una soluzione } \sigma : V \implies D : \sigma(c_i(\bar{x}_i)) = t$$

NOTA: a sarebbe l'arità della relazione.

La Global consistency afferma che ogni tupla rimasta in ogni vincolo è parte di una soluzione.

In particolare si può affermare che se $HG(I)$, dove I è il problema CSP, è aciclico allora:

$$Aciclico \implies LocalConsistency = GlobalConsistency \quad (8)$$

Garantire questo aspetto è importantissimo in quanto ci permette di affermare sempre che si ha una soluzione è che a prescindere dalla scelta presa in considerazione, sicuramente giungeremo ad una soluzione.

Inoltre l'affermazione vale pure in verso opposto:

$$LocalConsistency = GlobalConsistency \implies Aciclico \quad (9)$$

Local Consistency



$$\forall c_i(x_i), c_j(x_j) \in \overline{C}$$

$$\forall t = c_i(v_i, \dots, v_a) \in DB$$

$$\text{Esiste un mapping } \sigma : V \implies D : \sigma(c_i(x_i)) = t \wedge \sigma(c_j(x_j)) \in DB$$

Ciò afferma che per ogni tupla presente in una relazione che soddisfa il vincolo $c_i(x_i)$, esiste un'assegnazione che permette di garantire contemporaneamente le seguenti due proprietà:

- $\sigma(c_i(x_i)) = t$ e dunque che l'assegnazione corrente è coerente con il vincolo
- $\sigma(c_j(x_j)) \in DB$ e dunque che la precedente assegnazione ai valori delle variabili del vincolo c_j produce una tupla che appartiene ancora al DB e ciò soddisfa il vincolo c_j .

L'obiettivo adesso è capire come rendere *aciclico* un problema CSP.

Una prima tecnica è il **Cut-Set**.

Cut-Set

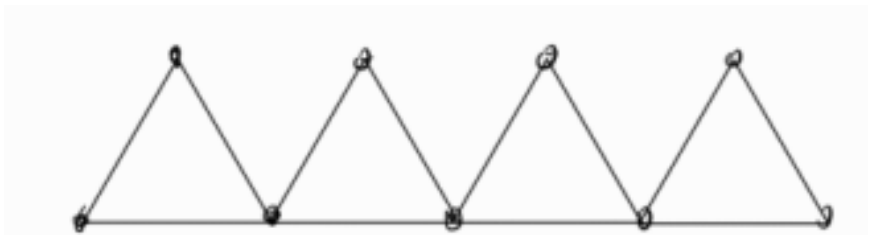
È il ragionamento che è stato effettuato precedentemente, cioè la rimozione di una variabile dal problema per poterlo semplificare.

È possibile identificare la **width** del problema che rappresenta la ciclicità di quest'ultimo. Ad esempio un problema che ha *width* 2 significa che bisogna eliminare 2 variabili per renderlo aciclico.

In particolare la **cut-set-width** è il numero minimo di variabili da *cancellare* per ottenere una struttura aciclica.

Però non sempre è conveniente applicare questa tecnica.

Supponiamo di avere la seguente struttura:



Si nota immediatamente come per poter eliminare i cicli si devono togliere molti nodi (in particolare stiamo facendo riferimento a quelli in basso)

Dunque non sono rari i casi in cui si debbano togliere un numero lineare di nodi.

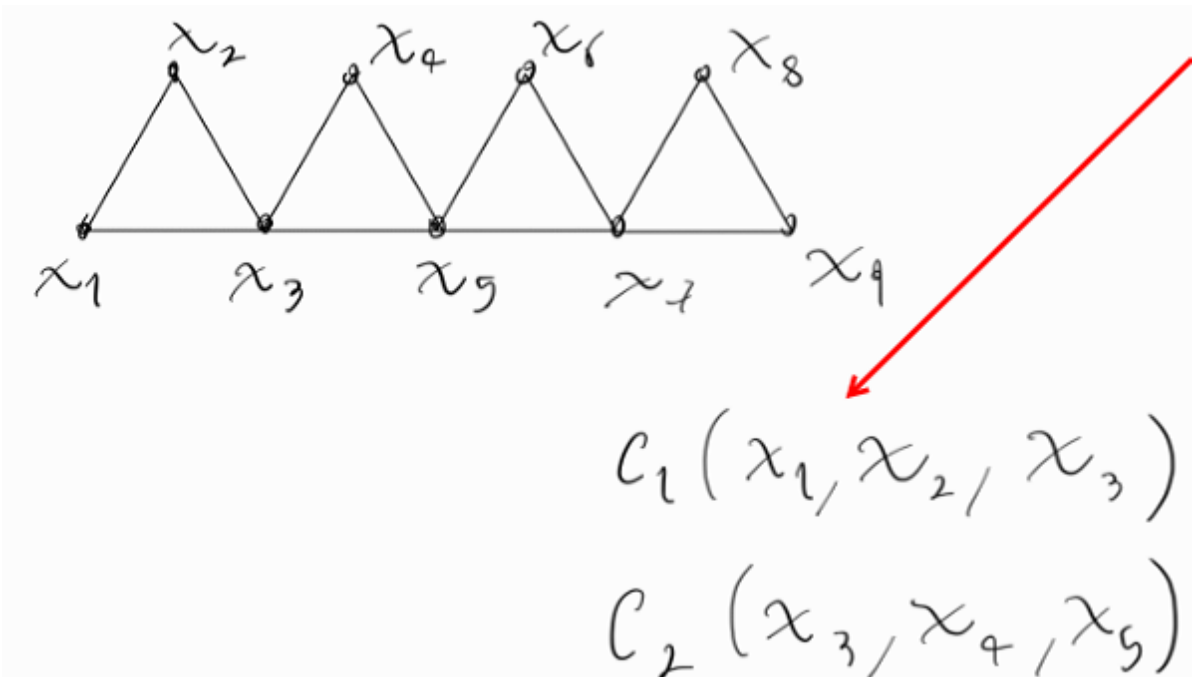
Il costo di questo algoritmo è il seguente: supponiamo che sia necessario eliminare k variabili per avere una struttura *aciclica*, è necessario testare tutti i valori del dominio di tutte le k variabili prima di poter trovare una soluzione (caso peggiore), si ha un costo pari a d^k .

Perciò, quando ci sono tante variabili in gioco da eliminare, non conviene effettuare il *cutset*.

Notiamo come è possibile generare una **catena di sottoproblemi** in cui i nodi che sono “condivisi” possono essere risolti tramite una *join* post risoluzione del singolo sottoproblema.

Dunque, il costo totale del CutSet è pari a:

$$d^k * (n - c)d^2$$

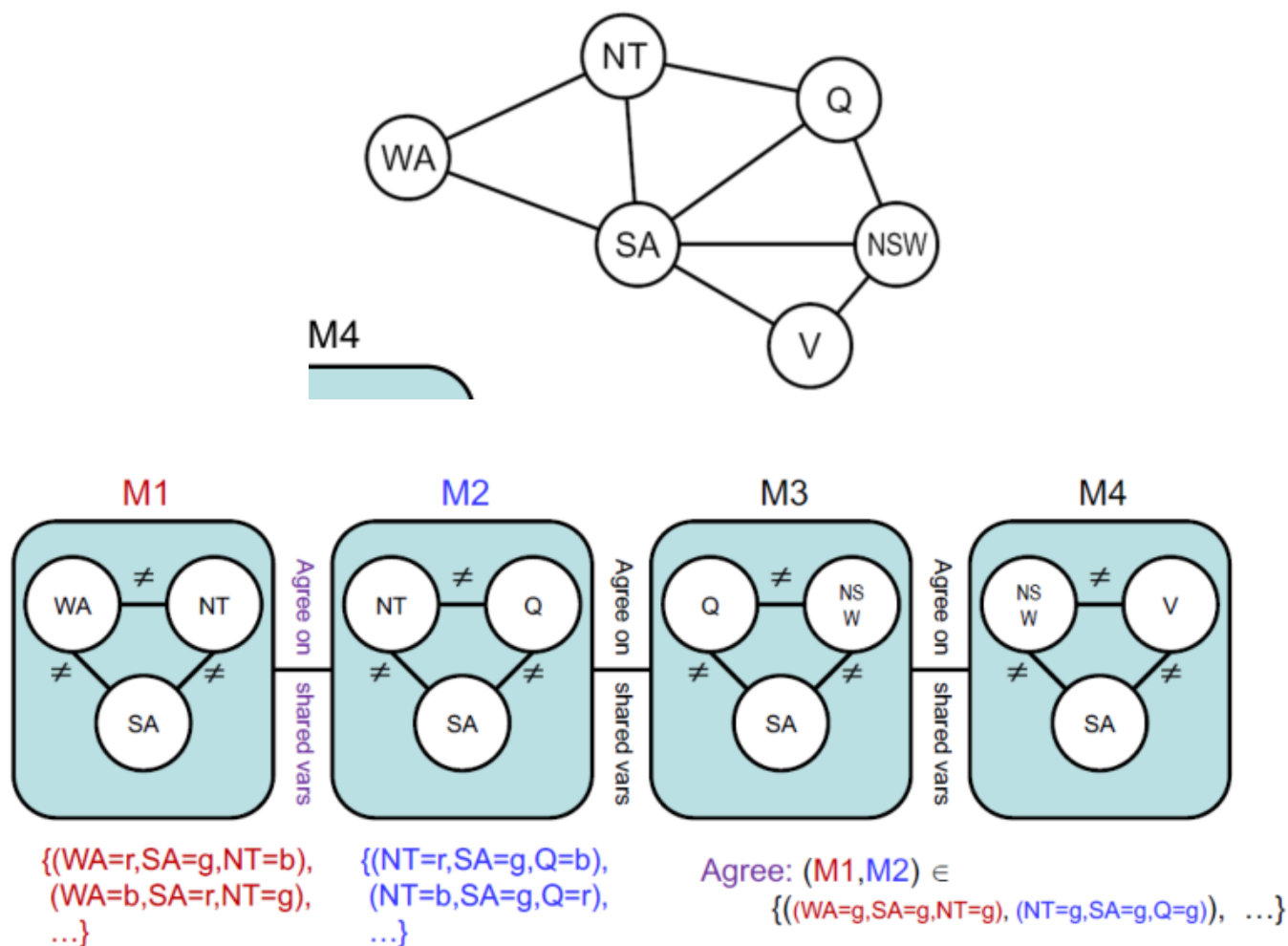


Si deve tenere in considerazione che non è detto che se si trova una soluzione *per ogni sottoproblema* allora è assicurata la **soluzione totale**, molto dipende anche dai vincoli delle variabili condivise nei sottoproblemi.

→ L'obiettivo è andare a trovare i sottoproblemi corretti che rendano la struttura aciclica in modo tale da poter applicare gli algoritmi analizzati in precedenza come *yannakakis* e *arc-consistency*.

La tecnica adesso analizzata prende il nome di **Tree Decomposition**.

Tree Decomposition



Ogni quadratino rappresenta un *sottoproblema* e si possono individuare come *iper-archi* del *join tree*.

Endomorfismi

Come detto in precedenza, i problemi CSP possono essere visualizzati come omomorfismi tra *strutture relazionali*.

Dunque si devono mappare *tuple di variabili* in *tuple di costanti*.

Ma se invece di mappare *variabili* → *costanti* si mappassero le *variabili* nelle *variabili*?

L'idea è di vedere se è possibile trasformare il *problema grosso* in un *problema più piccolo*, in questo modo se si risolvesse il problema più piccolo si risolverebbe anche il problema più grande.

Si consideri l'esistenza di una *struttura* A che si mappa con un omomorfismo in una *sotto-struttura* A' . (In questo modo le variabili di partenza vanno in un suo sotto-insieme)

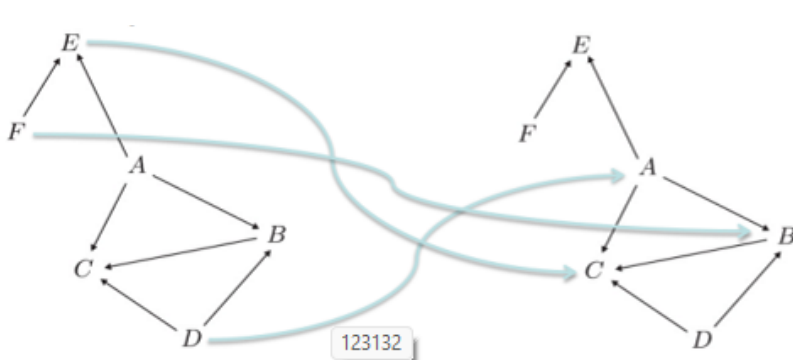
$$A \xrightarrow{h} A'$$

Il dettaglio importante è che $A = A'$, cioè sono equivalenti, perciò risolvere A' equivale a risolvere A .

Se poi si riesce a trovare un secondo omomorfismo h' che permette di passare ad un altro problema più semplice B , allora risolvendo B si risolve anche A .

$$A \xrightarrow{h} A' \xrightarrow{h'} B$$

Esempio Endomorfismo



Problema di Partenza

F	E
A	E
A	C
A	B
B	C
D	C
D	B

Si può notare che:

- $A \rightarrow E$
- $F \rightarrow E$

Quindi è necessario trovare un nodo che viene puntato da 2 altri nodi. Ciò lo ritroviamo in C.

