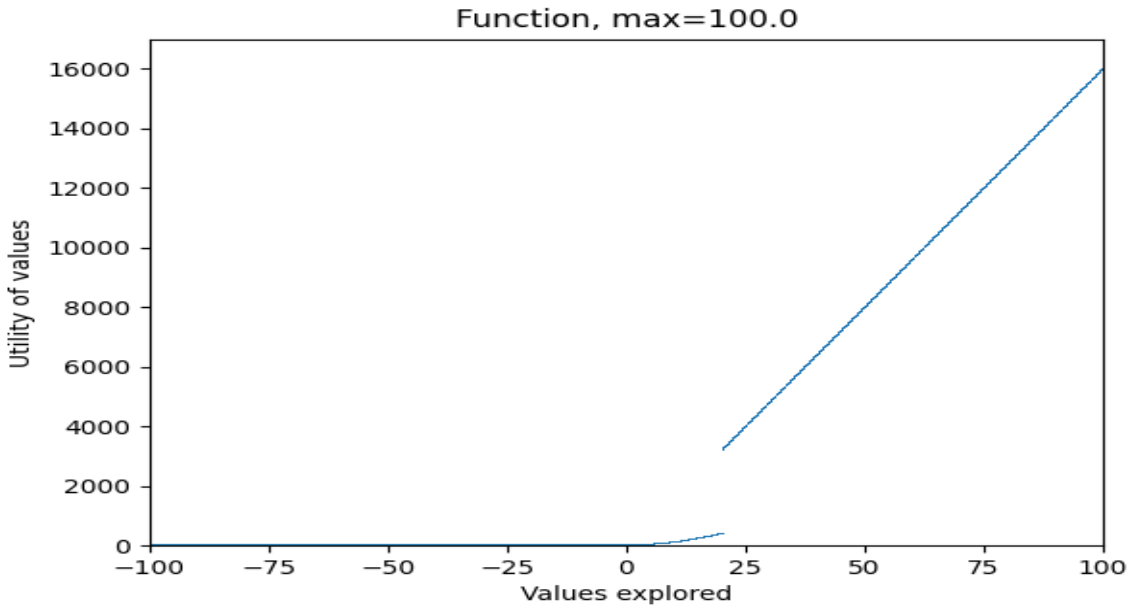
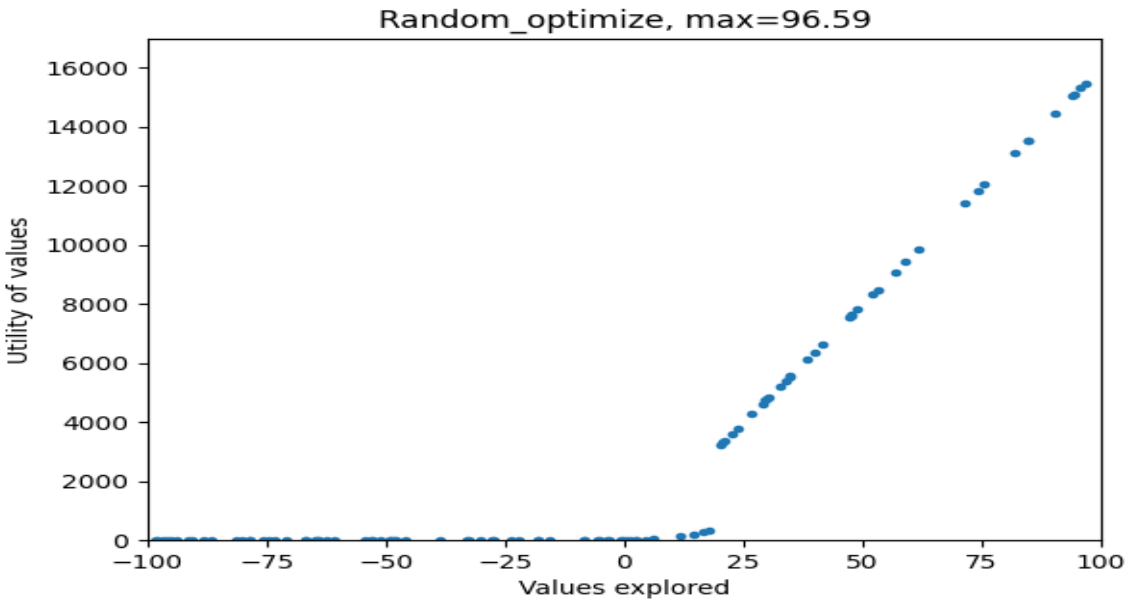
