Università di Perugia
Dipartimento di Matematica e Informatica

Appunti Knowledge Representation and Automated Reasoning

Autore: Chiara Luchini

Basati su:
- Slides del Prof. Stefano Bistarelli
- Lezioni del Prof. Stefano Bistarelli

Anno Accademico 2021/2022

Indice

Elenco delle figure

Capitolo 1

Costraint Satisfaction Problems

1.0.1 Constraint Satisfaction

Un vincolo è semplicemente una relazione logica tra diverse incognite (o variabili), ognuna delle quali assume un valore in un dato dominio. Un vincolo quindi restringe i possibili valori che le variabili possono assumere ne rappresenta alcune informazioni parziali sulle variabili di interesse. Formalmente, un costraint satisfaction problem (o CSP) è definito da:

- Un insieme di **variabili** $X1, X2, \cdots, X_n$;
- Una funzione che mappa ogni variabile a un dominio finito;
- Un insieme di **vincoli** $C1, C2, \cdots, C_m$;
- Un insieme D_i non vuoto di possibili valori per ogni variabile X_i.

Ogni vincolo C_i coinvolge alcuni sottoinsiemi di variabili e specifica le combinazioni di valori consentite per quel sottoinsieme. Uno **stato** del problema è definito da un'**assegnazione di valori** ad alcune o a tutte le variabili $\{X_i = v_i, X_j = v_j, \cdots\}$. Un'assegnazione che non viola alcun vincolo è chiamata assegnazione **coerente o legale**. Un'**assegnazione completa** è quella in cui viene menzionata ogni variabile, e una **soluzione** a un CSP è un'assegnazione completa che soddisfa tutti i vincoli. Alcuni CSP richiedono anche una soluzione che massimizzi una **funzione obiettivo**. Ciascun vincolo limita la combinazione di valori che un insieme di variabili può assumere contemporaneamente. Una soluzione di un CSP è l'assegnazione a ciascuna variabile di un valore dal suo dominio che soddisfi tutti i vincoli. Il compito è trovare una soluzione o tutte le soluzioni. Pertanto, il CSP è un problema combinatorio che può essere risolto mediante la ricerca.

1.0.2 Constraint Solving

La risoluzione dei vincoli differisce dalla soddisfazione dei vincoli poiché utilizza variabili con domini infiniti come i numeri reali. Inoltre, i singoli vincoli sono più complicati, ad esempio non lineari, uguaglianze...

1.0.3 Esempio: Map-Colouring

Lavoriamo in una mappa dell'Australia che mostra ciascuno dei suoi stati e territori e il compito ci viene affidato è di colorare ogni regione di rosso, verde o blu in modo tale che le regioni vicine non hanno lo stesso colore. Per formulare questo come un CSP, definiamo le variabili come le regioni: WA, NT, Q, NSW , V , SA e T. Il dominio di ciascuna variabile è l'insieme $\{rosso, verde, blu\}$. I vincoli richiedono che le regioni vicine abbiano colori distinti; ad esempio, le combinazioni consentite per WA e NT sono le coppie

$$\{(rosso, verde), (rosso, blu), (verde, rosso), (verde, blu), (blu, rosso), (blu, verde)\}$$

Ci sono molte soluzioni possibili, come $\{WA = rosso, AND = verde, A = rosso, NSW = verde, V = rosso, SA = blu, T = rosso\}$.

È utile visualizzare un CSP come **grafico di vincoli**. I nodi del grafico corrispondono a variabili del problema e gli archi corrispondono a vincoli.

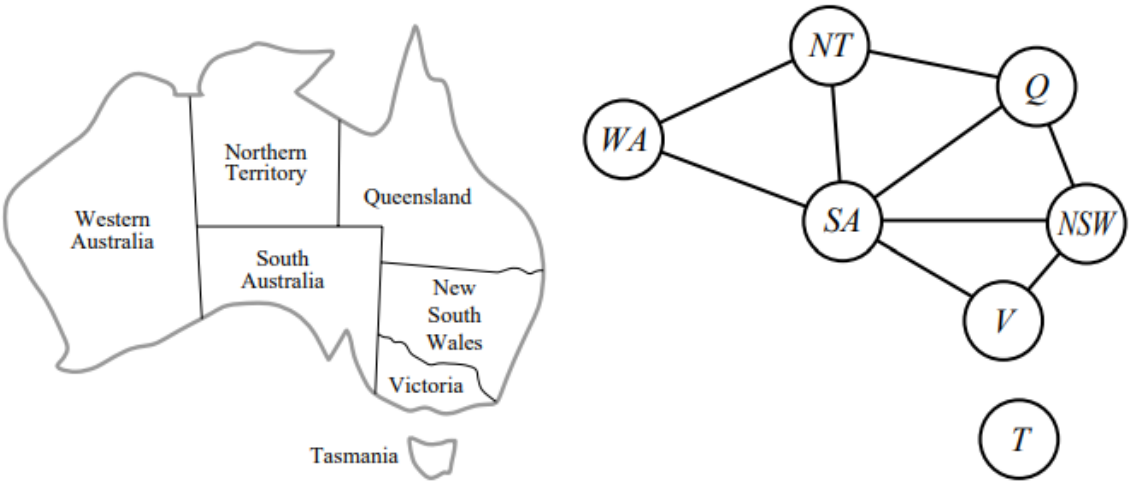


Figura 1.1: Map-colouring

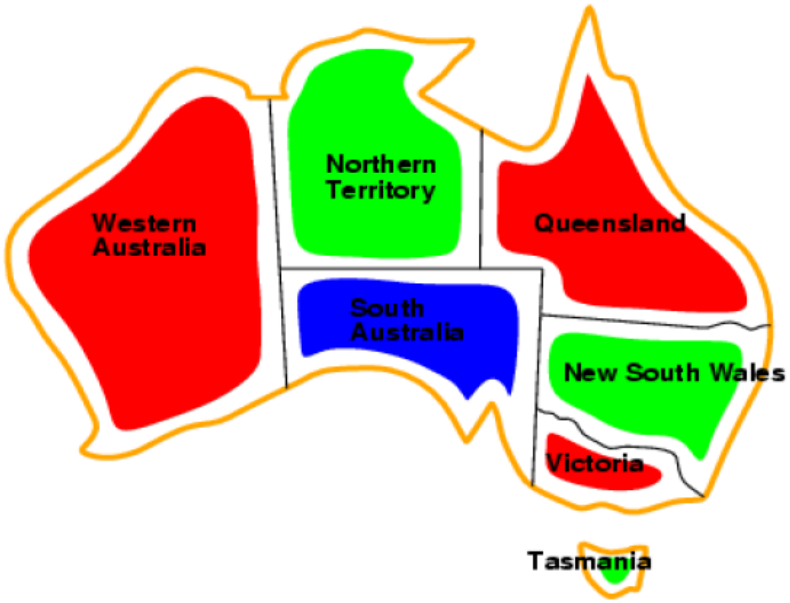


Figura 1.2: Soluzione Map-colouring

In un CSP binario ogni vincolo è in relazione con due variabile. Il tipo più semplice di CSP coinvolge variabili che sono discrete e hanno domini finiti, i problemi di colorazione della mappa sono di questo tipo.

Tipi di variabili. Le variabili discrete possono avere domini:

- **finiti**: grandezza dell'assegnamento completo $d \longrightarrow O(d^n)$;
- **infiniti**:
  - hanno bisogno di un linguaggio di vincoli
  - i vincoli lineari sono risolvibili, i non lineari sono indecidibili;

Le variabili continue con dei vincoli lineari sono risolvibili in un tempo polinomiale dai metodi LP.

Tipi di vincoli. I vincoli possono essere:

- **unari**: coinvolgono una sola variabile;
- **binari**: coinvolgono due variabili;
- **higher-order**: coinvolgono 3 o più variabili;

- **preferenze**: es. *il rosso è meglio del verde*, spesso rappresentabili come un costo associato a ogni assegnamento di variabile.

Inoltre i vincoli possono essere espressi in maniera:

- **Implicita**: non viene direttamente indicata la relazione fra gli elementi del dominio che sono permessi. Un esempio può essere x<y, dove non si elencano tutti i possibili assegnamenti delle variabili che non violano quel vincolo ma si possono calcolare;
- **Esplicita**: si elencano tutti i valori ammessi per le variabili coinvolte nel vincolo. Nell'esempio di prima si avranno tutte le coppie di valori ammessi in base a quel vincolo.