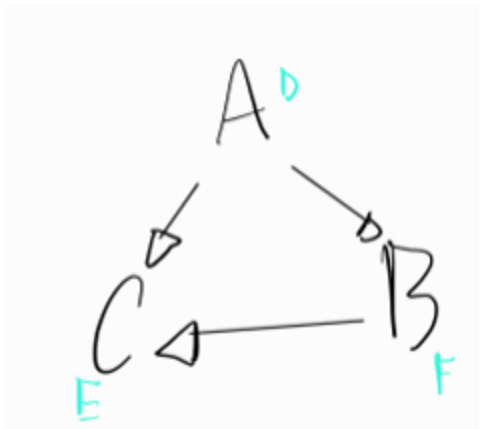
Dunque è possibile mappare:

- $h(C) = E$
- $h(B) = F$

Stesso ragionamento vale per D, che punta a due nodi differenti, di conseguenza può essere mappato su A.

- $h(A) = D$

Con queste scelte si ha il seguente problema semplificato:



In questo modo si rispettano comunque tutte le relazioni.

- F continua a puntare ad E, così come A.
- D continua a puntare sia a C che a B.

CORE

Esistono diversi tipi di *endomorfismi*, ma i più importanti per i problemi CSP prendono il nome di **CORE** in quanto ci garantiscono la massima *semplificazione* del problema.

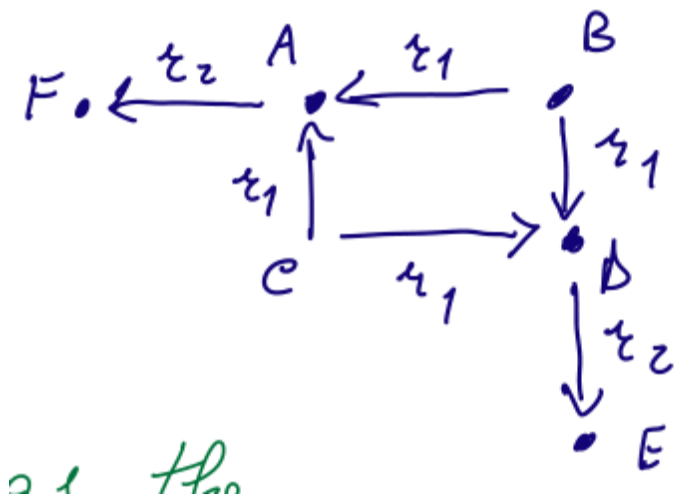


I **CORE** sono le più piccole strutture *endomorfiche* ottenibili da un problema CSP.

Rappresentano l'*endomorfismo minimale*.

Un aspetto peculiare dei CORE è che sono *isomorfi* dunque possono essere scambiati tra di loro senza apportare nessuna particolare modifica alla soluzione del problema.

Esempio: CORE



A e D presentano un comportamento simile, entrambi sono mappati da B e C dunque è possibile mappare D in A.

- $h(A) = D$

Inoltre ci interessa mantenere il puntamento di D verso E e di A verso F.

Per mantenerlo si può mappare E in F.

- $h(F) = E$

Inoltre, anche B e C hanno un comportamento simile (puntano entrambe verso A) di conseguenza si può mappare C in B.

- $h(B) = C$

In questo modo il problema di partenza si risolve in:



Esiste un teorema interessante che afferma che:

$$\begin{array}{c} \text{Se il core di una struttura è aciclico} \\ localConsistency \implies GlobalConsistency \end{array}$$

Questa prende il nome di **Semantic Aciclicity**.

Definizione Tree Decomposition



Sia $G = (V, E)$ un grafo.

Sia $T = (N, A)$ un albero.

Sia $\chi : N \implies 2^V$ una funzione che mappa i vertici dell'albero in insiemi di variabili del problema originale.

Si dice che (T, χ) è una **tree decomposition** di G se:

$$\begin{array}{c} \forall e \in E, \text{ esiste un vertice } v \in V, \text{ con } e \in \chi(v) \text{ tale che} \\ \forall v_1, v_2 \in N, \forall x \in \chi(v_1) \cap \chi(v_2) \\ x \text{ appare in ogni etichetta } \chi \text{ nel path che collega } v_1 \wedge v_2 \text{ in } T \end{array}$$

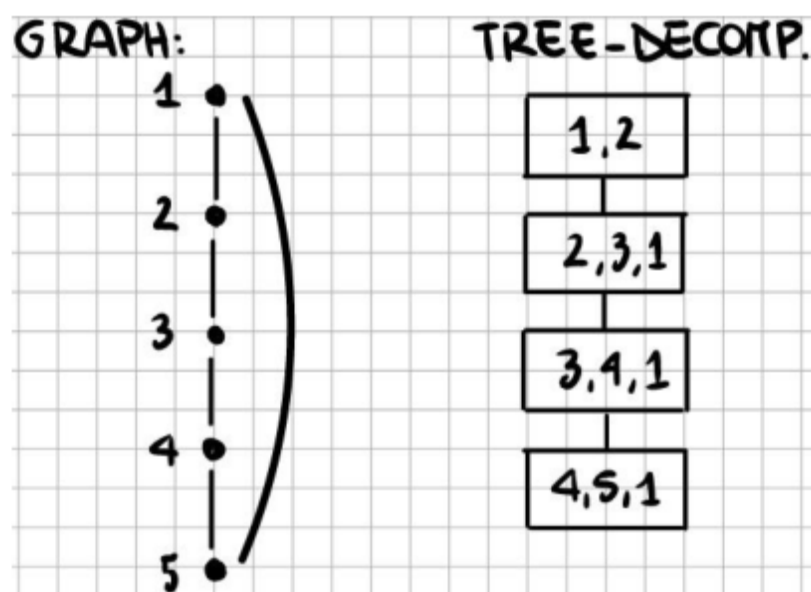
In altre parole:

1. Ogni *vincolo* del problema originario si trova da qualche parte nell'albero.
2. Se una variabile compare in due etichette distinte, deve apparire anche nel *path* che le collega (Se io ho il nodo $X_1 \wedge X_3$ e appare la variabile C, affinché sia vera questa cosa la variabile deve apparire anche in X_2 altrimenti non sarebbero collegate)

La domanda che si può porre è: Ma esiste sempre una *tree decomposition*?

La risposta è sì, perchè si potrebbe mappare tutto in una sola variabile e comunque verrebbe rispettata la proprietà, ma sarebbe inutile ai fini della risoluzione del problema.

Più in generale, ai fini della risoluzione si può affermare che i problemi con *struttura triangolare* e i *graph-ladder* sono sempre risolvibili, il problema è con i *cicli semplici*.



Per eliminare il ciclo è stato sufficiente “portare con sè” una variabile che garantisce che entrambe le proprietà venissero rispettate.

Tree Width

La Tree width è la *cardinalità minima* tra tutte le possibili decomposizioni.

$$\max_{v \in N} \{|\chi(v)| - 1\}$$

La si può interpretare come una misura di *ciclicità*.



La **Tree Width** di G è la più piccola *width* tra tutte le possibili *tree-decomposition* di G .

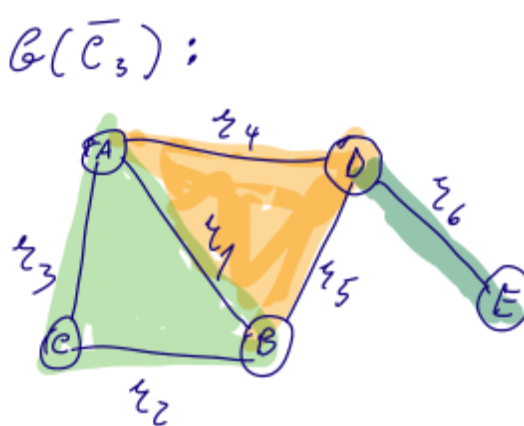
Si può affermare che un **grafo** è **aciclico** se la sua *treewidth* è pari ad 1.

Quanto costa calcolare la Tree Decomposition migliore?

In generale interessa sapere quanto costerebbe avere $tw(G) \leq k$.

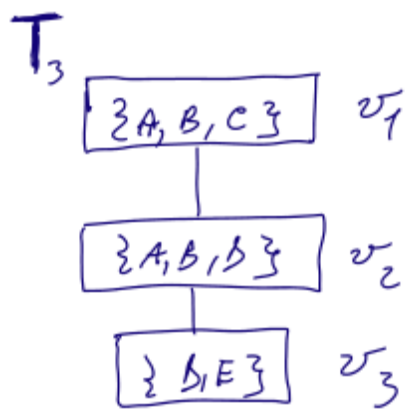
Se k è fisso il problema è *Lineare*.

Se k è variabile il problema è NP-Hard.

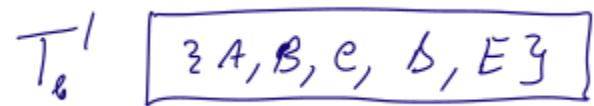


Esempio Di Partenza

Si possono generare diverse Tree Decomposition



Caso a



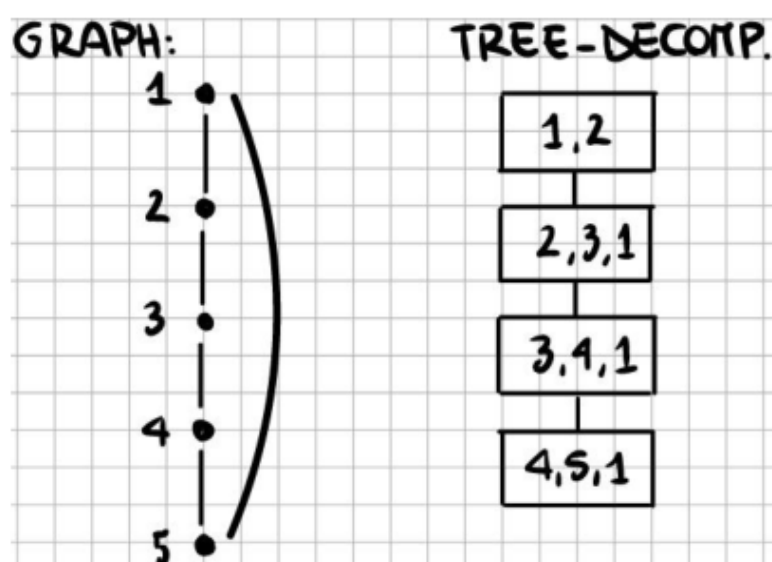
Caso b

Nel primo caso notiamo con la width sia pari a 2, nel secondo è pari a 4.

Nonostante siano entrambe tree decomposition una è migliore rispetto all'altra.

La prima è pari a 2 in quanto il grafo è ciclico.

Tornando all'esempio di Prima



Per mantenere la proprietà di connessione abbiamo portato l'1 tra i vari nodi.

Il problema è che non esiste alcuna relazione tra $\{3, 4, 1\}$ e $\{4, 5, 1\}$.

Quindi il sottoproblema come si deve risolvere?

Si deve effettuare il prodotto cartesiano tra il vincolo e tutti i valori possibili che può assumere $1r(3, 4) \times dom(1)$.

Quindi risolvere questo tipo di problemi è *esponenziale* nel numero di variabili che compaiono nella *tree width*.

Dato che la struttura diventa **ACICLICA** è possibile applicare gli algoritmi efficienti visti in precedenza.

In generale, per risolvere questi problemi si fissa un determinato valore di k e si verifica se esiste una **tree decomposition valida**.

Si analizzano i sottoproblemi associati e si risolvono con uno qualsiasi degli argomenti analizzati.

Dunque, i passi da seguire sono:

1. Calcolare, se esiste, una **Tree Decomposition** del **grafo G** con $width \leq k$.
2. Usando la **Tree Decomposition** si genera un'istanza I , aciclica, ed *equivalente* a I .

Il primo punto ha un costo che è **lineare in G**.

Il secondo punto è *polinomiale*, ma risulta essere *esponenziale* in k .

Perciò, il costo totale per risolvere il problema è pari a:

$$(d^{TW(G)+1} * \log d^{TW(G)+1}) * Vars(I)$$

Il parametro tra parentesi rappresenta la *merge* fra le relazioni.

In particolare le relazioni possono essere al massimo grandi tanto quanto $d^{TW(G)+1}$.

Hyper Tree Decomposition

Per i CSP si può fare di meglio ed usare l'Hyper Tree Decomposition.

L'idea è di considerare i *vincoli* anzichè le *variabili*.

La **ciclicità** è definita dal numero di *iperarchi* che coprono quelle variabili.



La Hyper Tree Decomposition è definita da una tripla:

$$\langle T, \chi, \lambda \rangle$$

I valori di T, χ possono essere visualizzati come una *tree decomposition*.

λ indica i vincoli che conviene utilizzare per coprire delle variabili nella χ .

cioè $\forall v \in T, \chi(p) \subseteq \cup_{h \in \lambda(v)} h$

Teorema per Problemi con Arità Limitata



Sia S una classe *ricorsivamente enumerabile* di strutture relazionali.