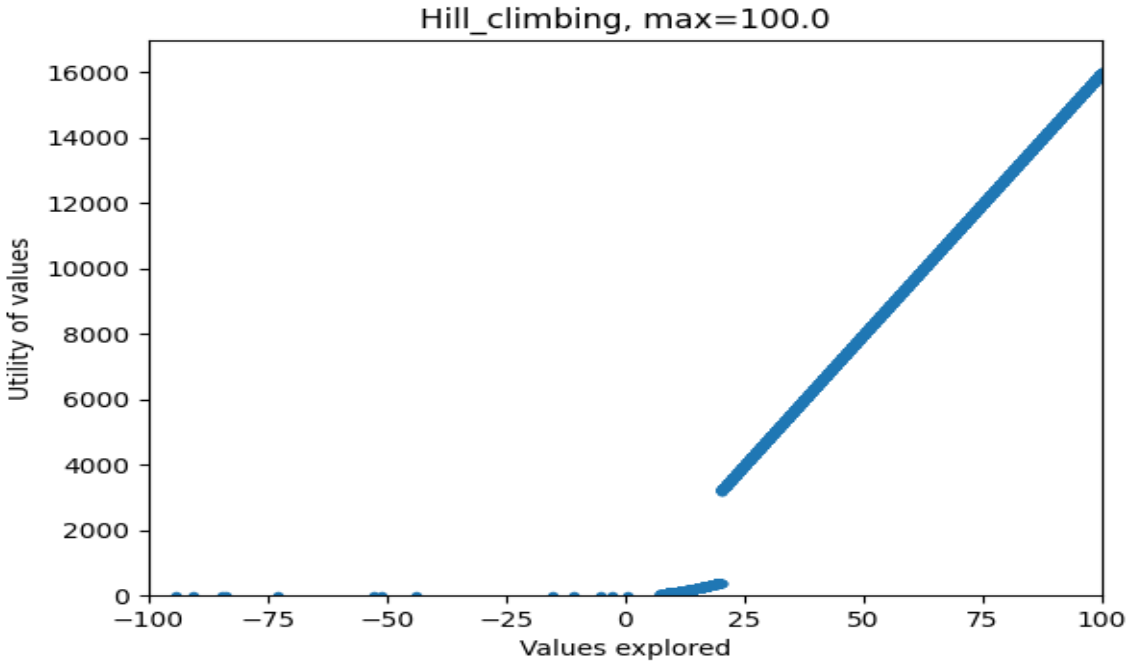
Work Project 2 – Ottimizzazione delle funzioni