1.0.4 Caratteristiche CSP

Alcune caratteristiche di questi problemi sono:

- Caso speciale di un problema di ricerca;
- I domini possono essere discreti o continui;
- Commutatività: l'ordine in cui applichiamo le azioni non ha effetto: ad ogni nodo, considera solo le assegnazioni ad una singola variabile;
- Durante la ricerca, una volta violato un vincolo, rimane tale (monotonicità): fermati e torna indietro non appena un vincolo viene violato;
- L'ordine in cui scegliamo le variabili e i loro valori fa una grande differenza: abbiamo bisogno di un'euristica intelligente;
- Il test dell'obiettivo è scomposto in un insieme di vincoli sulle variabili, piuttosto che in una singola scatola nera;
- Quando gli insiemi di variabili sono indipendenti (nessun vincolo tra di loro) il problema è scomponibile in sottoproblemi che possono essere risolti indipendentemente;
- A ogni passaggio dobbiamo verificare la coerenza. Abbiamo bisogno di metodi di propagazione dei vincoli.

1.0.5 Standard search formulation

Gli stati sono definiti dal valore assegnato finora:

- **Stato iniziale**: assegnamento vuoto {};
- **Funzione successore**: assegna un valore a una variabile non assegnata che non va in conflitto con l'assegnamento corrente;
- **Goal test**: l'assegnamento corrente è completo.

Questa formula viene usata per tutti i CSP e ogni soluzione appare a profondità n con n variabili. Il cammino è irrilevante, così che può usare la formulazione complete-state.

1.1 Metodi di ricerca sistematici

1.1.1 Generate & test

Probabilmente il metodo di risoluzione dei problemi più generale. L'algoritmo consiste nei seguenti due passaggi che si ripetono:

1. si generano le etichette;
2. si controlla se vanno bene gli assegnamenti.

Alcuni possibili miglioramenti sono ad esempio uno **smart generator**, ovvero si assegnano i valori alle variabili e se non vanno bene si effettuano dei cambiamenti sugli assegnamenti errati (ricerca locale). Un altro miglioramento consiste nel fare il test sugli assegnamenti e poi fare backtracking.

1.1.2 Backtraking

Supponiamo di avere un problema CSP con 4 variabili (A,B,C,D), supponiamo che i vincoli siano

$$A = D, B \neq D, A + C < 4$$

L'algoritmo consiste in:

1. assegna il valore alla variabile;
2. si controlla la consistenza;
3. finché tutte le variabili sono etichettate.

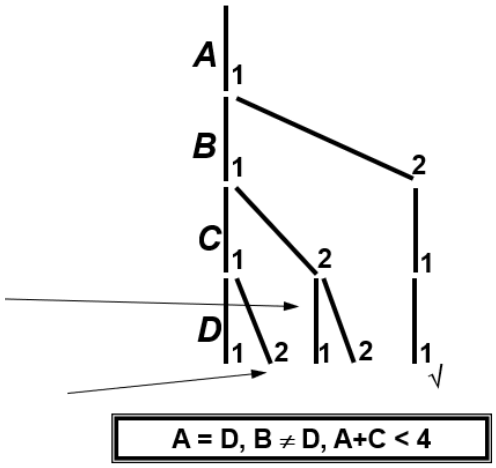


Figura 1.3: Esempio backtracking.

Se tutte le variabili non sono etichettate si torna al passo 1. L'assegnamento delle variabili è commutativo ad esempio

[WA = red allora NT = green] come [NT=green allora WA =red]

Si devono solo considerare gli assegnamenti alle singole variabili per ogni nodo $\longrightarrow b = d$ e ci sono d^n foglie. La **DFS** per i CSP son gli assegnamenti a variabili singole è chiamata ricerca **backtracking**, essa è l'algoritmo base uniformed per i CSP.

- Problem:

  X::{1,2}, Y::{1,2}, Z::{1,2}

  X = Y, X ≠ Z, Y > Z

generate & test

X	Y	Z	test
1	1	1	fail
1	1	2	fail
1	2	1	fail
1	2	2	fail
2	1	1	fail
2	1	2	fail
2	2	1	passed

backtracking

X	Y	Z	test
1	1	1	fail
		2	fail
	2		fail
2	1		fail
	2	1	passed

Figura 1.4: Esempio generate and test vs backtracking.

Ricerca backtracking: incrementale

La strategia di ricerca è la **DFS (Depth First Search):**

1. scegli una variabile non istanziata, scegli un valore dal suo dominio, controlla se qualche vincolo è violato;
2. se nessun vincolo viene violato, continua la ricerca in modo ricorsivo;
3. altrimenti, torna indietro: torna alla decisione precedente e fai un'altra scelta.

In termini di grandezza dell'albero di ricerca il numero di foglie è pari a d^n, dove n è il numero di variabili e d = $max|D_i|$.

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING([], csp)