Definiamo $CSP(S, _)$ il seguente problema:

Data una struttura $\bar{c} \in S$ e una struttura DB , con *stessa signature*, decidere se

$I = (\bar{c}, DB)$ è un'istanza del CSP.

- $_$ Significa che non ci sono limitazioni sul database DB . Prende il nome di *problema uniforme*.

Identifichiamo S_k tutte le strutture con $treeWidth \leq k$.

Sappiamo che $CSP(\bar{S}_k, _)$ appartiene a PTIME.

In particolare se si indica con S'_k tutte le strutture relazionali aventi un **CORE** di $tw \leq k$, si sa che $CSP(\bar{S}'_k, _)$ appartiene a PTIME.

Però non è possibile calcolare un core in tempo *PTIME* dunque come è possibile questa cosa?

Teorema [Grohe Etal]



Sia S una classe R.E. di strutture relazionali di arità limitata.

Allora $CSP(S, _)$ è in PTIME se e solo se i **core** della struttura in S hanno una **tree width limitata**.

Senza dimostrarlo cerchiamo di capire come possa valere.

Consideriamo un'istanza $I \in CSP(S, _)$.

Sia G il *grafo primale* di I .

Allora sappiamo che esiste un core di G , diciamo che sia G' , la cui **tree width** è minore di una costante k .

Consideriamo $k + 1$ variabili, supponendo di avere un numero di variabili pari ad n .

Si prendono in considerazione tutte le combinazioni possibili che sono pari a d^{k+1} .

Dunque per ogni vincolo ci saranno d^{k+1} valori.

Si considerano tutti i vincoli con $k + 1$ variabili e successivamente si inseriscono tutti i vincoli di partenza.

A questo punto si applica la *local consistency*, ciò che resta fa parte della soluzione.

Il problema dove sta? Che non esiste un algoritmo semplice che ci permette di verificare se l'istanza appartiene o meno ad S (cioè se ha un core con **tree width** limitata).

Dunque se si sa a priori che le istanze appartengono ad S allora la soluzione trovata è corretta.

Più in generale se l'algoritmo risponde di NO, allora sicuramente non c'è la soluzione.

Se risponde di SI però non si ha la certezza.

Planning

Un plan lo si può idealizzare come una sequenza di *azioni utili* per raggiungere un certo obiettivo.

Un plan è costituito da:

- **Descrizione degli stati iniziali del mondo.**
- **Obiettivo da raggiungere.**
- **Insieme di possibili azioni.**

Tra i planning più noti si hanno:

1. **State Space Planning:** ogni nodo rappresenta uno stato del mondo. Quindi il plan non è altro che un cammino lungo lo spazio degli stati che congiunge uno *stato di partenza* con uno *stato di goal*.
2. **Plan Space Planning:** ogni nodo è un insieme di operatori parzialmente istanziati con qualche vincolo in aggiunta.

Esistono diverse definizioni di Plan



- Uno schema, programma, o un metodo elaborato in anticipo per il raggiungimento di un obiettivo: un piano di attacco;
- Un progetto o una linea d'azione proposta o provvisoria: non aveva piani per la serata.
- Una disposizione sistematica di elementi o parti importanti; una configurazione o uno schema: una pianta dei posti a sedere; il piano di una storia;
- Un disegno o un diagramma in scala che mostra la struttura o la disposizione di qualcosa;
- Un programma o una polizza che stipula un servizio o un beneficio: un piano pensionistico.

Formalismo di un problema di Planning

Il primo passaggio da eseguire una volta individuato il problema è astrarre ed individuare le caratteristiche generali andando a rappresentare solo quello che il planner necessita.



Un **Sistema di Transizione** è una quadrupla:

$$\Sigma = (A, S, E, \gamma)$$

dove:

- S identifica l'insieme di stati possibili in cui il sistema può trovarsi.
- A è l'insieme finito di azioni che l'attore può eseguire.
- E è l'insieme di eventi esogeni, cioè gli eventi che non sono sotto il controllo dell'agente.
- γ è la funzione di transizione di stato:
 - Restituisce i possibili stati o il prossimo stato a seguito di un'azione.
 - $\gamma : S \times (A \cup E) \implies S$ oppure $S \times (A \cup E) \implies 2^S$

NOTA: Uno stato S è un insieme di atomi ground.

Modello concettuale di Attore

Il controller esegue le azioni sulla base del plan che gli viene passato.

A questo punto ci possono essere due scenari:

1. Il controller esegue il piano (Caso **Statico**)
2. Il controller esegue i passi del piano però si basa sui feedback che riceve (Caso **Dinamico**)

Nel **classical planning** siamo di fronte ad un contesto *statico*.



La differenza con uno *schedule* è questo riordina eventi che devono essere eseguiti totalmente. Invece il *planner* decide quali operazioni seguire e quali goal soddisfare sulla base di risorse che detiene.

Tipologie di planning

- **Domain Specific:** si specificano le azioni sulla base del dominio di interesse.
- **Domain Independent:** si scrivono algoritmi che funzionano su qualsiasi dominio di interesse.
- **Configurable:** Una via di mezzo è creare algoritmi che si adattano sulla base del dominio che viene passato in input.

Operatore-Azione e Notazione



Un operatore è una tripla $(nome, precondition, effect)$.

- Il *name* è un'espressione sintattica nella forma $n(x_1, \dots, x_n)$ dove:
 - n è un simbolo di operatore che deve essere **univoco**.
 - x_1, \dots, x_n è una lista di tutti i parametri.
- *precond* è l'insieme di **letterali** che devono essere *true* affinché l'azione possa essere eseguita.
- *effects* sono i **letterali** che l'operatore renderà true.

La notazione che si utilizzerà sarà:

- S^+ : Insieme dei letterali positivi in S.
- S^- : Insieme dei letterali negativi in S.

Per ciò che riguarda l'azione si avrà:

- $precond^+(a)$: Insieme di atomi positivi che appaiono in a.
- $precond^-(a)$: Insieme di atomi negativi che appaiono in a.
- $effects^+(a)$: Atomi che diverranno positivi una volta applicata a.
- $effects^-(a)$: Atomi che diverranno negativi una volta applicata a.



In particolare, diremo che un'azione a è applicabile se:

- $precond^+(a) \subseteq S^+$
- $precond^-(a) \cap S = \emptyset$

L'esecuzione porterà a rimuovere dall'insieme S^+ i letterali negativi e invece inserirà i positivi:

$$\gamma(s, a) = (s - effects^-(a)) \cup effects^+(a)$$

Planning Problems

Dato un dominio di planning (Linguaggio L ed Operatori O) si può definire:

- **O** la collezione degli operatori.
- s_0 uno stato iniziale.
- g un insieme di letterali che si vuole rendere positivi.

Il **Planning problem** invece è una tripla $P = (\Sigma, s_0, s_G)$ dove:

- s_0 è lo stato iniziale.
- s_g è l'insieme dei goal.
- $\Sigma = (S, A, \gamma)$ che è un insieme di **state transition**.
 - S è l'insieme degli **atomi ground**.
 - A è l'insieme delle **istanze ground**.
 - γ sono le **funzioni di transizione**.



Il **Plan** è una sequenza di azioni $\pi = \langle a_0, \dots, a_n \rangle$ dove ogni a_i è un'istanza di un operatore O .
Una **Soluzione** è un piano che soddisfa tutti i goal.

Per velocizzare i **plan** si possono anche usare delle *euristiche*.

Anche diminuire il numero di precondizioni nelle azioni permette di velocizzare il problema garantendo una *semplificazione*.

Set-Theoretic

I goal sono rappresentati come se costanti.

Ad esempio se si desidera avere un goal del tipo $top(a,b)$ si crea una costante $topab$.

In questo modo si ha un'esplosione di costanti e per risolverlo si utilizza *SAT*.

Algoritmi di Risoluzione dei Problemi

Forward Search

A partire dallo stato iniziale si espandono i nodi applicando le varie azioni possibili fino a raggiungere un lo stato finale desiderato.

```
Forward-search( $O, s_0, g$ )
   $s \leftarrow s_0$ 
   $\pi \leftarrow$  the empty plan
  loop
    if  $s$  satisfies  $g$  then return  $\pi$ 
     $E \leftarrow \{a \mid a \text{ is a ground instance an operator in } O,$ 
      and  $\text{precond}(a) \text{ is true in } s\}$ 
    if  $E = \emptyset$  then return failure
    nondeterministically choose an action  $a \in E$ 
     $s \leftarrow \gamma(s, a)$ 
     $\pi \leftarrow \pi.a$ 
```



Caratteristiche

- **Sound:** è considerato corretto in quanto a partire da un problema risolvibile è in grado di generare un plan e dunque trovare una soluzione valida e corretta.
- **Completo:** nel caso esiste una soluzione, la troverà sempre.

Implementazioni

Può essere implementato in diversi modi:

- Ricerca in Ampiezza.
- Ricerca in Profondità.
- A^* .
- Ricerca Greedy.

Problema

Il problema principale è dettato dalla possibilità di avere un *branching factor* molto alto.

Inoltre una formulazione non precisa del problema potrebbe portare all'esplorazione di percorsi che non portano un effettivo progresso nel raggiungimento della soluzione.

→ Per questo è necessaria una buona funzione euristica.

Backward Search

Si parte dallo stato finale e applicando una **Funzione di transizione inversa** $\gamma^{-1}(g, a)$ si prova a raggiungere lo stato iniziale.

Affinchè una scelta sia valida si devono verificare le seguenti condizioni:

- L'azione deve contribuire a raggiungimento dell'obiettivo rendendo vero almeno uno dei letterali nell'obiettivo.
- L'azione non deve negare alcuna parte dell'obiettivo.

```
Backward-search( $O, s_0, g$ )
   $\pi \leftarrow$  the empty plan
  loop
    if  $s_0$  satisfies  $g$  then return  $\pi$ 
     $A \leftarrow \{a | a \text{ is a ground instance of an operator in } O$ 
      and  $\gamma^{-1}(g, a) \text{ is defined}\}$ 
    if  $A = \emptyset$  then return failure
    nondeterministically choose an action  $a \in A$ 
     $\pi \leftarrow a.\pi$ 
     $g \leftarrow \gamma^{-1}(g, a)$ 
```

Il problema è che navigando in senso opposto il numero di azioni che potrebbero portare allo stato è molto più elevato, di conseguenza ci potrebbe essere molte azioni ridondanti

Per risolvere questo problema si applica l'operazione di **Lifting** che non fa altro che raggruppare tutte le operazioni che sono *simili*.

Ad esempio se per raggiungere il posto X si possono usare sia auto che treni ed esiste l'operazione di $move(transport, place)$, si può codensare tutto in $move(X, place)$ decrementando il numero di azioni.

Strips

Dato che solitamente i *goal* sono visualizzabili come insiemi separati di letterali da rendere positivi, tramite questo algoritmo si punta a risolvere singolarmente i goal.

NOTA: Si può applicare l'algoritmo solo se il goal è *Serializzabile*.


```

 $\pi \leftarrow$  the empty plan
do a modified backward search from  $g$ :
    instead of  $\gamma^{-1}(s,a)$ , each new set of subgoals is just  $\text{precond}(a)$ 
    whenever you find an action that's executable in the current state,
    go forward on the current search path as far as possible,
    executing actions and appending them to  $\pi$ 
    repeat until all goals are satisfied

```

Sussman Anomaly

Il problema dell'algoritmo Strips è dato dal fatto che questa tipologia non tiene conto degli effetti a catena delle azioni generando un fallimento da parte del planner.