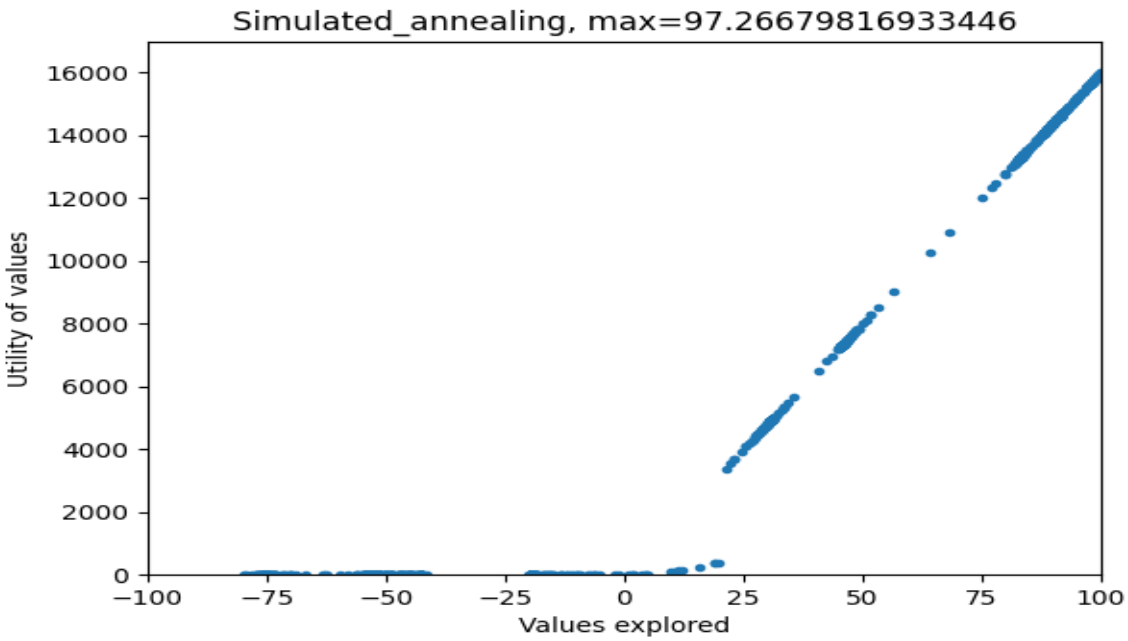
Esercizio 1

Studiando il codice fornito e correggendo eventuali errori presenti, mi sono reso conto che per utilizzare le funzioni di ottimizzazione fornite bisognava definire almeno due parametri: uno è *domain*, il quale è una lista di tuple di due elementi, dove ogni tupla contiene minimo e massimo dei valori che la variabile di stato associata a tale tupla può assumere; l'altro è *costf*, ovvero come secondo parametro bisogna passare una funzione di costo, la quale a sua volta deve essere definita in modo tale da prendere in ingresso le N variabili di dominio e restituire un valore basso se la combinazione delle N variabili costituisce una soluzione vicina all'ottimo, o viceversa una valore alto se le N variabili costituiscono una soluzione lontana dall'ottimo. A tal punto mi sono cimentato sul problema dell'ottimizzazione della funzione data, usando vari approcci. Per la funzione data, *domain* è una lista formata dalla sola tupla (-100, 100) e *costf* è invece 16001 – F(x), cioè avendo creato la funzione F(x), dovevo in qualche modo adattare *costf* in modo da considerare ottimo un valore basso piuttosto che un valore alto, dato che le funzioni del codice fornito funzionano così, e 16001 è sup(F(x)). Ho confrontato i risultati di random optimize, hill climbing e simulated annealing, vedendo che hill climbing era quello che, nel caso in cui lo starting point fosse maggiore di 5.2, si comportava meglio, dato che la funzione è strettamente crescente per x > 5.2. Poi, per complicare un po' il problema, invece di considerare un passo pari ad 1 per x, ho considerato un passo pari a 0.01. Per migliorare le prestazioni del simulated annealing, ho implementato una logica del tipo: "quando la temperatura è più alta, fai passi più grandi", in modo da farlo uscire più velocemente dalla spalla. Dopodiché, ho confrontato di nuovo i risultati delle tre funzioni, però stavolta usando un random restart sia per hill climbing sia per simulated annealing, in particolare: 100 iterazioni di random optimize, ovvero in pratica 100 punti a caso, prendendo il migliore; 25 iterazioni di hill climbing, ovvero 25 random restart di hill climbing, prendendo il risultato migliore; 10 iterazioni di simulated annealing. Per interpretare meglio cosa stava accadendo, anche se quando si parla di ottimizzazione delle funzioni si è interessati al risultato finale, ho modificato le funzioni in modo che accettassero come ulteriore parametro un set di punti esplorati, inizialmente vuoto ma continuato a riempire dopo ogni iterazione; quindi chiaramente ho usato un set distinto per ogni funzione. Ho quindi considerato i punti esplorati ed i loro valori di utilità, e ne ho fatto un plot, confrontandoli con la funzione stessa ed indicando con max l'ascissa a cui è associato il valore di utilità massimo:




Random_optimize, max=96.59
Utility of values
Values explored
16000
14000
12000
10000
8000
6000
4000
2000
0
−100
−75
−50
−25
0
25
50
75
100


Hill_climbing, max=100.0
Utility of values
Values explored
16000
14000
12000
10000
8000
6000
4000
2000
0
−100
−75
−50
−25
0
25
50
75
100

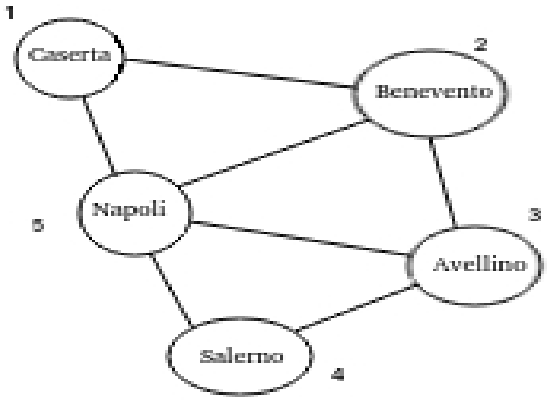


Si possono notare dai grafici le caratteristiche dei tre algoritmi: per il random optimize, si hanno tutti punti isolati. Per l'hill climbing, si hanno punti isolati sulla spalla, cioè dove la funzione è costante, ma dove la funzione è strettamente crescente "la montagna viene scalata con successo", esplorando i valori che portano fino al massimo, con uno step di 0.01. Nel simulated annealing, ci sono diversi "fasci" indipendenti tra di loro, i quali tendono a salire dove possibile, ma esplorano, anche sulla spalla, molti punti del dominio. In particolare, sulla spalla, dato che c'è il passo variabile con la temperatura, che è più grande all'inizio, si può pensare all'esplorazione dei valori un po' come ad un pendolo che ha un carattere oscillatorio smorzato, quindi all'inizio si sposta molto dal punto iniziale ma quando la temperatura comincia a diminuire, si sposta di meno ed anche con minore probabilità; quando riesce a "fuggire" dalla spalla, riesce a salire verso valori più alti. Notare che, anche se l'hill climbing, per com'è fatta la funzione, riesce a trovare l'ottimo, il simulated annealing è l'algoritmo che con meno iterazioni di random restart riesce ad esplorare il maggior numero di punti del dominio, quindi ipoteticamente sarebbe il candidato ideale per trovare un massimo globale dato da un picco improvviso nella zona costante della funzione, cosa che l'hill climbing e il random optimize farebbero con bassa probabilità.

Esercizio 2

La mappa della regione Campania è la seguente:

La rappresentazione conveniente per visualizzare i vincoli è quella a grafo:



Quindi consideriamo ogni provincia come un attributo, ed ogni attributo ha valori appartenenti tutti allo stesso dominio, che è quello dei colori. Quindi indichiamo con X_1 l'attributo "colore di Caserta sulla mappa" e così via, con i numeri mostrati sul grafo. Le constraint del problema possono chiaramente essere espresse anche nel formalismo CSP, che in maniera abbastanza relaxed per abbreviare la notazione è il seguente: $\{X_1 != X_2, X_1 != X_5, X_2 != X_3, X_2 != X_5, X_3 != X_5, X_3 != X_4, X_4 != X_5\}$