function RECURSIVE-BACKTRACKING(assigned, csp) returns solution/failure
    if assigned is complete then return assigned
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assigned, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assigned, csp) do
        if value is consistent with assigned according to CONSTRAINTS[csp] then
            result ← RECURSIVE-BACKTRACKING([var = value|assigned], csp)
            if result ≠ failure then return result
    end
    return failure
```

Figura 1.5: Algoritmo di backtracking.

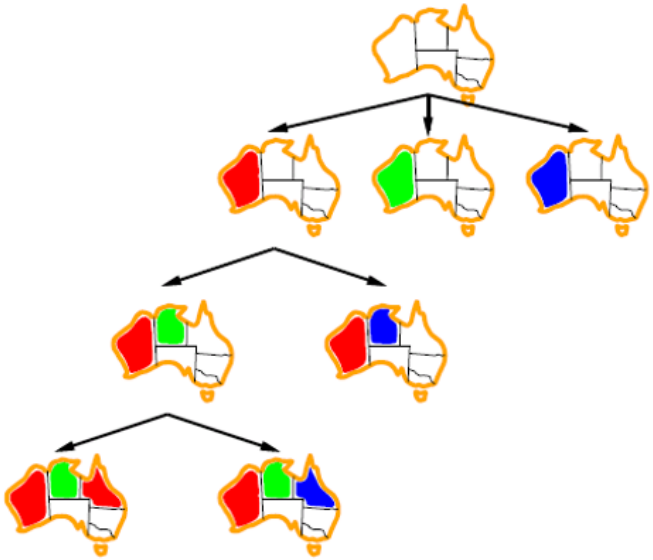


Figura 1.6: Esempio backtracking.

1.1.3 Miglioramenti dell'efficienza del backtracking

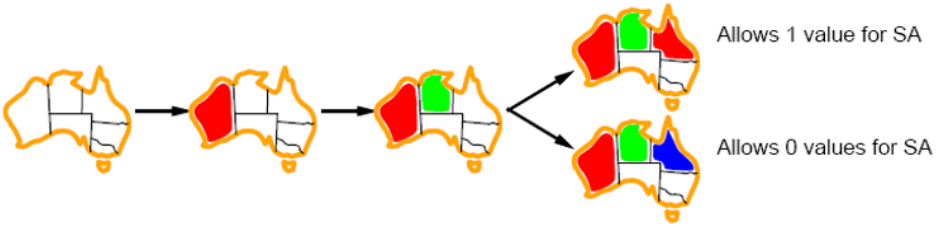
Per impostazione predefinita, SELECT-UNASSIGNED-VARIABLE seleziona semplicemente la successiva variabile non assegnata nell'ordine dato dalla lista VARIABLES[csp]. Questo ordinamento di variabili statiche raramente si traduce nella ricerca più efficiente. Ad esempio, dopo le assegnazioni per WA = rosso e NT = verde, c'è un solo valore possibile per SA, quindi ha senso assegnare SA = blue next piuttosto che assegnare Q. Infatti, dopo l'assegnazione di SA, le scelte per Q, NSW e V sono tutte forzate.

Variabile più vincolata In questo caso si va a scegliere fra le variabili disponibili per l'assegnamento quella più vincolata in base ad alcune caratteristiche:

- si sceglie la variabile con il **minor numero di valori legali** (minimum remaining values -MRV). È stata anche chiamata la "variabile più vincolata" o euristica "fail-first", quest'ultima perché seleziona una variabile che ha maggiori probabilità di causare un errore presto, potando così l'albero di ricerca.

- si sceglie la variabile con **più vincoli possibili** (degree heuristic) sulle variabili rimanenti.

- data una variabile, si sceglie **il valore meno vincolante** (least-constraining-value), quello che esclude il minor numero di valori nelle restanti variabili.



1.2 Propagazione delle informazioni attraverso vincoli

Finora il nostro algoritmo di ricerca considera i vincoli su una variabile solo nel momento in cui il variabile viene scelta da SELECT-UNASSIGNED-VARIABLE. Ma guardando alcuni dei vincoli all'inizio della ricerca, o anche prima dell'inizio della ricerca, possiamo drasticamente ridurre lo spazio di ricerca.

1.2.1 Forward Checking

Un modo per fare un uso migliore dei vincoli durante la ricerca è chiamato **forward checking**(controllo in avanti). Ogni volta che viene assegnata una variabile X, il processo di forward checking esamina ogni variabile non assegnata Y che è connessa a X da un vincolo e cancella dal dominio di Y qualsiasi valore non coerente con il valore scelto per X.

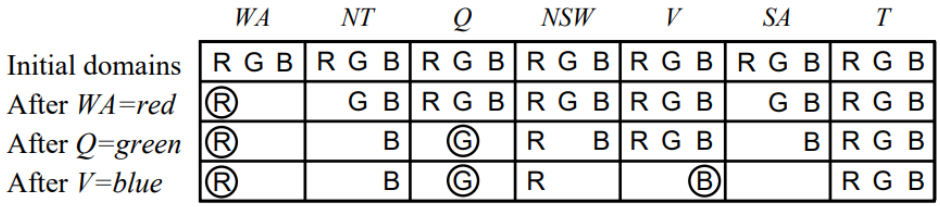


Figura 1.7: Forward checking applicata a Map-colouring problem.

Ci sono due punti importanti da notare su questo esempio. Innanzitutto, si noti che dopo aver assegnato WA = rosso e Q = verde, i domini di NT e SA sono ridotti ad un unico valore; abbiamo eliminato del tutto la ramificazione su queste variabili di propagare le informazioni da WA e Q. L'euristica MRV, che è un partner ovvio per il controllo in avanti, selezionerebbe automaticamente SA e NT successivamente.

1.2.2 Constraint propagation

Sebbene il forward checking rilevi molte incoerenze, non le rileva tutte. Per esempio, consideriamo la terza riga della Figura 1.7. Essa mostra che quando WA è rosso e Q è verde, sia NT che SA sono costretti a essere blu ma questo non è possibile perchè sono due zone vicine. Il forward checking non rileva questo come un'incoerenza, perché non guarda abbastanza avanti. La propagazione del vincolo (constraint propagation) è il termine generale per propagare le implicazioni di un vincolo su una variabile su altre variabili; in questo caso dobbiamo propagare da WA e Q su NT e SA, e quindi sul vincolo tra NT e SA per rilevare l'incoerenza.

Domain Consistency

L'idea di **arc consistency** fornisce un metodo veloce di propagazione dei vincoli sostanzialmente più forte del forward checking. Una variabile è **consistente al dominio** (domain consistent) se nessun valore del dominio del nodo è è dichiarato impossibile da uno qualsiasi dei vincoli. L'idea è di "potare" il più possibile prima di selezionare un valore per le variabili. La domain consistency è stata definita solo per i vincoli che coinvolgono una sola variabile. Quando i vincoli sono binari, possiamo usare gli archi per indicare che un vincolo vale tra una coppia di variabili:

- un **nodo** per ogni variabile;
- un **arco** per ogni vincolo.

Node Consistency. In questo caso tutti i vincoli sono unari, le variabili sono rappresentate da dei **vertici**. Il vertice è **node consistent** se ogni valore nel dominio delle variabili soddisfa tutti i vincoli unari imposti sulla variabile X. Spesso vengono rappresentati come un cappio o arco che ritorna sullo stesso nodo.

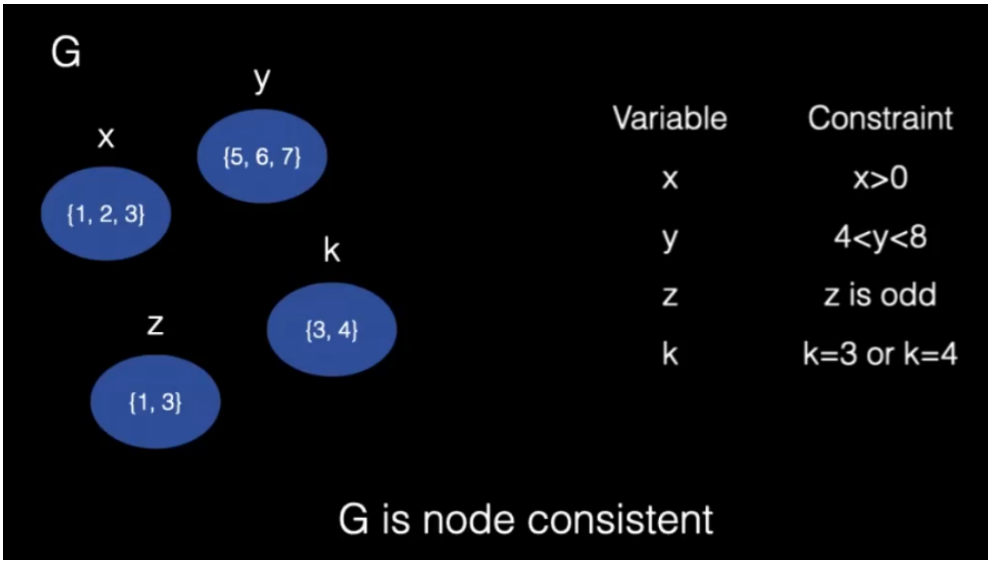


Figura 1.8: Esempio grafo node consistent.

Arc Consistency. Quando tutti i vincoli sono binari si parla di **arc consistency**. Un arco (u,c) è consistente se per ogni valore x del dominio dom(u) esiste un valore y nel dom(v) tale che un assegnamento u=x e v=y soddisfa tutti i vincoli binari che coinvolgono sia u che v.

Arc Consistency Algorihtm. Per fare diventare un arco (u,v) consistente si cancellano tutti i valori x dal dom(u) che sono incosistenti con tutti i valori in dom(v). Restituisce true se è stata fatta una modifica al dominio di u.

```
procedure REVISE((u,v))
  DELETED <-  false
    for each x in dom(u) do
      if there is no such y in dom(v) such that (x, y) is consistent then
        delete X from dom(u)
        DELETED <- true
      end if
    end for
  return DELETED
end REVISE
```

Figura 1.9: Procedura REVISE.

AC-1 Un singolo passo dell'agoritmo REVISE non è sufficiente. L'algoritmo base per arc consistecy è AC-1, il quale esegue l'algoritmo REVISE finchè il dominio delle variabili cambia. In questo si ripete la procedura REVISE ogni volta che viene modificato un valore in un dominio.