Block Stacking Alghoritm

È un algoritmo specifico per problemi di blocchi, segue regole definite per muovere i blocchi in modo ordinato. L'algoritmo è molto semplice ma non è detto che trovi sempre la soluzione ottima,
 → Possibile soluzione al problema di Sussman.

```

loop
    if there is a clear block  $x$  that needs to be moved
        and  $x$  can be moved to a place where it won't need to be moved
        then move  $x$  to that place
    else if there's a clear block  $x$  that needs to be moved
        then move  $x$  to the table
    else if the goal is satisfied then return the plan
    else return failure
repeat

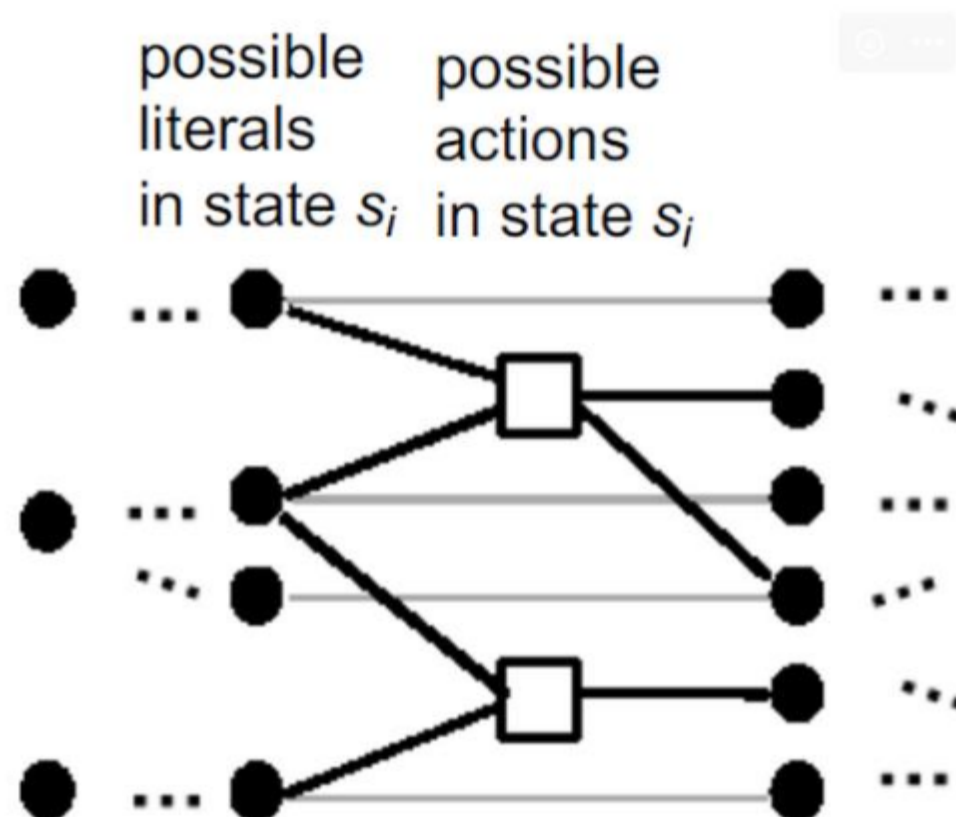
```

GraphPlan

Per ridurre il fattore di *diramazione* si potrebbe creare un problema rilassato restringendo il numero di ricerche da dover effettuare.

Si espande il grafo per k livelli e si ricerca una possibile soluzione, **se viene identificata** allora si effettua una *backward search* per restituire il plan.

Ogni livello del grafo rappresenta una possibile evoluzione del sistema dopo l'applicazione di un'azione.



The Planning Graph

Il grafo può essere visualizzato come un alternarsi di *stati* e *azioni*

Ci sono tre elementi:

1. **Nodo Stato** (cerchio): Letterale che può essere o meno vero al passo i .
2. **Nodo Azione** (quadrato): Azione che è possibile eseguire al passo i .
3. **Archi**: In ingresso indica una *pre-condizione* in uscita indica un *effetto*.

Un aspetto che deve essere garantito dal grafo è la **Mutua Esclusione**.



Due azioni sullo stesso livello sono *mutualmente esclusive* se NESSUNA delle seguenti condizioni si verifica:

- Producono effetti inconsistenti; una nega l'altra
- Interferenza; una cancella una pre-condizione dell'altra
- Competing Needs; le pre-condizioni di una esclude quelle dell'altra.

Due letterali sullo stesso *livello di stato* sono mutualmente esclusivi se:

- Uno è la negazione dell'altro.
- Hanno un supporto inconsistente: tutti i modi per raggiungerli sono mutualmente esclusivi a coppie. Le azioni che portano a raggiungere quei letterali sono mutualmente esclusive

▼ Spiegazione Chat GPT

Il secondo aspetto della **mutua esclusività** tra due letterali su uno stesso livello di stato si riferisce a situazioni in cui **tutte le azioni** o i **percorsi** che portano al raggiungimento di quei letterali sono, a loro volta, **mutualmente esclusivi**. In altre parole, due letterali possono essere mutualmente esclusivi non solo perché uno è la negazione dell'altro (come nel primo punto), ma anche perché **non è possibile ottenere entrambi contemporaneamente** a causa delle azioni necessarie per raggiungerli.

Vediamo un esempio più concreto per chiarire questo concetto:

Esempio

Immaginiamo una situazione in cui abbiamo due letterali L_1, L_2 su un certo **livello di stato** (livello in un **grafico di pianificazione**):

- L_1 potrebbe essere "La porta è aperta".
- L_2 potrebbe essere "La porta è chiusa".

Entrambi questi stati non sono direttamente negazioni l'uno dell'altro (come lo sarebbero, ad esempio, "La porta è aperta" e "La porta non è aperta"), ma è chiaro che **non possono verificarsi simultaneamente**: la porta non può essere aperta e chiusa allo stesso tempo.

Ora supponiamo che ci siano due azioni che possono portare a questi stati:

- **Azione A_1** : "Apri la porta" porta a L_1 (la porta è aperta).
- **Azione A_2** : "Chiudi la porta" porta a L_2 (la porta è chiusa).

Ora, queste due azioni A_1, A_2 sono **mutualmente esclusive** tra loro perché:

1. Non possono essere eseguite contemporaneamente (aprire e chiudere la porta allo stesso momento è impossibile).
2. Sono azioni che si escludono a vicenda: eseguire una impedisce il successo dell'altra.

Quindi, siccome le azioni che portano a L_1, L_2 sono **mutualmente esclusive**, anche i letterali L_1, L_2 diventano **mutualmente esclusivi** su questo livello di stato. In altre parole, **tutti i modi per raggiungere uno di questi letterali escludono automaticamente l'altro**.

Generalizzazione

Il principio generale è che due letterali sono mutualmente esclusivi per **supporto inconsistente** quando **tutte le combinazioni di azioni** che portano al raggiungimento di quei letterali sono **mutualmente esclusive** tra loro. Ciò significa che **non esiste alcuna combinazione di azioni** che possa far verificare contemporaneamente entrambi i letterali, rendendo quindi impossibile che si verifichino entrambi nello stesso livello di stato.

Questo concetto è fondamentale nella costruzione di **grafi di pianificazione** o **pianificatori basati su grafici**, dove l'obiettivo è identificare conflitti logici e temporali tra azioni e stati.

Gli archi stessi potrebbero rappresentare la *mutua esclusività*.

Fast Forward FF

È molto simile al graph plan.

Va in avanti e l'euristica che utilizza è quella di ignorare le liste di cancellazioni. Non cancellando è ovvio che prima o poi compariranno gli elementi che compongono il goal.

Si ottiene un rilassato che non garantisce soluzione.

Nella pratica funziona abbastanza bene.

Ci sono diversi modi di rilassare → ignorare qualcosa nelle pre-condizioni o negli effetti; FF ignora qualcosa negli effetti in particolare le liste di cancellazioni.

Answer Set Programming

Tratta la *programmazione logica disgiuntiva* che permette di risolvere problemi più complessi di quelli risolti tramite SAT.

È possibile catturare problemi della *seconda gerarchia polinomiale*, in particolare $\Sigma P2 \wedge \Delta P3$.

Catturare non significa risolvere, ma significa poter rappresentare i problemi tramite quel linguaggio.

Il linguaggio utilizza una *logica* estremamente **dichiarativa**, di conseguenza non importa l'ordine ma semplicemente la modalità di espressione.

Ciò avviene in quanto descriviamo una **proprietà del problema** ma non si esplicita come risolverlo.

Il vantaggio di questo tipo di modellazione è che se si cambia la logica del programma NON si devono modificare molte linee di codice, ma è sufficiente modificare la *regola logica* che definisce quell'aspetto.

Datalog



È un linguaggio che implementa il paradigma di *programmazione ASP*.

Le convenzioni sono le seguenti:

- *Minuscolo* le costanti.
- *Maiuscolo* le variabili.

Un **Atomo** è dato da un *predicato* con i suoi argomenti.

Dunque è un predicato istanziato.

$$p(t_1, \dots, t_n) \tag{1}$$

Un *atomo* può essere anche essere negato.

In particolare un *atomo*, che sia negato o meno, prende il nome di *letterale*.



Datalog è la parte *non disgiuntiva* di DLP.

Un **programma Datalog** è un insieme di regole sviluppate nella seguente forma:

$$Rule : a : -b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m$$

La regola è composta da una testa $H(r)$ e da un body $B(r)$. In particolare il corpo può essere suddiviso in parte positiva $B^+(r)$ e parte negativa $B^-(r)$.

Il corpo è una *congiunzione* di atomi legati da una serie di *AND*.

Sia corpo che testa sono *atomi*.

Esiste anche una versione più primitiva di Datalog che prende il nome di **pure datalog** in cui non esistevano disgiunzioni e negazioni.

Facts



I **Fatti** sono regole ground, dunque prive di corpo.

L'assenza di un corpo indica che quella regola è sempre vera a prescindere dalle condizioni degli altri atomi.

Più in generale possono essere immaginati come gli elementi della *conoscenza principale*.

Un aspetto di interesse è la differenza tra *predicati EDB* e *IDB*.

- **EDB:** I predicati che appaiono solo nei corpi o nei fatti, prendono il nome di **predicati della base estensionale**.
- **IDB:** Sono i predicati definiti dalle regole.

Intuitivamente l'EDB è il database che contiene le conoscenze sullo stato, tramite l'answer Set si possono fare delle query che ci confermano o meno delle constatazioni logiche.

Ricorsione

È spesso utilizzata la **ricorsione**.

Analizziamo l'esempio del calcolo dell'antenato

$$\begin{aligned} ancestor(A, B) &: \neg parent(A, B) \\ ancestor(A, C) &: \neg ancestor(A, B), ancestor(B, C) \end{aligned}$$

Interpretazioni



Sia P un *programma*, I un insieme di atomi ground.

Si avrà che:

$$\begin{aligned} \text{atomo } l \text{ è vero} &\implies l \in I \\ \text{atomo } l \text{ è falso} &\implies l \notin I \end{aligned}$$

I risulta essere un'interpretazione per P .



Sia P un *programma* ed M un'interpretazione.

Allora M è un modello se tutte le regole sono soddisfatte.

$$\begin{aligned} &\forall r, H(r) : \neg B(r), r \in P \\ &\left\{ \begin{array}{l} H(r) \in M \\ \text{oppure} \\ B^+(r) \not\subset M \vee B^-(r) \cap M \neq \emptyset \end{array} \right. \end{aligned}$$

In generale avere un corpo vero implica avere una testa vera. Avere una testa vera non implica avere un corpo vero.

B(r)	H(r)	B(r)⇒H(r)
1	1	1
1	0	0
0	1	1
0	0	1

Esempio

a :- b, c
c :- d
d

I modelli di questo programma sono i seguenti:

- $M = \{a, c, d\}$

- $M_2 = \{c, d\}$

In particolare il secondo modello M_2 è **minimale**.



Un **Modello è Minimale** se $\forall I \subset M, I$ non è un modello.

Ciò significa che eliminare una qualsiasi altra variabile dal set renderebbe nullo il Modello.

Teorema



Se **P** non contiene negazioni, allora esiste un unico modello minimale, chiamato **modello minimo**.

Si può avere anche una regola del tipo

$$: \neg B(r)$$

Quando la testa è vuota significa che la regola è falsa, se in una qualsiasi interpretazione si rende positiva questa regola allora si è violato il vincolo.

Operatore Conseguenze Immediate



Sia I un'interpretazione e P un programma ground, allora:

$$T_p(I) = \{a \mid \exists r \in P \text{ con } a \in H(r) \wedge B(r) \text{ vero rispetto ad } I\}$$

Cioè, le **conseguenze immediate** sono l'insieme di tutti gli atomi a tali che esiste una **regola R** tale che:

1. a è la testa di r .
2. Il corpo di r è vero relativamente ad I .

→ Su un programma Datalog Positivo P , T_p ha sempre un **fixpoint minimo** coincidente con il modello minimale di P .

Il ragionamento è semplice, si parte dall'insieme vuoto $\{\}$ e man man si inseriscono le *conseguenze immediate* fino a quando non si raggiunge il **fixpoint**.

Esempio

$$\begin{cases} a : \neg b, c \\ c : \neg d \\ d \end{cases}$$

- Step 1: $I_0 = \{\}$
- Step 2: $I_1 = \{d\}$
- Step 3: $I_2 = \{d, c\}$

Non si deriva nulla di nuovo dopo di questo punto dunque si è raggiunto il punto fisso e ci si ferma.

Si può notare come si è riusciti ad arrivare al modello minimo da un problema maggiore, ma ciò è sempre possibile?

Se partiamo dall'insieme vuoto e si gestiscono le *conseguenze immediate* man mano si. Se si ha un punto di partenza invece ciò non vale.

Inoltre, non si può affermare che l'**output** dell'operatore delle conseguenze immediate è un modello, perchè il risultato è dato a convergenza e potrebbero esserci elementi extra.

Teorema 1



Se $I = T_p(I)$, ovvero se l'interpretazione è un punto fisso dell'operatore $T_p(_)$ allora I è un modello

Supponiamo per assurdo che I non sia un modello.

Significa che esiste almeno una regola il cui il corpo è vero ma la testa è falsa → Una **Regola Violata**.

$$\exists r \in P : a = H(r) \notin I \wedge B(r) \text{ è vero rispetto ad } I$$

Però per ipotesi sappiamo che $I = T_p(I)$ di conseguenza, per definizione, la **regola r** deve derivare necessariamente **a**.

Dunque $a \in T_p(I)$ perciò si ha una contraddizione.