```
procedure AC-1(G)
  repeat
    CHANGED  false
      for each arc (u,v) in G do
        CHANGED <- REVISE((u,v)) or CHANGED
      end for
  until not(CHANGED)
end AC-1
```

Figura 1.10: Algoritmo AC-1.

Una sola revisione riuscita di un arco su una particolare iterazione causa la revisione di tutti gli archi nella prossima iterazione anche se gli archi non sono influenzati dal cambiamento.

AC-2 AC-2 è un algoritmo che può fare arc consistecy in un solo passo attraverso i nodi. Il risultato è ottenuto passando per i nodi in un ordine numerico:

- allegare ad un nodo tutti i valori che non sono in conflitto con i nodi precedentemente assegnati;
- Guardando i vicini di questo nodo che sono stati già valutati; se un valore non ha un'assegnazione corrispondente per lo stesso arco, eliminalo;

- Ogni volta che qualsiasi valore è cancellato da un arco, guarda ai suoi vicini a sua volta, e si controlla se un loro valore può essere eliminato. Se può essere eliminato, si continua il processo iterativamente finchè non ci sono più cambiamenti che possono essere fatti. Poi si prosegue con gli altri archi.

In sostanza si sceglie un ordine fra i nodi, prendiamo ad esempio y come primo e controlliamo tutti i vincoli fra y e k se c'è qualche valore di dom(y) che va in conflitto allora si elimina. Stessa cosa si per l'arco (x,y) se si fa qualche modifica si rimette in coda l'arco in modo da controllare se ci sono altre modifiche da fare. Se un valore b del nodo i è rimosso allora si aggiunge tutti (k,i) alla coda Q, per il controllo degli archi.

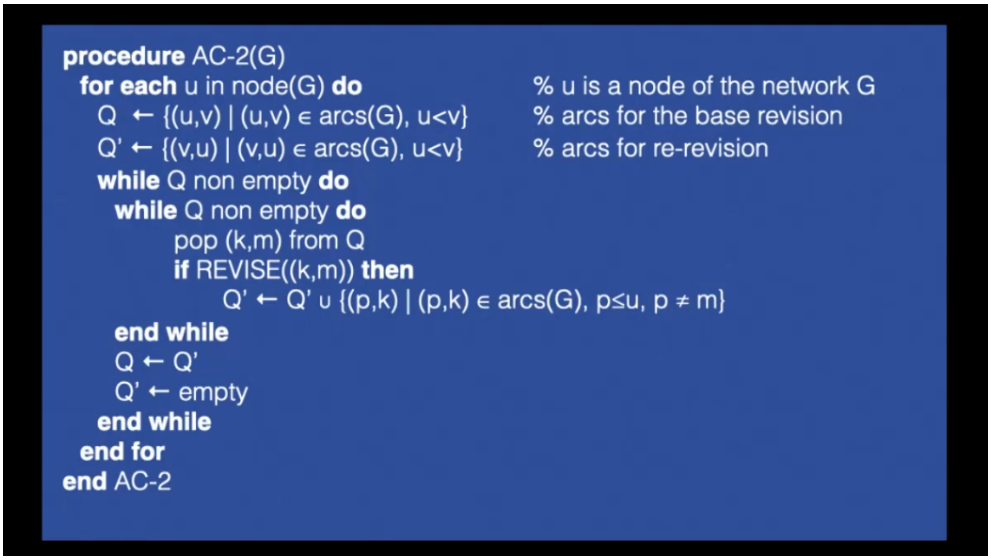


Figura 1.11: Algoritmo AC-2.

AC-3 AC-3 è un miglioramento di AC-2, alcuni di essi possono essere già essere nella coda Q. Se è così allora non dovrebbero essere inseriti di nuovo.

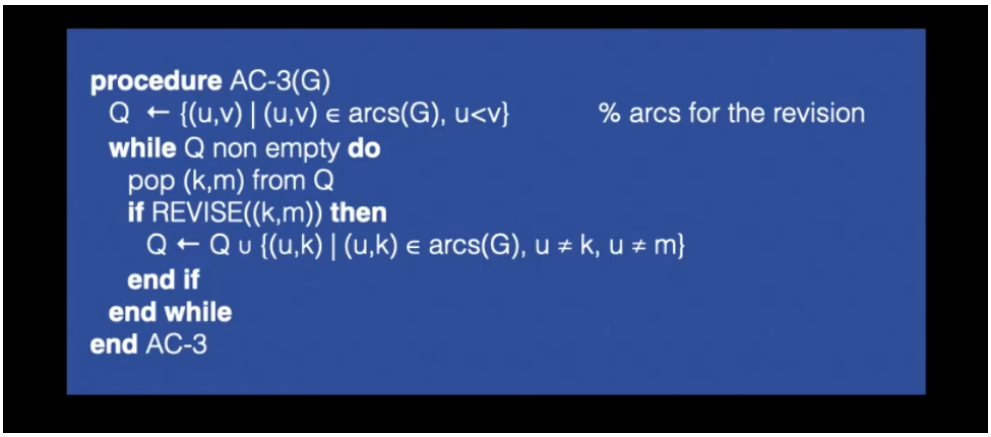


Figura 1.12: Algoritmo AC-3.

In sostanza si prende un arco dalla coda Q, si fa la REVISE, se faccio una modifica allora si aggiunge alla coda Q tutti gli archi (u,k) che non sono stati già controllati ovvero $u \neq k$ e $u \neq m$.

1.3 Soft Constraint Satisfaction Problems

I vincoli che abbiamo visto in precedenza sono dei vincoli **assoluti**, la cui violazione esclude una possibile soluzione. Molti dei problemi CSP reali includono i vincoli **preference** i quali indicano quali soluzioni sono preferite. Per esempio in un problema di timetable in università ci potrebbe essere il professore X che preferisce insegnare la mattina mentre il professore Y preferisce insegnare il pomeriggio. Un timetable dove il prof X insegna alle 14 e il prof Y alle 9 potrebbe essere una soluzione, ma non è quella ottimale viste le preferenze. I vincoli sulle preferenze possono essere codificati spesso come dei **costi** applicati sugli assegnamenti individuali delle variabili. Riprendendo l'esempio di prima possiamo dare all'assegnamento prof X = (lezione alle 14) un costo di 2, mentre all'assegnamento prof X = (lezione alle 9) un costo di 1. In questo modo si cerca fra le possibili soluzioni quella ottimale andando a minimizzare (o massimizzare...) il costo della soluzione.
Supponiamo di avere un problema di colorazione del grafo e cerchiamo di trovare una soluzione ottimale.

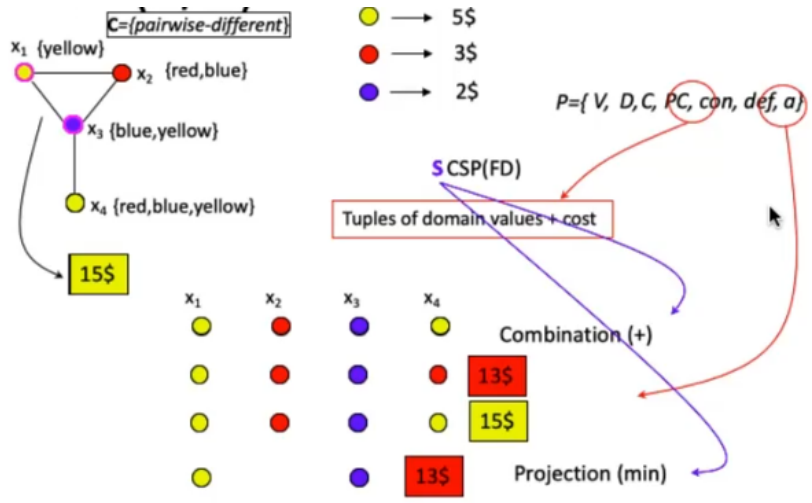


Figura 1.13: Esempio di SCSP su grafo.

Partiamo riprendendo la definizione di un CSP, esso è definito come P={V,D,C,PC,con,def,a} i quali indicano:

- V= insieme delle variabili;
- D= insieme dei domini associati alle variabili;
- C= insieme di vincoli, definito come l'associazione variabile-vincolo ovvero quali variabili sono coinvolte in quale vincolo;
- PC= sono i vincoli primitivi e variano in base al vincolo e al dominio. A seconda del tipo di vincolo che possiamo avere i primitivi possono essere diversi, esempio se lavoro sugli interi esso conterrà vincoli con gli operatori <,>,= . Sostanzialmente esso indica i tipi di vincoli che posso usare se lavoro su un dominio specifico. Inoltre indicano se il vincolo è implicito (tutti i colori diversi) o esplicito (valgono le coppie <r,g,b>, <g,r,b>, ...).
- con= funzione che definisce quali variabili sono coinvolte in quale vincolo, essa dato un vincolo restituisce le variabili che sono connesse a questo;
- def= funzione che indica quali sono i valori del dominio possibili per una specifica variabile. In un SCSP questa funzione oltre a dire i valori possibili per una variabile deve indicare anche il costo associato a quei valori, nell'esempio in Figura 1.13 se passiamo in input alla funzione def la variabile x_1 essa ci ritornerà

come valori possibile il colore giallo con costo 5$. Questa è la differenza con la funzione def di un problema CSP.

- a= sottoinsieme di V contenente tutte le variabili interessate nella soluzione del SCSP, ad esempio nel caso del grafo abbiamo che le variabili che ci interessano per la soluzione ottimale sono solo x_1 e x_3, non tutte quante.

Nell'esempio in Figura 1.13 i vincoli binari sono hard, perché la soluzione richiede per forza che i nodi collegati abbiamo colori diversi sennò si violano i vincoli, mentre i vincoli unari (quelli del costo sul colore) sono soft. Nel caso di un CSP quando utilizziamo un algoritmo di ricerca per trovare una soluzione esso si può fermare alla prima soluzione che trova oppure, se richiesto, le cerca tutte quante. Nel caso di SCSP siamo costretti a trovare tutte le possibili soluzioni in modo da scegliere la migliore. Normalmente si utilizzano due operazioni per trovare la soluzione ottimale in un SCSP:

- **Combinazione** $(+)$: dove metto insieme tutti gli assegnamenti, combinando i vincoli, quindi si calcola anche il costo totale ad esempio con l'operazione di somma;
- **Proiezione** $(\times)$: operazione di scelta della soluzione migliore data dalla combinazione in base al minimo o al massimo del costo.

1.3.1 Semiring

Un semiring <A,+,×,0,1> è una struttura costituita dai seguenti simboli:

- **A**: l'insieme degli elementi che mi rappresentano i costi, quindi il dominio dei costi, esso può essere l'insieme dei reali oppure un intervallo specifico [0,1]
- +: operatore di proiezione, usato per fare la scelta fra le soluzioni trovate dalla combinazione, può essere il minimo o massimo. Possiamo definire alcune proprietà:
  - **idempotente**: se faccio a+a dove +=minimo il risultato è sempre a, quando un operatore è idempotente è possibile definire un ordinamento ovvero

$$a \leq b \text{ b è meglio di a} \iff a + b = b$$

- $\times$: operatore di combinazione, utilizzato per combinare i vincoli, può essere somma,moltiplicazione etc... dipende dal problema. Possiamo definire alcune proprietà:
  - **commutativa**: si considera il set di vincoli invece delle tuple.
- **0**: rappresenta il valore minimo (peggiore) dell'insieme A, ovvero il bottom sotto il quale non si può andare, per l'intervallo [0,1] è 0;
- **1**: rappresenta il valore massimo (migliore) di A, il top, per l'intervallo [0,1] è 1.

Tutti questi che abbiamo visto sono simboli quindi quando si utilizza l'operatore + non si ci si riferisce alla somma ma a un operatore per la proiezione che potrebbe essere minimo o massimo o or.

Differenti tipi di semiring

Esistono diversi tipi di semiring:

- **Probabilistico**: $<\Re^+$, min, +, $+\infty$, 0$>$ si minimizza la probabilità;
- **Weighted**: <[0,1],max,x,0,1> massimo il costo dato dalla combinazione, dove si fa il prodotto;
- **Fuzzy**: <[0,1],max,min,0,1>;
- **Classical**: <{false,true}, $\vee$, $\wedge$, false, true>

1.3.2 Local consistency

Cerchiamo ora di vedere se è possibile applicare una tecnica di constraint propagation come arc consistency ai problemi SCSP in modo da rendere più efficiente la ricerca della soluzione.

Teorema dell'estensività Se tolgo dei vincoli da un problema SCSP, la soluzione migliore perchè si moltiplica di meno (meno calcoli da fare),se li aggiungo la soluzione peggiora. Togliere vincoli vuol dire **rilassare** il problema.
Se l'operatore x è idempotente allora possiamo applicare arc consistency, un esempio di operatore idempotente è il minimo se facciamo

$$1 \times 1 = 2 \text{ dove } \times = \text{somma}$$

$$1 \times 1 = 1 \text{ se } \times = \text{minimo}$$

Nei Semiring Fuzzy è possibile applicare arc consistency perchè $\times = \min$. Vediamo un esempio.

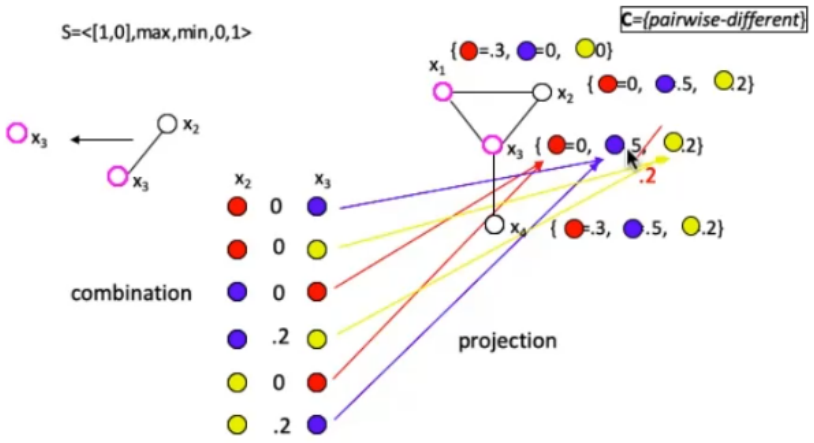


Figura 1.14: SCSP arc-consistency.

Nel caso dei SCSP quando si fa arc-consistency non si va a togliere elementi dal dominio ma si vanno a modificare i costi associati a quell'assegnamento. Ad esempio per il vincolo (x_3, x_2) andiamo a combinare tutti i vincoli e riportiamo tutte le coppie con i relativi costi, in questo caso si prende il minimo fra i due valori. E poi si proietta su x_3 i valori che abbiamo trovato e andiamo a modificare i costi associati ai colori, per $x_3 =$ rosso e $x_3 =$ giallo non cambiano perchè il $max_{rosso}(0, 0) = 0$ e $max_{giallo}(0, 0.2) = 0.2$ mentre per il blu abbiamo $max_{blue}(0, 0.2) = 0.2$ quindi $x_3 = 0.5 \rightarrow 0.2$.
DA COMPLETARE