Teorema 2



Dato un **Programma Positivo Ground P**, il punto fisso T_p ottenuto a partire da $T_p^\uparrow(\emptyset)$ è il modello minimale di P .

La dimostrazione procedere per induzione.

In generale sappiamo che $T_p(_)$, se fatto partire da \emptyset , accumula tutti gli atomi del punto fisso

$T_p^\uparrow(\emptyset)$, dunque è sufficiente dimostrare che $I_i = T_p(I_{i-1})$.

- Al **Passo 1** I_0 conterrà esclusivamente i fatti del *programma P*, perciò l'induzione è verificata.
- **Passo Induttivo:** Al generico passo k avremo l'interpretazione I_k ottenuta applicando più volte l'operatore delle conseguenze immediate → $I_k = T_P(I_{k-1})$

La supposizione da fare è che I_k non è un punto fisso.

L'obiettivo è dimostrare che valga anche per I_{k+1} .

Considerando $I_{k+1} = T_P(I_k)$.

Gli atomi inseriti in I_{k+1} , dato l'operatore, sono tutti atomi che presentano un corpo che è positivo, perciò $B(r)$ è vero per ogni modello di P e di conseguenza lo è anche $H(r)$.

Ciò dimostra ciò che si voleva.

In particolare, partire dall'insieme vuoto ci permette sempre di ottenere il modello minimale in quanto si inseriscono informazioni che sono sempre necessarie, ma mai quelle solo sufficienti; questo avviene perchè l'operatore è **incrementale**.

Problemi Not-Ground

Nel caso di problemi non ground come ci si dovrebbe comportare?

Capita che i fatti vengano riportati esplicitamente in un file separato.

In generale, per risolvere i problemi not-ground, è sufficiente sostituire tutte le possibili costanti alle variabili → Simile ad un problema di soddisfacimento dei vincoli.

Se ad esempio alla variabile X associo il valore "Value1", allora per tutte le ricorrenze di X devo associare il valore "Value1".

Però la tecnica sopra descritta potrebbe portare a risultati insensati (che senso avrebbe associare ad una variabile che rappresenta una persona la costante "oggetto")

Perciò l'obiettivo è sviluppare problemi che siano i più piccoli possibili.

Una tecnica diffusa è la *semi-naive-evaluation* in cui si inseriscono nuovi fatti solo quando si sviluppano a partire da una certa regola ad una certa iterazione.

Consideriamo il seguente problema:

$$\begin{aligned}
 grandParent(X, Y) &: \neg parent(X, Z), parent(Z, Y) \\
 &parent(a, b), parent(b, c)
 \end{aligned}$$

Alla prima iterazione si inseriscono i fatti:

$$I_0 = \{parent(a, b), parent(b, c)\}$$

A questo punto si verificano le regole che sono state derivate. In questo caso si nota come i due valori abbiano generato il fatto $grandParent(a, c)$ che può essere inserito nella nuova iterazione:

$$I_1 = \{parent(a, b), parent(b, c), grandParent(a, c)\}$$

Dato che non si ha altro da derivare ci si ferma a questo punto.

Un altro esempio

$$\begin{aligned}
 ancestor(X, Y) &: \neg parent(X, Z), parent(Z, Y) \\
 ancestor(X, Y) &: \neg parent(X, Z), ancestor(Z, Y) \\
 &parent(a, b), parent(b, c), parent(c, d)
 \end{aligned}$$

Alla prima iterazione si inseriscono i fatti:

$$I_0 = \{parent(a, b), parent(b, c), parent(c, d)\}$$

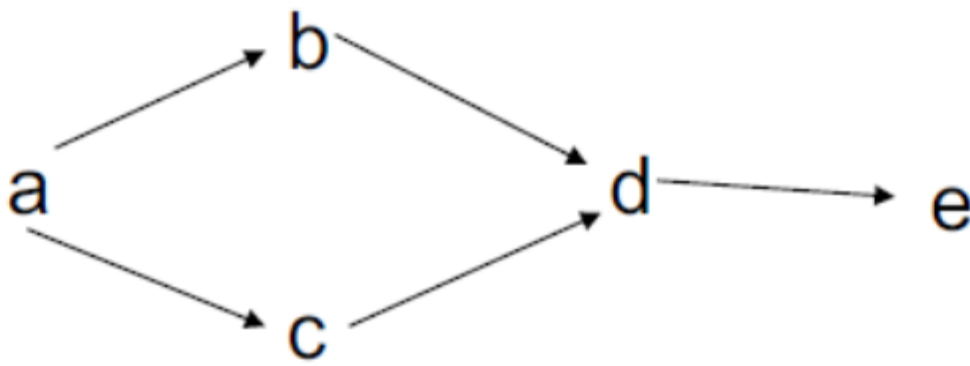
Inseriamo le regole derivate:

$$I = \{parent(a, b), parent(b, c), parent(c, d), ancestor(a, c), ancestor(b, d)\}$$

Al passo successivo si può valutare la seconda regola

$$\begin{aligned}
 I = \\
 \{parent(a, b), parent(b, c), parent(c, d), \\
 ancestor(a, c), ancestor(b, d), ancestor(a, d)\}
 \end{aligned}$$

Un altro esempio ancora è la copertura di un grafo



$$\begin{aligned}
 Path(X, Y) &: \neg Edge(X, Y) \\
 Path(X, Y) &: \neg Path(X, W), Edge(W, Y)
 \end{aligned}$$

Negazione

La presenza della negazione dà molto più potere espressivo.

Tutto ciò che non è dimostrabile tramite verità risulta essere falso.

Safety



Una regola è **Safe** quando:

- Tutte le variabili nella testa,
- Tutte le variabili in un letterale negativo
- Tutte le variabili in un operatore di confronto

Compaiono anche in un letterale positivo.

Cioè, tutte le variabili devono apparire positive almeno una volta nel corpo della regola.

Ad esempio:

- $S(X) : -a$ non è **safe** in quanto X compare solo nella testa, nel caso in cui a sia vero allora qualsiasi valore associato ad X sarà sempre vero, anche nel caso in cui non lo sia.
- $S(Y) : -b(Y), \text{ not } r(X)$: anche in questo caso non è **safe** in quanto X può essere sostituito da qualsiasi valore. In generale non si ha un *vincolo* su X.

Per sistemarla si dovrebbe avere qualcosa del genere $S(Y) : -b(Y), q(X), \text{ not } r(X)$

Recursion

Un altro problema che si incontra con la negazione è la ricorsione.

Consideriamo il seguente problema:

Example:

IDB: $p(X) :- q(X), \text{ not } p(X).$

EDB: $q(1). q(2).$

- Alla prima iterazione si inseriranno $Q = \{q(1), q(2)\}$ mentre $P = \{\}$
- Alla seconda iterazione però varrà la prima regola in quanto sia p(1) che p(2) sono falsi mentre è vero q(1) e q(2).

$$Q = \{q(1), q(2)\} \wedge P = \{p(1), p(2)\}$$

- Alla terza iterazione però la prima regola non varrà più in quanto sia p(1) che p(2) sono positivi e ci rimarranno quest

$$Q = \{q(1), q(2)\} \wedge P = \{\}$$

L'operatore delle conseguenze immediate è diventato dunque alternante e non più monotono e proprio per questo non termina più in un tempo *finito*.

Per risolvere questo problema si sfruttano i tipi di problemi che hanno la **Negazione Stratificata**.

Negazione Stratificata



Avere la **Negazione Stratificata** significa che quando si usa la *negazione* la si usa su predicati che non derivano dallo stesso predicato che si sta definendo, in questo modo non si ha una *ricorsione*.

A partire dalle *Negazione Stratificata* è possibile pure generare il **Dependencies Graph**.

Esempio 1

$$p(X) \text{ :- } q(X), \text{ not } p(X).$$

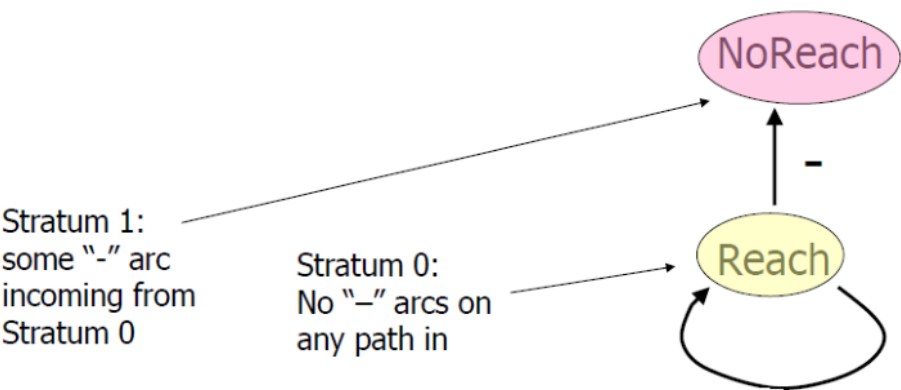


Il problema è non soddisfatto in quanto c'è un *ciclo* con all'interno un arco con un -.

NOTA: La negazione si contrassegna con il meno sull'arco.

Esempio 2

$$\begin{aligned} reach(X) &: - source(X). \\ reach(X) &: - reach(Y), arc(X, Y). \\ noReach(X) &: - target(X), not reach(X) \end{aligned}$$



Questi problemi prendono il nome di *stratificati* perchè si possono vedere come componenti create tra loro che formano un *grafo aciclico per le negazioni*.

La domanda sorge spontanea, ma perchè non inserire direttamente la negazione sul reach per generare il *NoReach*? Perchè altrimenti la regola non sarebbe *Safe*.

Minimal Models

Quando non si ha la negazione, un programma *Datalog* possiede un unico e solo modello minimale.

Con le negazioni è facile ricadere nel problema di avere più modelli validi e anche minimali.

$$a : - \text{ not } b$$

I modelli sono 2:

- Il primo è quando *b* è falso, dunque il modello sarà $\{a\}$.
- Il secondo quando invece *b* è vero, dunque il modello sarà $\{b\}$

In generale la *negazione* migliora la potenza **espressiva** ma peggiora la **complessità**.

Disjunctive Logic Programming

Gli elementi più importanti inseriti in aggiunta in questa tipologia di programmazione sono:

- **Weak Constraints:** che sono vincoli che si preferirebbe avere soddisfatti, ma non è detto che debbano esserli.
- **Funzioni Aggregate:** che permettono di eseguire delle funzioni aggiuntive.

Interpretazione Regola Disgiuntiva

Se si inseriscono delle disgiunzioni nella testa allora vale il seguente ragionamento: se il **corpo è vero** allora almeno uno dei due elementi nella testa è vero.

Il problema è che non si sa a priori quale è quello vero.

In generale, dato che si prova a minimizzare, si sceglie solo uno dei due atomi come positivo (a meno di altre regole che lo vincolano), però in generale nessuno vieta di considerare positivi entrambi gli atomi.

$$a_1 | \dots | a_n : -b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, b_m$$

La difficoltà maggiore è, appunto, nel trovare i modelli minimali.

Supponiamo di avere il seguente set di regole:

$$\begin{aligned} & InterestedInDLP(john) | isCuriousInDLP(john) : -attendsDLP(john) \\ & attendsDLP(john) \end{aligned}$$

Considerando il corpo vero si hanno 3 diversi modelli:

- $M_1 = \{Interested, Curioso\}$
- $M_2 = \{Interested\}$
- $M_3 = \{Curioso\}$

Nonostante siano tutti e 3 validi, solo $M_2 \wedge M_3$ sono minimali.

Program Instantiation



Un'istanza di un *Programma Ground* significa che ogni variabile deve essere sostituita con una costante dell'universo di *Herbrand*.

Si hanno i seguenti elementi:

- U_P : o **Universo di Herbrand** è l'insieme delle costanti che compaiono in un programma P .
- B_P : o **Base di Herbrand** è l'insieme degli *atomi ground* costruibili da U_P e dall'*insieme dei predicati*.
- Istanza Ground Di una Regola R: Ogni variabile nella regola R sostituita da una variabile che è in U_P .
- Istanziamento ground di un Programma: Insieme delle istanze ground delle regole.

Dunque, per creare un *programma ground* a partire da un *programma not ground* è sufficiente sostituire alle variabili le costanti ricavate da U_P .

Semantica per Programmi Positivi

La semantica dei programmi positivi è ciò che prende il nome di **Answer Set Programming**.



I è un **Answer Set** per il programma positivo P se è un *modello minimale*.

Semantica per Programmi con Negazione

In questo caso si fa riferimento al **ridotto**.



Un *Problema Ridotto* si indica con P^I ed è un **programma positivo**, che si ottiene tramite il meccanismo di risoluzione degli strati nei *programmi stratificati*.

Cioè si cancellano tutte le regole con un letterale *negativo falso* e si cancellano i *letterali negativi* dal corpo delle regole rimanenti.

Ad esempio:

$$\begin{array}{l} a : \neg d, \neg b \\ b : \neg \neg d \\ d \end{array}$$

A partire dal fatto d l'interpretazione ottenuta è $I = \{a, d\}$

| Un **Answer Set** è sempre un *modello minimale*.

Teoremi Utili

Teorema 1



Sia P un *programma logico*.

Se l'insieme vuoto $\{\}$ è un **Answer Set** per P , è unico.

Questo perchè, se $\{\}$ è un *Answer Set* è anche il modello minimo, che è unico in quanto lo si ottiene tramite l'operatore delle conseguenze immediate.

Teorema 2



Sia P un *programma logico*.

Sia $facts(P)$ il sottoinsieme di tutti i fatti di P .

Se S è un **AnswerSet** per P , allora $facts(P) \subseteq S$

Teorema 3



Sia P un *programma logico*.

Sia $facts(P)$ il sottoinsieme di tutti i fatti di P .

Se $facts(P)$ è un **AnswerSet**, allora è l'unico.

Teorema 4

Prima di definire il teorema 4 è necessario appuntare il significato di *Atomo Supportato*.

Atomo Supportato



Sia P un *programma logico*.

Sia I un'interpretazione per P .

un atomo $a \in I$ si definisce **supportato in I** se $\exists rule\ r \in P : body(r)$ è vera ed è vero che $head(r) \cap I = a$

Cioè a è l'unico atomo vero che appare nella testa di r .

Consideriamo il seguente esempio:

$$\begin{array}{l}
d \\
c : \neg d \\
k : \neg \neg d \\
a|b : \neg c \\
e|c
\end{array}$$

Si parte dalla seguente interpretazione: $I = \{d, c, k, a, e\}$

In particolare d, c, a sono supported in quanto sono derivati dalle regole.

k, e invece non sono supportati perchè sono “forzatamente” inseriti nell’interpretazione.

Ritornando al teorema:



Sia P un programma logico.

Sia I un modello per P .

I è un **AnswerSet** per P solo se $\forall a \in I, a$ è un atomo supportato.

Ad esempio:

$$\begin{array}{l}
a|b \\
b|c \\
a|c
\end{array}$$

L’**AnswerSet** corretto che segue questo teorema è il seguente $I = \{a, b\}$.

Perchè:

- b lo si ricava anche dalla seconda regola
- a lo si ricava anche nella terza

Quindi la prima regola non è supportata, ma le altre due sì.

Cioè sono tutti dei fatti (body vuoto) se scegliessimo c come answer set la prima regola sarebbe totalmente falsa e non può avvenire in quanto sono fatti, appunto.

Esempio: 3-COL

Il primo step è capire quali sono i nodi collegati tra di loro:

$$edge(x, y) : \neg edge(y, x)$$

In questo modo si garantisce la simmetria.

Si definisce cosa è un nodo:

$$node(x) : \neg edge(x, _)$$

Per noi un nodo è tutto ciò che è collegabile.

Definiamo la colorabilità, essendo 3-COL consideriamo Rosso, Blu, Verde

$$col(R, x) \vee col(B, x) \vee col(G, x) : \neg node(x)$$

Infine è necessario specificare cosa NON deve avvenire, mettiamo una regola con testa vuota (deve essere sempre falsa) dove si definisce che due nodi vicini non devono avere lo stesso colore:

$$: \neg edge(x, y), col(C, x), col(C, y)$$

Estensione K-COL

In questo caso si avranno in più le definizioni dei colori

$$Color(C_1), Color(C_2), \dots$$

A questo punto è possibile definire la colorazione:

$$Col(C, X) : \neg node(X), Color(C), \neg diffCol(X, C)$$

Il *diffCol* serve per poter indicare che quel nodo non deve essere colorato di NESSUN altro colore

$$diffCol(C, X) : \neg color(C), col(C_1, X), C <> C_1$$

la parte *color(C)* serve esclusivamente per la **safety**, cioè dobbiamo avere nel corpo gli elementi che nella testa sono negati.

Infine come prima la regola con testa vuota per indicare la colorabilità:

$$: \neg col(C, X), col(C, Y), edge(X, Y)$$

Esempio: Vertex Cover



Sia $G = (V, E)$ il grafo.

S è un **vertex cover** se $\forall e \in E, S \cap e \neq \emptyset$

Il ragionamento è semplice, un nodo può o non può essere nel vertex cover

$$vc(x) \vee nvc(x) : \neg vertex(x)$$

In questo modo TUTTI i nodi potrebbero non far parte, per questo si vanno a scartare i nodi che non soddisfano la condizione, cioè se c'è un arco, almeno uno dei due deve stare dentro l'insieme:

$$: \neg edge(x, y), \neg vc(x), \neg vc(y)$$

L'approccio utilizzato prende il nome di **Guess and Check**, in cui una parte delle regole provano ad indovinare in maniera non deterministica la condizione, e poi la restante parte effettua l'effettivo **check**.

Guess and Check

Si suddivide in due fasi:

- Una *regola disgiuntiva* prova ad indovinare la **soluzione candidata**.
- I **vincoli di integrità** controllano la sua ammissibilità.

Più in generale, il guess genera lo spazio degli stati, il check definisce il pruning.

Weak Constraints

Sono vincoli che è possibile violare.

Lo si utilizza in ASP per gestire problemi di *ottimizzazione*.

Non si devono vedere come vincoli opzionali, ma da violare solo in caso di necessità.

Si può dire che permettono di esprimere delle preferenze.

Bisogna ricordare che

Programmazione senza Disgiunzione	Problemi al più NP
Programmazione con Disgiunzione	Problemi al più SigmaP2

In generale un Weak Constraint è caratterizzato da una testa vuota e presenta una tilde:

$$:\sim rule$$

L'obiettivo è generare gli **answer set** di cardinalità minima.

In particolare, dato che i *weak constraint* rappresentano delle preferenze, ci sono diversi pesi che possono essere associati a questi.

I **Pesi** possono essere visti come valori di *priorità*.

$$:\sim p(x, y), q(x, z) [z@2]$$

Dove z è il peso e 2 è il livello di priorità.

Teoria dei Giochi

I Giochi vengono classificati sulla base di due parametri: **Struttura** e **Informazioni**

Struttura	Informazioni
Forma normale	Informazione Perfetta
Forma estensiva	Informazione Imperfetta
	Informazione Incompleta

I giocatori, sulla base delle strategie adottate generano una certa **utilità**.

Ovviamente, l'esito finale non dipende solo dalle proprie scelte, ma anche da quelle degli avversari.

- Nei **Giochi in Forma Normale** i giocatori compiono un'unica azione in contemporanea con gli altri.
- Nei **Giochi in Forma Estensiva** i giocatori compiono un'azione in maniera sequenziale a turni.

Per quanto riguarda le informazioni:

- Nei **Giochi ad Informazione Imperfetta** il giocatore non è a conoscenza delle azioni passate.
- Nei **Giochi ad Informazione Incompleta** la differenza è che l'assenza di informazioni è dettata dalle regole.
- Nei **Giochi ad Informazione Perfetta** i giocatori conoscono sia le mosse degli avversari che le loro utilità.

Altre tipologie di giochi sono quelli *cooperativi* e *non cooperativi*.

Aste

Alcune volte le regole sono ben definite dal gioco, a volte siamo noi stessi a definirle in modo tale che canalizziamo l'esito verso un nostro stato di favore.

Nelle aste si hanno

$$\begin{aligned} &Agenti : p_1, \dots, p_m \\ &Oggetti : o_1, \dots, o_n \end{aligned}$$

Gli agenti attribuiscono ad ogni *oggetto* un determinato **valore**, questo ovviamente rimane segreto.

$$Values : v_1, \dots, v_m$$

Quale è l'utilità di un *agente i*?

$$\begin{cases} u_i = v_i - c_i \text{ dove } c_i \text{ è il prezzo pagato per l'oggetto} \\ u_i = 0 \text{ se } i \text{ non ha vinto l'asta} \end{cases}$$

È da valutare se conviene o meno giocare in maniera **truthful**.

Tipi Di Aste

1. Asta all’Inglese

- Si parte da un prezzo base.
- Ogni partecipante fa offerte progressivamente più alte.
- L’asta termina quando finisce il tempo o quando nessuno rilancia.
- **Vince:** chi ha fatto l’offerta più alta.

2. Asta Giapponese

- Il prezzo aumenta automaticamente a intervalli regolari e prestabiliti.
- I partecipanti decidono se restare o ritirarsi man mano che il prezzo sale.
- **Vince:** chi rimane fino alla fine.

3. Asta Olandese

- Si parte da un prezzo molto elevato.
- Il banditore riduce gradualmente il prezzo fino a quando un partecipante lo blocca.
- **Vince:** chi blocca la discesa, aggiudicandosi l’oggetto al prezzo annunciato.

4. Asta a Busta Chiusa (First Price Auction)

- Ogni partecipante presenta un’offerta a busta chiusa.
- Quando le buste vengono aperte:
 - **Vince:** chi ha fatto l’offerta più alta, che paga il prezzo offerto.

5. Asta Second Price

- Simile all’asta a busta chiusa.
- Ogni partecipante presenta un’offerta a busta chiusa.
- Quando le buste vengono aperte:
 - **Vince:** chi ha fatto l’offerta più alta, ma paga un prezzo pari alla **seconda offerta più alta**.

In particolare nelle aste *Second Price* è sempre conveniente adottare una strategia *truthful*.

Verifichiamo il perchè analizzando le diverse situazioni.

Si parte dal concetto base dell’utilità:

$$u_i = v_i - c_i$$

In particolare, v_i è il valore che attribuiamo all’oggetto, c_i è quanto effettivamente lo paghiamo.

Ogni partecipante, una volta iniziata l’asta, dovrà effettuare una **bid** che identificheremo con b_i .

- **CASO 1:** $b_i > v_i$

Nel caso in cui l’agente riuscisse ad aggiudicarsi l’asta pagherebbe un prezzo pari a b^* .

Ma dato che $b_i > b^*$ nessuno assicura che $b^* \leq v_i$.

In questo caso l’agente pagherebbe più di quanto valuta l’oggetto andando ad avere un *utilità negativa*.

- **CASO 2:** $b_i < v_i$

L’agente potrebbe rischiare di non vincere l’asta e perdere l’oggetto.

- **CASO 3:** $b_i = v_i$

In questo caso l’agente offre quanto realmente valuta l’oggetto andrà sempre in positivo con l’utilità.

Perchè o perde l’asta e perde l’oggetto, oppure vince l’asta ma siccome si deve prendere la seconda offerta più alta si avrà

$$b_i > b^*, b_i = v_i \implies v_i > b^* \\ \text{dunque} \\ u_i = v_i - b^* >$$

Giochi Strategici



In questa tipologia i giocatori effettuano le scelte in maniera totalmente *indipendente* senza sapere cosa faranno gli avversari.

Ogni giocatore possiede una propria preferenza i rispetto ai possibili risultati.

Se un giocatore i preferisce un insieme di risultati ottenuti a partire dal profilo di azioni a rispetto al profilo di azioni b allora lo si indica con

$$a \sim^i b$$

Questa è la vera differenza rispetto ad un problema di *ottimizzazione*, entrano in gioco anche le scelte degli avversari.

Per rendere le preferenze matematicamente comparabili è preferibile utilizzare una **funzione di payoff** $u_i(\cdot)$.

In particolare:

$$u_i(a) \geq u_i(b) \text{ iff } a \sim^i b$$

Avere queste funzioni permette di generare una certa **ordinalità** nelle scelte.

Per capire meglio vediamo un esempio

Esempio: Prisoner’s Dilemma

	Confess	Don’t Confess
Confess	-3,-3	0,-4
Don’t confess	-4,0	-1,-1

I valori contenuti nelle celle sono i valori di *payoff* dei due giocatori.

Si può notare come non ci sia un effettivo *equilibrio stabile* in quanto la scelta migliore per entrambi sarebbe di ***non confessare***, però per il bene del singolo (dando per scontato che l’altro non parlerebbe) sarebbe meglio ***confessare***.

Strategie Pure

Nelle *Strategie Pure* è possibile scegliere un’unica azione dall’insieme delle *possibili azioni*.

La scelta è dunque **deterministica**.

Nel caso di ***Strategie Miste*** entra in gioco la randomicità.



Un **Gioco Strategico** è un gioco composto da:

- L’insieme N dei giocatori.
- Per ogni giocatore $i \in N$ l’insieme S_i delle sue possibili azioni.
- Per ogni giocatore $i \in N$ la sua relazione di preferenza \sim^i sull’insieme dei profili di azione $S \equiv \times_{j \in N} S_j$

Il profilo di azione non è altro che la combinazione delle azioni scelte da tutti i giocatori.

Il prodotto cartesiano permette di identificare l’insieme delle combinazioni tra le proprie mosse e quelle del giocatore j .

Alcune notazioni:

- s_i è l'azione adottata dal giocatore i .
- s_{-i} sono tutte le azioni degli avversari esclusa quella del giocatore i .
- Di conseguenza l'insieme di azioni di tutti i giocatori escluso i sarà $S_{-i} \equiv \times_{j \in N \setminus \{i\}} S_j$

Equilibrio di Nash



In un gioco strategico il profilo di azioni s^* è un **equilibrio di Nash** se per ogni giocatore i ed ogni sua azione s_i , s^* è altrettanto buono quanto il profilo di azioni (s_i, s_{-i}^*) rispetto alle preferenze di i .

$$u_i(s^*) \geq u_i(s_i, s_{-i}^*), \forall s_i \in S_i, \forall i \in N$$

In generale ciò significa che dettato il *comportamento degli avversari*, il **giocatore i** non è incentivato a modificare la sua mossa.