Capitolo 2

Argumentation Framework

Argumentation Framework. Un argumentation framework (AF) è una coppia (A,R) dove

- A è un set di argomentazioni
- $R \subseteq A \times A$ è una relazione rappresentante gli "attacchi" ("sconfitte")

F({a,b,c,d,e}, {(a,b),(c,b),(c,d),(d,c),(d,e),(e,e)})

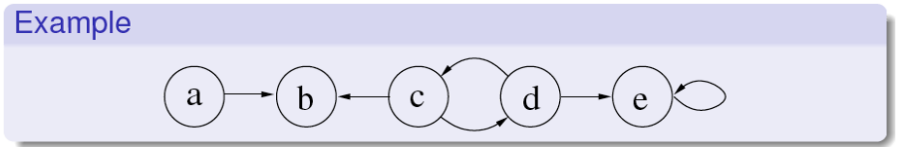


Figura 2.1: Argumentation framework.

Si ha un attacco quando si ha un'espressione logica (frase, dato...) che è in contraddizione con un'altra. Gli attacchi possono essere anche pesati, essi possono dipendere anche da chi ha detto quella frase.

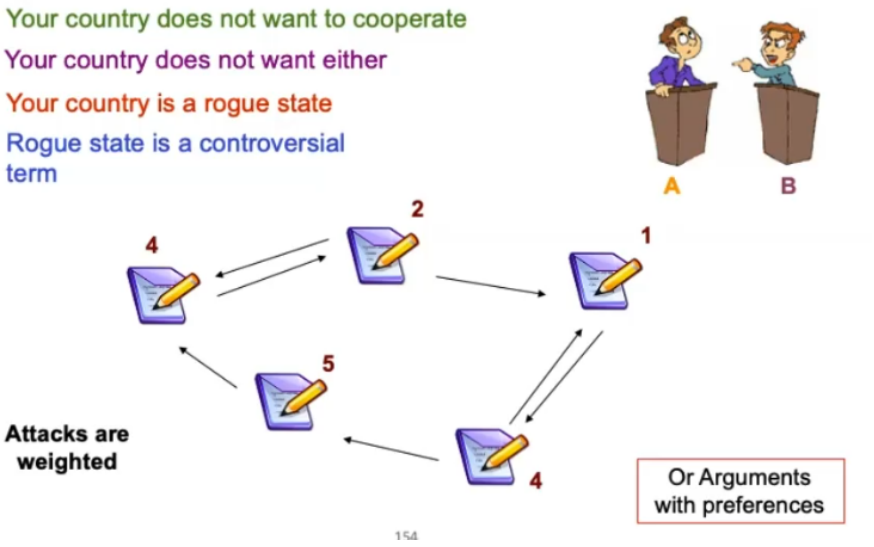


Figura 2.2: Esempio di argumentation framework.

Ci possono essere anche casi in cui è noto chi dice l'argomento e altri in cui non lo è. Per quest'ultima vogliamo selezionare gli argomenti che sono più validi rispetto agli altri, ad esempio in Figura 2.2 il 4 e il 2 sembrano buoni argomenti perchè attaccano gli altri e contrattaccano nel caso siano attaccati. Lo scopo è di definire dei criteri per trovare gli argomenti più forti, validi (che stanno "in piedi da soli"), in modo da selezione i conflitti che riescono a sopportare gli attacchi dall'esterno.

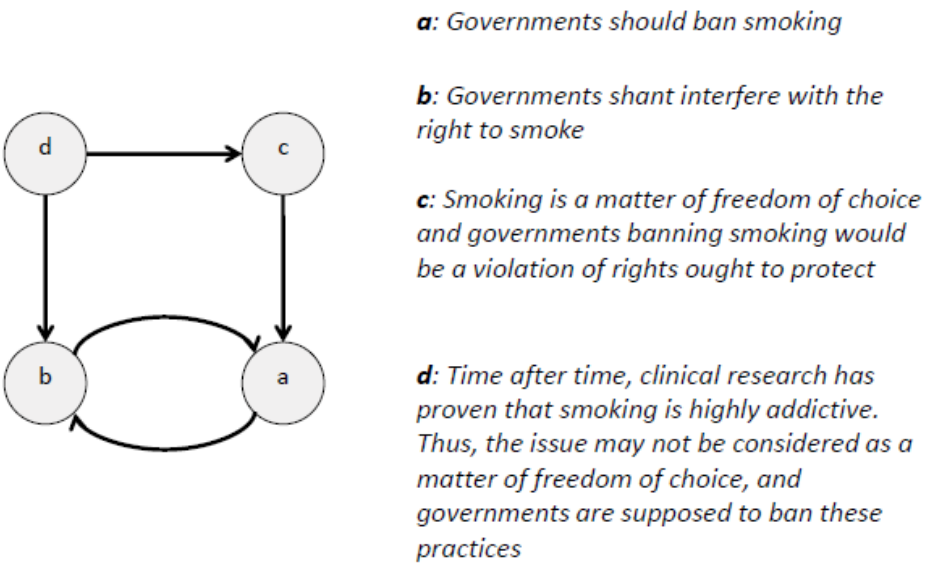


Figura 2.3: Altro esempio di argumentation framework.

Dobbiamo trovare gli argomenti che "stanno bene insieme", la prima nozione di questo tipo è un insieme di argomenti senza conflitti.

2.0.1 Extension-Based Semantics

Queste solo le semantiche (criteri) che vanno a studiare i sottoinsiemi di argomenti per stabilire l'accettabilità degli stessi, ovvero se un argomento è un'estensione accettabile altrimenti viene rigettato.

Conflit-free extensions. Dato un AF. F=(A,R). Un insieme $S \subseteq A$ è **conflict-free** in F, se, per ogni $a, b \in S, (a, b) \notin R$.

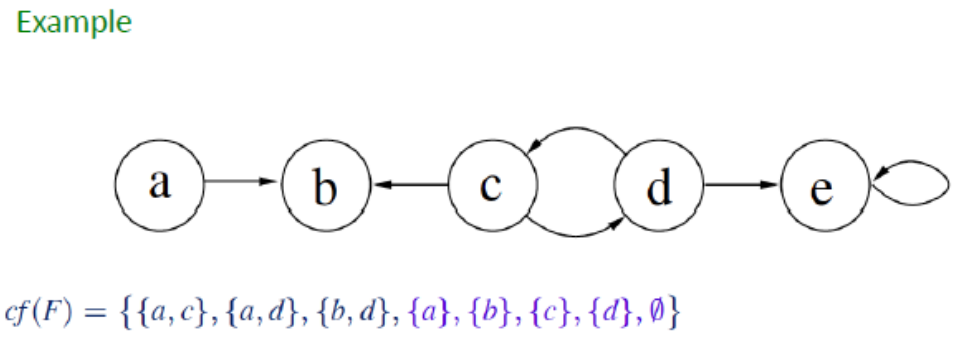


Figura 2.4: Esempio insieme conflict-free.

In questo caso andiamo a scegliere come coppie gli argomenti che non sono in conflitto (quindi che non si attaccano) fra di loro, ({a,c},{a,d} ma non {a,b}), e anche i singoli argomenti tranne e poichè esso si contraddice da solo visto che ha un cappio.

Insiemi Ammissibili (Admissible). Dato AF, F=(A,R). Un insieme $S \subseteq A$ è **ammissibile** in F, se

- S è conflict-free in F
- ogni $a \in S$ è difesa da S in F
  - $a \in A$ è difesa da S in F, se per ogni $b \in A$ con $(b, a) \in R$, esiste una $c \in S$, tale che $(c, b) \in R$

Quindi gli insiemi admissible sono quelli senza conflitti e gli argomenti si difendono.

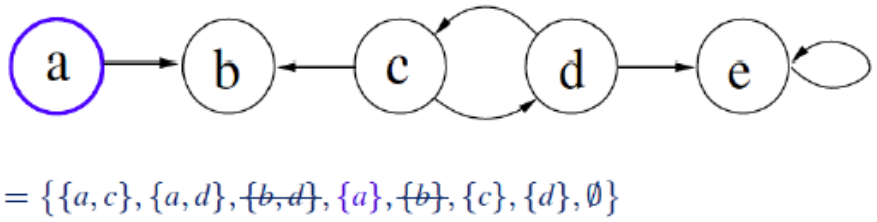


Figura 2.5: Esempio insieme ammissibile.

Non è necessario che sia lo stesso argomento a difendersi da altri attacchi, ad esempio in questo caso ,supponendo che non ci sia a, b è attaccato da c ma se prendo d lui oltre che difendere se stesso da c (poichè attaccato) difende anche b. Guardando la definizione di difesa, un argomento a è difeso da un argomento b, $(b, a) \in R$, nel momento in cui esiste un argomento c tale che c attacca b, $(c, b) \in R$. Il sottoinsieme {b,d} non viene scelto poichè d è attaccato da c ma a si difende a sua volta contrattaccando, mentre b è attaccato sia da a che c, nel primo caso nessuno lo difende nel secondo d difende b perchè attacca c.

Insieme Completo (tutti difesi). Dato un AF, F=(A,R). Un insieme $S \subseteq A$ è **completo** in F, se

- S è ammissibile in F
- ogni $a \in A$ difeso da S in F è contenuto in S
  - $a \in A$ è difeso da S in F, se per ogni $b \in A$ con $(b, a) \in R$, esiste una $c \in S$, tale che $(c, b) \in R$.

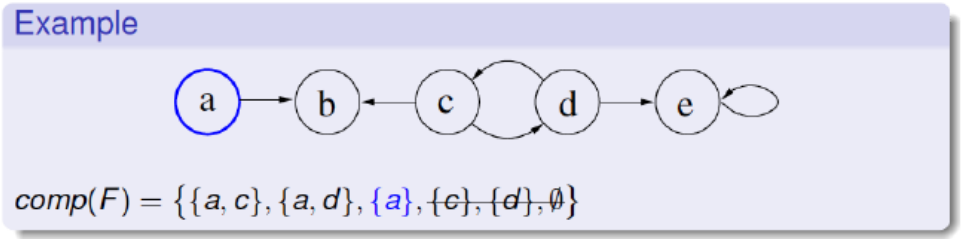


Figura 2.6: Esempio insieme completo.

Quindi un insieme complete contiene tutti gli insiemi ammissibili e anche tutti gli argomenti che sono difesi.

Grounded (scettica). Dato un AF F=(A,R). Un insieme $S \subseteq A$ è **grounded** in F, se

- S è completo in F
- per ogni $T \subseteq A$ completo in $F, T \not\subseteq S$.

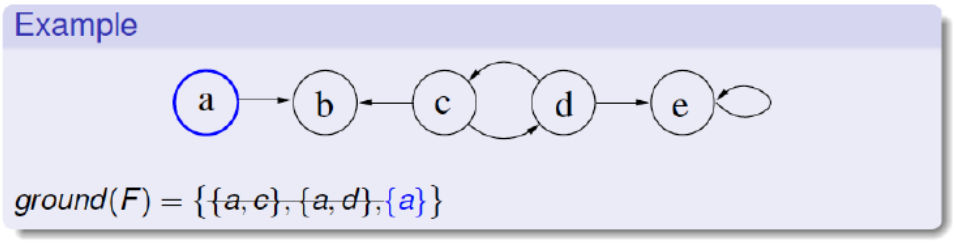


Figura 2.7: Esempio grounded.

Sceglie tra gli argomenti che sono nella complete, gli argomenti che compaiono in tutte le estensioni, è una semantica scettica perchè vuole essere sicuro che gli argomenti che sceglie provengano da una decisione prudente.

Preferred. Dato un AF, F= (A,R). Un insieme $S \subseteq A$ è **preferito** in F, se

- S è ammissibile in F
- per ogni $T \subseteq A$ ammissibile in T, $S \not\subseteq T$

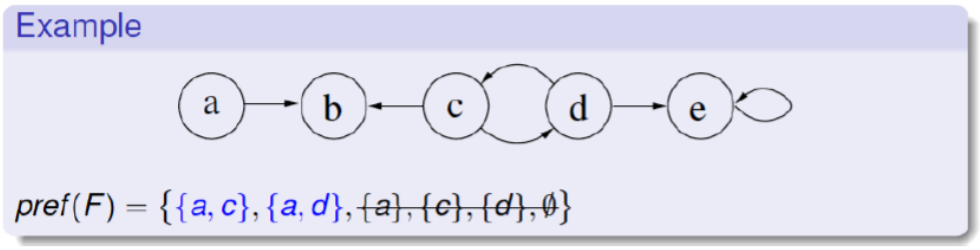


Figura 2.8: Esempio preferred.