Per intenderci:

- $u_i(s^*)$ è il valore di utilità per il giocatore i considerando la mossa s^*
- $u_i(s_i, s_{-i}^*)$ sarebbe il valore di utilità ottenuto nel caso in cui il giocatore i modificasse la sua mossa ma gli altri giocatori la lasciassero inalterata.

Dunque, si ha un **equilibrio di Nash** quando la mossa scelta è la migliore per se stessi e cambiarla andrebbe a peggiorare semplicemente l'utilità.

Se ciò vale per ogni giocatore si ha un equilibrio di Nash, cioè se ogni giocatore cambiando mossa peggiora la sua utilità.

Strettamente collegato all'**Equilibrio di Nash** si ha il concetto di migliore risposta.

Best Response



Per qualsiasi profilo di azione attuato dagli avversari del *giocatore i*, $s_{-i} \in S_{-i}$, si definisce tramite $B_i(s_{-i})$ la funzione **best response** come l'insieme delle migliori risposte che il giocatore i può adoperare dato il comportamento dei suoi avversari.

$$B_i(s_{-i}) \equiv \{s_i \in S : u_i(s_i, s_{-i}) \geq u_i(s'_i, s_{-i}), \forall s'_i \in S_i\}$$

Cioè tenendo fisse le mosse degli avversari, le best response sono le azioni che permettono di massimizzare l'utilità del giocatore i .

Tenendo conto di questa definizione l'**Equilibrio di Nash** lo si può intendere come:

$$s_i^* \in B_i(s_{-i}^*), \forall i \in N$$

Cioè se l'avversario risponde sempre con la propria *best response* a prescindere dal comportamento degli avversari.

Strategie Miste

Nelle strategie miste è necessario inserire un nuovo concetto: Le Lotterie.

Solitamente ci si aspetta che ad un'**azione** corrisponda una **conseguenza** ed un giocatore sceglie le sue azioni per poter avere la **conseguenza desiderata**.

Però non è detto che la relazione tra *azione e conseguenza* sia **deterministica**, può accadere che sia *stocastica*.

L'idea è che le azioni sono delle *lotterie* e dobbiamo scegliere quale biglietto acquistare per partecipare a questa.

Si può intuire come il rapporto non è più diretto in quanto non si può sapere a priori quale sarà il risultato.



Sia Z un insieme di **conseguenze (premi) finito**.

Una *lotteria* P è una *distribuzione di probabilità* dove con $p(z)$ si indica la probabilità con la quale nella lotteria p si vinca il *premio* z .

Si indica con $\Delta(Z)$ l'insieme di tutte le *possibili distribuzioni*.

L'obiettivo è dunque definire le *relazioni di preferenza* su $\Delta(Z)$.

Il parametro da prendere in considerazione è quello dell'**Utilità Attesa**, indicata tramite $U(\cdot)$.

Ad ogni premio è dunque associato un valore $v(z)$, mentre la *lotteria* è valutata in base al valore atteso.

Dunque un giocatore che preferisce la lotteria p alla lotteria q seguirà i seguenti parametri:

$$U(p) \equiv \sum_{z \in Z} p(z)v(z) \geq U(q) \equiv \sum_{z \in Z} q(z)v(z)$$

Partendo da questo concetto è fondamentale che nelle *strategie miste* i giocatori abbiano le loro preferenze.

Si pone $\Sigma \equiv \Delta(S)$, in generale $\Sigma_i \equiv \Delta(S_i)$.

Ogni giocatore sceglie una *strategia mista*, che non è altro che una lotteria e dunque una distribuzione di probabilità sulle strategie pure.

$$\sigma_i \in \Sigma_i \equiv \Delta(S_i)$$

Anche in questo caso si ha $\Sigma \equiv \times_{i \in N} \Sigma_i$.

Con la notazione σ_{-i} si intendono tutte le strategie miste adottate dagli avversari.

Per ogni strategia mista si indica con $\sigma_i(x)$ la *probabilità* che la strategia mista assegna alla corrispondente strategia pura.

Si definisce con supporto di σ_i tutte le *strategie pure* che hanno una probabilità strettamente positiva in σ_i .

$$\text{supp}(\sigma_i) \equiv \{s_i \in S_i : \sigma_i(s_i) > 0\}$$

Si suppone inoltre che le strategie miste dei vari giocatori siano indipendenti tra di loro.

Dunque, la probabilità che un *dato risultato si verifichi* è dato dal prodotto delle probabilità che ogni giocatore effettui la mossa che porterebbe a quel risultato:

$$\prod_{j \in N} \sigma_j(s_j)$$

L'utilità del singolo giocatore è ottenuta così:

$$U_i(\sigma) \equiv \sum_{s \in S} \left(\prod_{j \in N} (\sigma_j(s_j)) \right) u_i(s)$$

Ricordando che la notazione $\sigma_j(s_j)$ rappresenta la probabilità assegnata alla strategia mista σ_j ottenuta a partire dalla strategia pura s_j .

Tramite la somma si esplorano tutti i profili di strategie pure.

Con la produttoria invece si calcola la probabilità che il profilo s si verifichi prendendo in considerazione tutte le *strategie miste* scelte dagli altri giocatori.

In pratica si calcola l'utilità media del giocatore i .

Quando tutti gli insiemi S_i sono finiti allora si ha:

$$U_i(\sigma) \equiv \sum_{s_i \in S_i} \sigma_i(s_i) U_i(e(s_i), \sigma_{-i})$$

Anche in questo caso la sommatoria esplora tutte le strategie, però in questo caso esplora in particolare le strategie pure del giocatore i .

- $\sigma_i(s_i)$ è la probabilità che la strategia mista ottenuta a partire da s_i si verifichi.
- $U_i(e(s_i), \sigma_{-i})$ calcola l'utilità per il giocatore i di scegliere s_i tenendo conto che gli altri giocatori giocano secondo σ_{-i} .

Equilibrio di Nash Misto



In un *gioco strategico* un profilo di strategie σ^* è un **Equilibrio di Nash Misto** se per ogni giocatore i e per ogni strategia mista σ'_i egli non preferisce la lotteria indotta da $(\sigma'_i, \sigma^*_{-i})$ rispetto a quella indotta da σ^* .

$$U_i(\sigma^*) \geq U_i(\sigma'_i, \sigma^*_{-i}), \forall \sigma'_i \in \Sigma_i, \forall i \in N$$

Si ha un teorema il quale afferma:

|

Ogni gioco strategico finito possiede almeno un equilibrio di Nash Misto.

Anche in questo caso è possibile definire la Best Response

Best Response Mista



Per qualsiasi profilo di strategie miste attuato dagli avversari del giocatore i , $\sigma_{-i} \in \Sigma_{-i}$, definiamo tramite $B_i(\sigma_{-i})$ la *funzione best response* per il giocatore i , come l'insieme delle migliori strategie pure che il giocatore può adoperare dato il comportamento dei suoi avversari.

$$B_i(\sigma_{-i}) \equiv \{s_i \in S_i, u_i(s_i, \sigma_{-i}) \geq u_i(s'_i, \sigma_{-i}), \forall s'_i \in S_i\}$$

Teorema



Sia G un *gioco strategico finito*.

Il profilo di **strategie miste** $(\sigma_i^*)_{i \in N}$ è un **equilibrio di Nash Misto** se e solo se per ogni giocatore $i \in N$ tutte le strategie pure nel supporto di σ_i^* sono una best response alle strategie σ^*_{-i} .

In altre parole, un profilo di strategie miste è un equilibrio di Nash Misto se:

- Ogni giocatore massimizza le proprie utilità attese rispetto alle strategie degli altri.
- Tutte le strategie pure, con probabilità positiva nella strategia mista di ogni giocatore sono **best response** alla strategia mista degli altri giocatori.

Equilibrio di Nash Misto

Si ha il seguente problema:

	Bach	Stravinski
Bach	2,1	0,0
Stravinsky	0,0	1,2

Ad entrambi i giocatori conviene scegliere lo stesso artista per non andare da soli.

Si indicano con:

- $\sigma_1(B)$ la probabilità con cui il giocatore 1 scelga Stravinsky.
- $\sigma_2(B)$ la probabilità con cui il giocatore 2 scelga Stravinsky.

A partire dalle due mosse è necessario calcolare l'utilità attesa.

Questa sappiamo che si ottiene a partire dalla formula $U_i(s_i, \sigma_{-i})$.

Dunque si avrà che l'utilità attesa per il giocatore 1 nel caso in cui il giocatore 2 scelga Batch è la seguente:

$$U_1(B) \equiv 2 * \sigma_2(B) + 0 * (1 - \sigma_2(B))$$

NOTA: Si usa $1 - \sigma_2(B)$ perchè si hanno solo due casi e la somma delle probabilità deve essere 1.

Dunque l'utilità per il giocatore 1 è: $U_1(B) \equiv 2 * \sigma_2(B)$

Adesso calcoliamo l'utilità attesa nel caso in cui il giocatore 1 scelga Stravinsky

$$U_1(S) \equiv 0 * \sigma_2(B) + 1 * (1 - \sigma_2(B))$$

Che dunque è pari a $U_1(S) \equiv 2 * (1 - \sigma_2(B))$

Affinchè si abbia l'equilibrio di Nash i due giocatori devono essere **indifferenti** rispetto alle due proposte, quindi l'utilità deve essere uguale.

$$\begin{aligned} 1 - \sigma_2(B) &= 2 * \sigma_2(B) \\ \sigma_2(B) &= \frac{1}{3} \end{aligned}$$

Ora si effettua lo stesso per il giocatore 2.

$$\begin{aligned} U_2(B) &\equiv 1 * \sigma_1(B) + 0 * (1 - \sigma_1(B)) \\ U_2(B) &= \sigma_1(B) \end{aligned}$$

Per il caso 2

$$\begin{aligned} U_2(S) &\equiv 0 * \sigma_1(B) + 2 * (1 - \sigma_1(B)) \\ U_2(S) &= 2 - 2\sigma_1(B) \end{aligned}$$

Uguagliando ed effettuando i calcoli si avrà:

$$\sigma_1(B) = \frac{2}{3}$$

Coalitional Games

Sono i giochi in cui i giocatori possono collaborare tra di loro per poter ottenere una certa **worth**.

Ogni giocatore ha un goal da raggiungere e un insieme di possibili azioni.

I giocatori potrebbero essere sia agenti singoli che un gruppo.

Un gioco di *coalizione* è definito dai seguenti elementi:

- L'insieme degli N *giocatori*.
- Una *funzione* v , definita su 2^N che restituisce le possibili conseguenze associate assegnate ad ogni coalizione.
- Le relazioni di preferenza dei giocatori in N sui possibili risultati.

La coalizione che include tutti i giocatori prende il nome di **Grand Coalition**.

Le due domande principali dei giochi di coalizione sono:

1. Quali coalizioni si formeranno?
2. Come deve essere suddivisa la **worth** tra i vari membri della coalizione vincente?

Per il secondo punto si utilizzerà la **Transferable Utility Assumption**, questa afferma che il valore ottenuto può essere liberamente distribuito tra i membri è sufficiente che la valuta utilizzata sia universale.

Un aspetto caratteristico dei giochi di *coalizione* è il **Solution Concept**.

Solution Concept



Un **Solution Concept** è una *funzione* che associa ad ogni gioco alcuni possibili risultati per ciascun giocatore.

Anche se i giocatori sono in una *coalizione* ognuno di loro avrà un peso più o meno maggiore al fine di raggiungere l'obiettivo, dunque ognuno di loro deve aver assegnato un valore.

Ovviamente per un agente deve convenire partecipare ad una *coalizione*.

Quindi tramite il **Solution Concept** si assegnano i valori ai singoli.

Dunque, come detto in precedenza, ad ogni coalizione è associato un **worth**.

La distribuzione del **worth** avviene tramite la funzione v che prende il nome di **funzione caratteristica**.

$$G = \langle N, v \rangle, v : 2^N \implies \mathbb{R}$$

Dunque, $v(S)$ dà l'informazione su quanto una coalizione dovrebbe ricevere.

Si definisce un **insieme di imputazione** $X(G)$ (con imputazione si intende l'**assegnazione ammissibile di valore ad ogni giocatore**):

$$x \in X \begin{cases} Efficiency : x(N) = v(N) \\ IndividualRationality : x_i \geq v(\{i\}), \forall i \in N \end{cases}$$

Affinchè sia **efficiente** la somma totale guadagnata deve essere **distribuita per intero** a tutti i giocatori.

Affinchè valga la **Individual Rationality** il giocatore deve prendere almeno quanto prenderebbe se fosse in una coalizione da solo.

Tramite i **Solution Concepts** devono essere caratterizzati da:

- Fairness: Cercare di mantenere le cose eque tra i giocatori.
- Stability: Cioè i giocatori non devono andarsene a loro piacimento perchè lo ritengono più conveniente.