Al contrario di grounded dove si andava a scegliere l'insieme con l'elemento in comune con gli altri insieme, in questo caso si va a scegliere tra gli insiemi ammissibili quelli che sono più grandi. Da notare che si sceglie tra gli insiemi ammissibili ma si può dimostrare che si può scegliere da quelli complete.

Stable. Dato un AF, F= (A,R). Un insieme $S \subseteq A$ è **stabile** in F, se

- S è conflict-free in F
- per ogni $a \in A \setminus S$, esiste una $b \in S$, tale che $(b, a) \in R$.

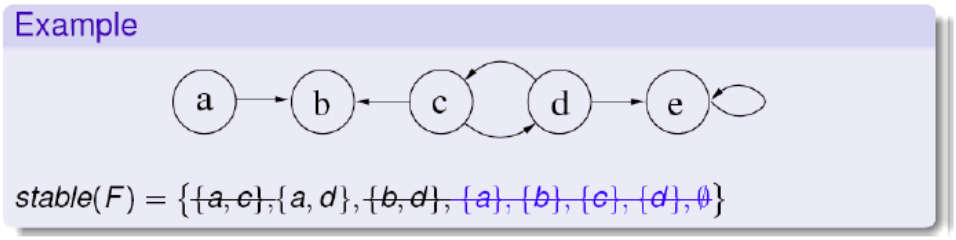


Figura 2.9: Esempio insieme stable.

In questo caso si sceglie l'insieme i quali elementi attaccano tutti gli elementi fuori dall'insieme conflict-free, ad esempio {a,c} non si prende perchè a attacca b e c attacca d e b ma nessuno dei due attacca e. Mentre nell'insieme {a,d} a attacca b e d attacca sia c che e. Esiste anche una semantica **semi-stabile** la quale nel caso cui esista un insieme stabile allora essa coincide con quest'ultimo ma quando non c'è la stabile allora sceglie tra gli insieme preffered quelli che ne attaccano di più tra gli insiemi fuori a quest'ultimo. L'obiettivo della semantica stabile è di avere cardinalità più grande possibile e di attaccare tutti gli insiemi fuori.

Complessità

La complessità è descritta da due valori:

- **Cred**: (credulous) rappresenta il costo associato all'operazione di controllo sulle semantiche, si controlla se un determinato insieme contiene un argomento passato;
- **Skept**: (scettico) controlla se un argomento è incluso in tutte le estensioni.

Manca seconda parte

2.0.2 Labelling-Based Semantics

Reinstatement Labelling

Finora abbiamo visto delle estensioni in cui gli argomenti venivano accettati (validi, quelli che si trovavano all'interno delle estensioni) o rigettati. Con le labelling semantics

manteniamo queste due sfumature e ne aggiungo un'altra ovvero gli argomenti vengono suddivisi in:

- **IN**: argomenti accettati, sono IN se sono attaccati solo da argomenti OUT;
- **OUT**: argomenti rigettati, sono OUT se sono attaccati solo da argomenti IN;
- **UNDEC**: altrimenti, tutto il resto .

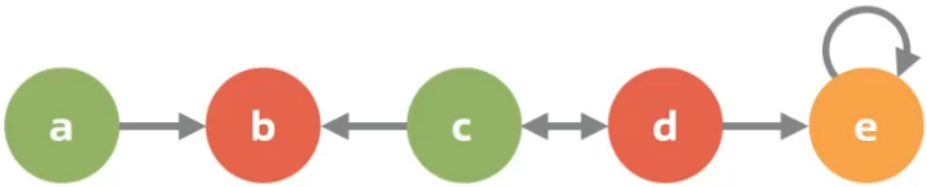


Figura 2.10: Esempio Reinstatement Labelling .

Conflict-free. Per ogni $a \in A$ si ha che:

- se a è etichettata IN allora non ha un attaccante che è IN e
- se a è etichettata OUT allora ha almeno un attaccante che è IN

La Figura 2.10 è conflict-free perchè valgono le due regole citate per tutti gli argomenti.

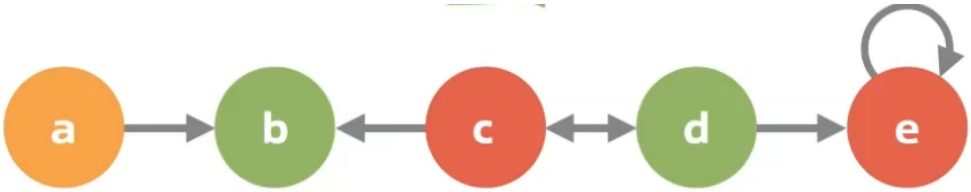


Figura 2.11: Esempio conflict-free con labelling .

Anche in questo caso abbiamo un insieme conflict-free.

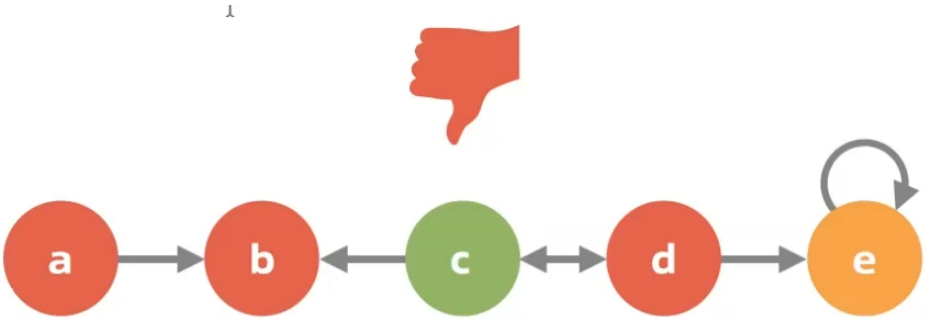


Figura 2.12: Esempio no conflict-free con labelling .

In questo ultimo esempio non abbiamo un insieme conflict-free perchè a viene considerato OUT, quindi rigettato, anche se non viene attaccato da nessuno e infatti non ci sono attaccanti IN per quest'ultimo.

Admissible. Per ogni $a \in A$ si ha che:

- se a è etichettata IN allora tutti gli attaccanti sono OUT e
- se a è etichettata OUT allora ha almeno un attaccante che è IN

L'interpretazione che possiamo dare alla prima regola è che un argomento IN è accettato come nell'insieme admissible se i suoi attaccati sono tutti sconfitti. Riprendo l'esempio in Figura 2.10 è un insieme admissible, poichè a non è attaccato da nessuno quindi la prima condizione è vera e anche per c poichè essa è attaccata solo da argomenti OUT ovvero d. Anche l'esempio in Figura 2.11 è admissible.

Completo. Per ogni $a \in A$ si ha che:

- a è etichettata IN **se e solo** se tutti gli attaccanti sono OUT e
- a è etichettata OUT **se e solo se** ha almeno un attaccante che è IN

Quesro vuol dire che se un argomento è difeso deve stare per forza dentro l'estensione. Perciò un attaccante è IN se ha solo attaccanti OUT e viceversa se un argomento ha solo attaccanti OUT allora è per forza etichettato come IN. Stessa discorso vale per la seconda regola, dobbiamo vederla come divisa in due parti la parte di sinistra e di destra al se e solo.

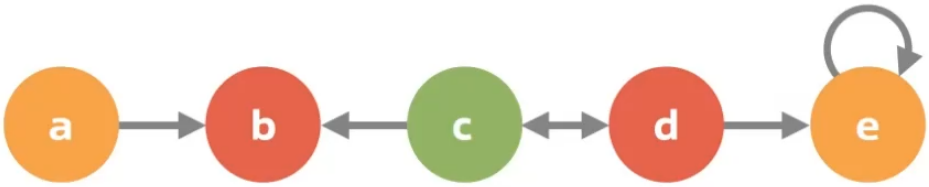


Figura 2.13: Esempio no complete con labelling .

In questo esempio non abbiamo un insieme complete perchè l'argomento a che è etichettato come UNDEC non ha attaccanti e quindi dovrebbe essere etichettato come IN.

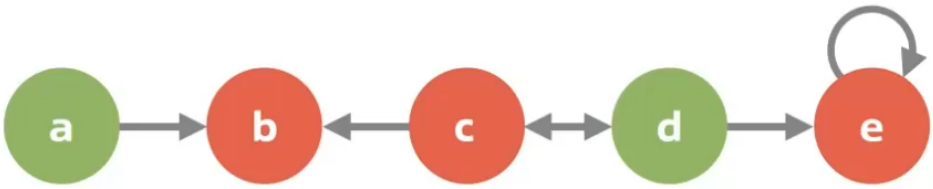


Figura 2.14: Esempio complete con labelling .

Ogni etichettatura degli argomenti verifica le regole scritte in precedenza infatti a non ha attaccanti quindi è IN, b ha un attaccante IN quindi è OUT, c ha un attaccante IN quindi è OUT, d ha tutti attaccanti OUT ovvero c e per ultimo e potrebbe essere UNDEC ma essendo che ha almeno un attaccanti IN allora è considerato OUT.

Preferred. L'etichettatura

- è un insieme complete, e
- l'insieme di argomenti IN è **massimale** fra tutte le etichette complete.

In questo caso si ragiona sulla cardinalità quindi si va a prendere il numero maggiore di argomenti che sono considerati completi. Questa semantica è preferita nel caso in cui si vogliano tirare in ballo più argomenti e sentire più opinioni.

Grounded. L'etichettatura

- è un insieme complete, e
- l'insieme di argomenti IN è **minimale** fra tutte le etichette complete.

Invece per la grounded si vanno a prendere tutti i labelling complete e si sceglie quello in cui ho meno argomenti IN.

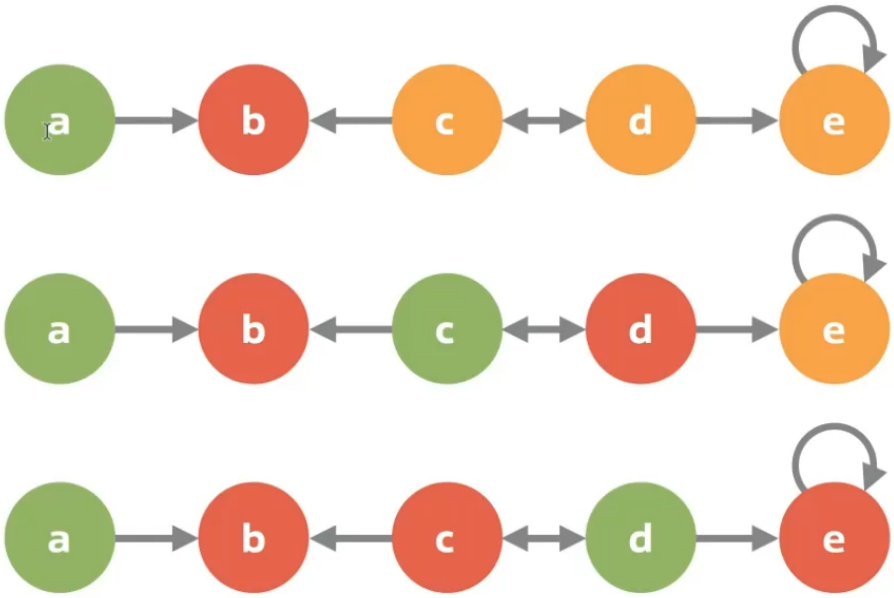


Figura 2.15: Esempi insiemi grounded e preferred .

Il primo è un esempio di insieme grounded, gli ultimi due sono preferred. Il secondo insieme è anche grounded perchè le etichette che sono state assegnate come IN sono minimali e massimali, supponiamo di mettere b e d a IN se b è IN non è complete perchè non ha attaccanti OUT stessa cosa vale per d se si etichetta come IN. Se invece assegniamo a c OUT l'insieme non è complete perchè c per essere OUT deve avere almeno un attaccante IN. Lo stesso discorso vale per l'ultimo esempio.

2.0.3 Ranking-Based Semantics

Si trasforma l'AF in un sistema di classificazione, in questo caso con le estensioni non si vanno a scegliere dei sottinsiemi di argomenti ma si a fare una classificazione di quest'ultimi. Per stabilire se un argomento è migliore di un altro si usano dei criteri per studiare la validità, ad esempio possono contare quanti attacchi diretti riceve un argomento o posso misurare la lunghezza dei path da un argomento a un altro etc...

Categorizer. Si tratta di una funzione di ranking degli argomenti, la quale assegna un valore agli argomenti controllando quali sono gli attaccanti diretti di questi.

$$Cat(x) = \begin{cases} 1 & \text{if } R_1^-(x) = 0 \\ \dfrac{1}{1 + \sum_{y \in R_1^-(x)} Cat(y)} & \text{otherwise} \end{cases}$$

Figura 2.16: Funzione categorizer .

Se un argomento non ha attaccanti $R^{-1}(x) = 0$ allora il valore è 1 sennò è dato dalla formula $\dfrac{1}{1 + \sum_{y \in R^{-1}(x)} Cat(y)}$, questo significa che se ho attaccanti forti allora l'argomento x è debole.

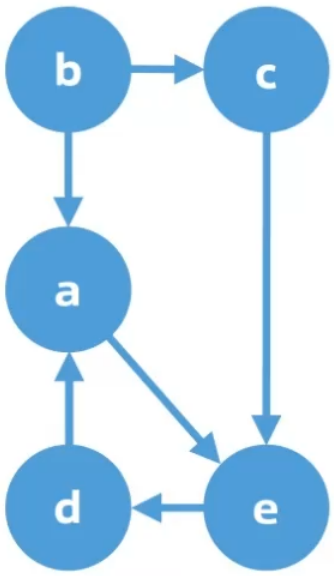


Figura 2.17: Esempio categorizer .

In questo caso Cat(b)=1 perchè non ha attaccanti e Cat(c)=0.5 perchè è attaccato da b con punteggio 1. Mentre Cat(a)=0.38, Cat(d) =0.65 e Cat(e) = 0.53 quindi la classificazione di questo insieme è

$$b \succ^{Cat} d \succ^{Cat} e \succ^{Cat} c \succ^{Cat} a$$

2.0.4 Graded semantics

I principi sono simili a quello precedente:

- più è grande il numero degli attaccanti su un argomento b, più è debole il livello di giustificazione di b
- più è grande il numero di argomenti che difendono a, più è forte il livello di giustificazione di a.

Graded defense

In questo caso andiamo a ricercare delle partizioni degli argomenti con $d_n^m(X)$, il quale è un insieme di argomenti che non hanno almeno m attaccanti che non sono contro attaccati da almeno n argomenti in X. Questo vuol dire che un argomento $a \in d_n^m(X)$ se non ci sono almeno m attaccanti che sono difesi da almeno n argomenti.

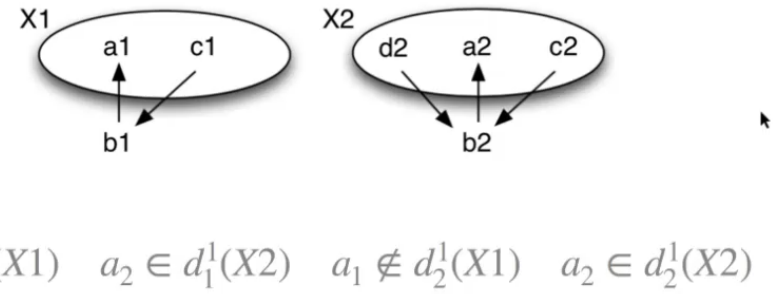


Figura 2.18: Esempio graded defense .

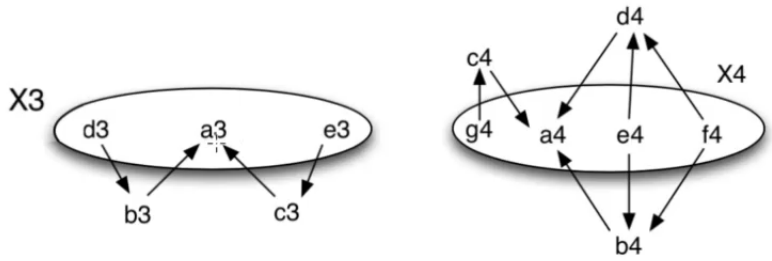
Ranking functions. Le regole sono:

- meno attaccanti ho meglio è
- più difensori ho meglio è

Scritto in formula

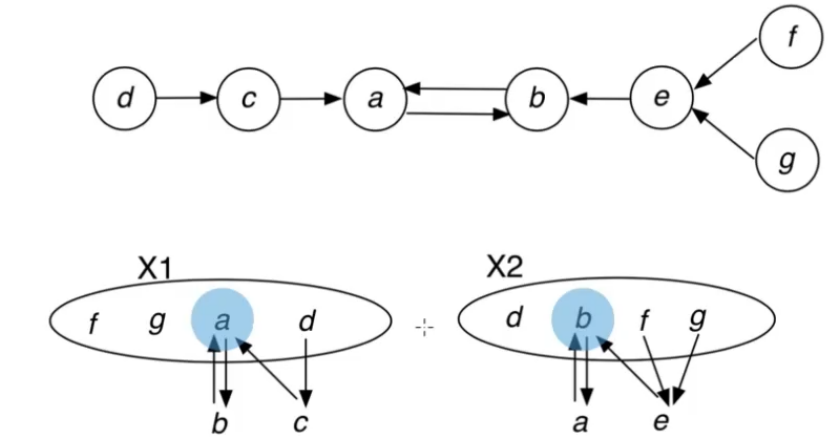
$$d_n^m \triangleright d_t^s \iff m \leq s \quad \text{AND} \quad t \leq n$$

Ci possono essere delle funzioni che sono incomparabili. Nell'esempio successivo vediamo che $a_3 \in d_3^3$ e $a_4 \in d_4^4$ quindi dalla regola vista in precedenza ci viene che il n deve essere maggiore quindi fra a_3 e a_4 è meglio a_3 ma allo stesso tempo m deve essere minore quindi a_4 è un'argomentazione migliore. Questo porta a una contraddizione perchè non riusciamo a classificare i due argomenti.



$$a_3 \notin d_2^2(X3) \quad a_3 \in d_3^3(X3) \quad a_4 \in d_2^2(X4) \quad a_4 \notin d_3^3(X4)$$

Figura 2.19: Esempio graded defense partial order.



$$a \notin d_2^1(X1) \quad b \in d_2^1(X2)$$
$$a \in d_1^2(X1) \quad b \in d_1^2(X2)$$

$$b \succ^{pref} a$$

Figura 2.20: Applicazione di graded semantics a AF .