Esempio: Voting Game

Ci sono 4 partiti politici: A, B, C, D

Rispettivamente con 45, 25, 15, 15 rappresentanti ciascuno.

Per approvare una legge con un guadagno di 100 milioni di euro serve la maggioranza, dunque almeno 51 voti.

Le coalizioni vincenti appartengono all'insieme W : ad ogni coalizione che appartiene a W si assegna un valore pari ad 1: $v(S) = 1$.

Classi di Giochi Coalizionali

Esistono diverse classi di giochi:

- **SuperAdditive Games**

Per ogni coppia di *coalizioni disgiunte*, il valore della coalizione in cui si fondono queste due è maggiore della somma del valore delle coalizioni singole

$$\forall S, T \subset N, \text{ if } S \cap T = \emptyset, \text{ then } v(S \cup T) \geq v(S) + v(T)$$

- **Additive (or inessential) Games**

Il valore di una coalizione risultante è esattamente la somma dei valori delle coalizioni individuali che la compongono. In pratica, non ci sono sinergie né penalizzazioni.

Caratteristica: Non vi è alcun "vantaggio" o "perdita" derivante dalla collaborazione.

$$\forall S, T \subset N, \text{ if } S \cap T = \emptyset, \text{ then } v(S \cup T) = v(S) + v(T)$$

- **Giochi a Somma Costante**

La somma dei valori delle coalizioni complementari (cioè S e il resto dei giocatori N\S) è costante ed è sempre uguale al valore totale v(N).

Caratteristica: È un tipo di gioco a somma zero, dove il guadagno di una coalizione è una perdita per il resto.

$$\forall S \subset N, v(S) + v(N \setminus S) = v(N)$$

Ciò significa che se una coalizione prende una parte del totale, il resto va all'altra coalizione, la somma però deve rimanere sempre fissa.

Esempio: Se ci sono 10kg di torta e la prima coalizione prende 6kg, gli altri 4 devono andare necessariamente all'altra coalizione.

In particolare se S è una *coalizione vincente* allora $N \setminus S$ è una *coalizione perdente*.

- **Convex Games**

Si tratta di una generalizzazione dei giochi superadditivi. Il valore di una coalizione combinata tiene conto dell'intersezione tra le due coalizioni.

Caratteristica principale: Promuove le collaborazioni, poiché le coalizioni più grandi tendono ad avere vantaggi maggiori, riducendo le perdite derivanti dalle sovrapposizioni.

$$\forall S, T \subset N, v(S \cup T) \geq v(S) + v(T) - v(S \cap T)$$

Assiomi Per i Coalitional Games

- **Assioma di Simmetria**

I giocatori che sono interscambiabili devono ricevere lo stesso valore

$$\forall S \text{ che non contiene nè } i \text{ nè } j, v(S \cup \{i\}) = v(S \cup \{j\})$$

- **Dummy Players**

$$\forall S : i \notin S, v(S \cup \{i\}) = v(S)$$

La quantità $v(S \cup \{i\}) - v(S)$ è detto **contributo marginale**.

In altre parole, i giocatori dovrebbero ottenere un valore pari esattamente alla quantità che ottengono da soli.

- **Additività**

Dati due giochi con lo stesso insieme di giocatori, la somma delle loro *funzioni caratteristiche* produce un nuovo gioco.

Il valore del nuovo gioco per qualsiasi coalizione è dato dalla somma dei valori delle coalizioni nei due giochi originali.

$$\begin{aligned} \Psi_i(N, V_1 + V_2) &= \Psi_i(N, V_1) + \Psi_i(N, V_2) \\ \text{dove} \\ (V_1 + V_2)(S) &= V_1(S) + V_2(S), \forall S \end{aligned}$$

Shapley Value

È una **pre-imputazione** che soddisfa tutti gli assiomi.

Dato che **esiste sempre** non si può parlare di *pre-imputazione*.

La **contribuzione marginale** del giocatore si calcola nel seguente modo:

$$\phi_i(N, v) = \frac{1}{N!} \sum_{S \subset N \setminus \{i\}} |S|!(|N| - |S| - 1)! [v(S \cup \{i\}) - v(S)]$$

Lo *Shapley Value* rappresenta il contributo medio del **giocatore i** al valore della *coalizione* a cui partecipa, calcolato su tutte le possibili permutazioni dei giocatori.

- La coalizione S considerata non include il *giocatore* i .
- Il valore $v(S \cup \{i\}) - v(S)$ è il **contributo marginale** (dummy player) di i nella coalizione S .

Il valore di Shapley misura quanto ogni giocatore "aggiunge al gioco" in media, considerando tutte le possibili combinazioni di coalizioni.

La sua forza risiede nella combinazione degli assiomi che garantiscono un risultato equo e razionale.

Esempio: Voting System

ESEMPIO

TOT = 100

if votes ≥ 51

$v(S) = 1$

else $v(S) = 0$

A = 45

B = 25

C = 15

D = 15

Shapley Value di A

Tutte le coalizioni senza A:

$v(B, C, D) = 1$

$v(B, C) = 0$

$v(B, D) = 0$

$v(C, D) = 0$

$v(B) = 0$

$v(C) = 0$

$v(D) = 0$

CASO

$\{A, C, D\}$

↓

CASO CON 2

$\phi_A = \frac{1}{4!} \{ 3! \cdot 0! (1-1) + 3 \cdot [2! \cdot 1! (1-0)] +$

CASO CON 1

CASO DA SOLO

$+ 3 \cdot [1! \cdot 2! (1-0)] + 0! \cdot 3! (0-0) \} =$

$= \frac{1}{4!} \{ 0 + 6 + 6 + 0 \} = \frac{1}{24} \cdot 12 = \frac{1}{2} \rightarrow$

A Prende la metà del Totale

Lo *Shapley Value* però non soddisfa sempre la *proprietà di stabilità*.

Ricordando il concetto di stabilità: I giocatori non devono essere incentivati ad abbandonare la coalizione.

Con lo *Shapley Value* ciò non avviene in quanto è una **pre-imputazione** e distribuisce il guadagno sulla base del *cointributo marginale* dato dal singolo giocatore partecipando alla **grande coalizione**.

Però se si ha un sottoinsieme di giocatori questi guadagnerebbero di più.

Riprendendo l'esempio di prima, se partecipano tutti alla coalizione grande si avrà che A prende la metà e gli altri distribuiscono il resto.

Però se la coalizione fosse formata solo da A il partito B (con 25 voti) prenderebbe di più.

Core

Con il **Core** invece la nozione di stabilità è mantenuta.

In generale diremo che un *vettore di payoff* appartiene al **core** se:

$$\forall S \subset N, \sum_{i \in S} x_i \geq v(S)$$

Cioè un partecipante si unisce ad una coalizione solo se guadagna di più o almeno lo stesso di ciò che si aspettano.

Questo *solution concept* non sono non è *unico* ma potrebbe pure essere vuoto.

Esempio: Voting Game

$$v(\{A, B, C, D\}) = v(N) = 1$$
$$v(\{i\}) = 0 \quad \forall i \in N$$
$$v(\{B, C, D\}) = 1$$
$$v(\{A, B\}) = 1$$
$$v(\{A, C\}) = 1$$
$$v(\{A, D\}) = 1$$

$$\begin{cases} x_A + x_B + x_C + x_D = 100 \\ x_B + x_C + x_D \geq 100 \\ x_A + x_B \geq 100 \\ x_A + x_C \geq 100 \\ x_A + x_D \geq 100 \end{cases} \rightarrow \begin{cases} // \\ x_A = 0 \\ x_B = 100 \\ x_C = 100 \\ x_D = 100 \end{cases}$$

Ma ciò non è possibile di conseguenza il **Core** è vuoto

↳ è violata la prima condizione

Nucleolo

Un altro possibile approccio mira alla minimizzazione dell'*insoddisfazione* delle varie coalizioni.



Dato un vettore \overline{X} , si definisce **Excess Function**:

$$e(x, S) = v(S) - \overline{X}(S)$$

Dove $\overline{X}(S) = \sum_{i \in S} x_i$, cioè è la differenza tra quanto si voleva e quanto viene effettivamente viene effettivamente distribuito.

In particolare:

- Se $e(x, S) = 0$ allora la coalizione ha guadagnato quanto si aspettava.
- Se $e(x, S) < 0$ allora la coalizione ha guadagnato più quanto si aspettava.

- Se $e(x, S) > 0$ allora la coalizione ha guadagnato meno di quanto si aspettava.

Riprendendo l'esempio di prima sappiamo che gli Shapley Value, cioè quanto viene distribuito è il seguente:

$$\begin{aligned}\phi_A &= \frac{1}{2}, \\ \phi_B = \phi_C = \phi_D &= \frac{1}{6}\end{aligned}$$

Calcoliamo l'*insoddisfazione*:

- $e(x_\phi, A) = 0 - \frac{1}{2} * 100 = -50$
- $e(x_\phi, B) = e(x_\phi, C) = e(x_\phi, D) = 0 - \frac{1}{6} * 100 = -\frac{50}{3}$
- $e(x_\phi, \{A, C\}) = e(x_\phi, \{A, D\}) = e(x_\phi, \{A, B\}) = 100 - (\frac{1}{2} + \frac{1}{6}) * 100 = \frac{2}{3} * 50$
- $e(x_\phi, \{B, C, D\}) = 100 - (\frac{1}{6} + \frac{1}{6} + \frac{1}{6}) * 100 = 50$

In questo caso la coalizione B,C,D è la più **insoddisfatta** a seguire gli insiemi con A.

L'idea sarebbe quella di andare a trovare un valore che minimizza la *massima insoddisfazione*.

L'insoddisfazione viene indicata con y .

Avremo dunque che:

$$\frac{2}{3} * 50 < y < 50$$

Sviluppiamo il sistema:

$$\begin{cases} 100 - x_b - x_c - x_d = y [1] \\ 100 - x_a - x_c = y \\ 100 - x_a - x_b = y \\ 100 - x_a - x_d = y \end{cases}$$

Risolvendo il sistema si avrà:

$$y = \frac{200 - 3x_a}{2} [2]$$

Per l'*efficienza* si avrà:

$$x_a + x_b + x_c + x_d = 100$$

Però sappiamo che $x_b + x_c + x_d = 100 - y$ (ricavato dall'equazione [1]) quindi

$$x_a = y$$

Sostituendo all'espressione [2]:

$$y = 40$$

Dunque:

$$x_b + x_c + x_d = 60$$

Per simmetria si ottiene:

Xa	Xb	Xc	Xd
40	20	20	20

Nonostante la minimizzazione non tutti avranno lo stesso livello di insoddisfazione dunque si deve definire il **vettore delle insoddisfazioni**.



Dato $\bar{x} \in X(G)$ dove $X(G)$ è l'insieme di tutte le imputazioni del gioco, il **vettore delle insoddisfazioni** $\bar{\theta}(\bar{x})$ contiene i 2^N valori di eccesso $e(\bar{x}, S), \forall S \subset N$ ordinati in modo non crescente.

A questo punto è possibile definire il **nucleolo**



Il **nucleolo** è l'insieme delle *imputazioni* che minimizzano *lessicograficamente* il vettore degli eccessi.

In particolare esiste un teorema il quale afferma:

Se il **core** non è vuoto, allora il *nucleolo* si trova al suo interno.

Algoritmo di Calcolo del Nucleolo

A seguire i vari passi:

1. Per ogni coalizione, determina l'eccedenza, ossia la differenza tra il valore della coalizione e la somma dei pagamenti proposti ai membri.
2. Ordina le eccedenze in modo decrescente.
3. Cerca la distribuzione dei pagamenti che minimizza queste eccedenze in modo lessicografico.
4. Se necessario, risolvi il problema iterativamente, riducendo le eccedenze maggiori fino ad ottenere una distribuzione stabile.

Algoritmo per calcolare il Nucleolo
(sequenza di programmi lineari)

ϵ -CORE

LP_1

$$\begin{cases} \min \epsilon \\ x(N) = v(N) \\ x(S) \geq v(S) - \epsilon, \forall S \subseteq N \\ \epsilon \geq 0 \end{cases}$$

Calcolo l'ottimo, sia esso ϵ_1

Considero le coalizioni critiche:

$$S_i : \begin{matrix} x^*(S_i) = v(S_i) - \epsilon_1 \\ \uparrow \\ \text{sol. ott.} \end{matrix}$$

Sia F_1 l'insieme di tali coalizioni

LP_2

$$\begin{cases} \min \epsilon \\ x(N) = v(N) \\ x(S) \geq v(S) - \epsilon, \forall S \in 2^N \setminus F_1 \\ \vdots \\ x(S) = v(S) - \epsilon_1, \forall S \in F_1 \end{cases}$$

BIBLIOGRAPHY

01

SLIDE FRANCESCO SCARCELLO



02

NOTE FRANCESCO SCARCELLO



03

APPUNTI MATTEO GRECO



04

APPUNTI STEFANO MONEA



05

APPUNTI FRANCESCO COZZA



Letti, Studiati, Elaborati, Assemblati da Danilo Fortugno