Dunque, usiamo l'algoritmo backtracking search per CSP, con le seguenti policies: "minimum remaining values" riguardante la scelta del prossimo attributo da assegnare, "least constraining value" per la scelta del valore da assegnare all'attributo tale da giungere più velocemente alla soluzione, "forward checking" come inferenza.
Nel caso 1, abbiamo solo 2 colori: R e B => $D_i = \{R,B\}$, i=1...5
$X_1 = R \Rightarrow D_2$, $D_5 = \{B\}$
$X_2 = B \Rightarrow D_3 = \{R\}$, $D_5 = \{\} \Rightarrow$ *failure*
Torno indietro: assegno $X_1 = B$, poi $X_2 = R$ e fallisco di nuovo. Quindi, risulta che il problema non è risolvibile con due colori.
Nel caso 2, abbiamo 3 colori: R-G-B => $D_i = \{R,G,B\}$, i=1...5
$X_1 = R \Rightarrow D_2$, $D_5 = \{G,B\}$
$X_2 = G \Rightarrow D_3 = \{R,B\}$, $D_5 = \{B\}$
$X_5 = B \Rightarrow D_3 = \{R\}$, $D_4 = \{R,G\}$
$X_3 = R \Rightarrow D_4 = \{G\}$
$X_4 = G$
Soluzione: $\{X_1 = R, X_2 = G, X_3 = R, X_4 = G, X_5 = B\}$
Abbiamo trovato una soluzione, quindi il problema è risolvibile con tre colori, e la soluzione trovata è un pattern per tutte le altre soluzioni, dato che possiamo assegnare a piacimento i colori.
Nel caso 3, abbiamo 4 colori: C-M-Y-K => $D_i = \{C,M,Y,K\}$, i=1...5
$X_1 = C \Rightarrow D_2$, $D_5 = \{M,Y,K\}$
$X_2 = M \Rightarrow D_3 = \{C,Y,K\}$, $D_5 = \{Y,K\}$
$X_5 = Y \Rightarrow D_3 = \{C,K\}$, $D_4 = \{C,M,K\}$
$X_3 = C \Rightarrow D_4 = \{M,K\}$
$X_4 = M$
Soluzione: $\{X_1 = C, X_2 = M, X_3 = C, X_4 = M, X_5 = Y\}$
Ovviamente anche con quattro colori il problema è risolvibile, e questa volta sul pattern delle soluzioni abbiamo dei gradi di libertà in più, ad esempio non è detto che X_4 debba essere uguale ad M, poteva anche essere scelto uguale a K.

Esercizio 3

Per risolvere un problema con un Algoritmo Genetico, bisogna trovare una rappresentazione della soluzione generica come stringa, ovvero la stringa deve essere assimilabile ad un'assegnazione delle variabili. Nel caso in esame, ci sono 5 variabili, ed ogni variabile può assumere un valore preso da un dominio formato da 4 elementi, quindi si può fare la scelta C=1, M=2, Y=3, K=4, ed in tal caso un gene sarebbe del tipo:
12243

Ovviamente si poteva anche evitare questo passaggio ai numeri ed utilizzare direttamente le lettere. Comunque, supponendo di usare i numeri, si ha quindi una popolazione composta da geni simili a quello preso come esempio, con disposizioni dei numeri da 1 a 4 su 5 posizioni. Si ottiene quindi una nuova popolazione in seguito a: selezione, cross-over, mutazione. Per la selezione, bisogna definire una funzione di *fitness*, ed in tal caso ha senso scegliere come funzione il numero di constraint rispettate: associo al carattere in posizione 1 della stringa il valore da assegnare all'attributo X_1 e così via, quindi ottengo un'assegnazione degli attributi, e poi conto il numero di constraint rispettate da tale assegnazione. Per questo problema, c'è un totale di 7 condizioni, quindi se sono rispettate 7 constraint allora significa che l'algoritmo ha prodotto una soluzione. I meccanismi di cross-over e mutazione sono tipicamente definiti da chi implementa in modo generalizzato gli Algoritmi Genetici, in modo che essi lavorino su delle stringhe. Comunque ad esempio un cross-over potrebbe essere 3:2, tipo:
{122|43, 421|31} => {42143, 12231}
Ed un esempio di mutazione, che comunque avviene con probabilità bassa:
12243 => 12213
Quindi una mutazione sceglie una posizione e converte il valore presente in quella posizione in un altro valore dello stesso dominio.

Esercizio 4

4.1) Dato un insieme KB di proposizione logiche, ci chiediamo se possiamo stabilire, a partire dalla base di conoscenza KB, la veridicità di una proposizione logica α. Dire che α consegue logicamente da KB vuol dire dimostrare che, laddove le proposizioni di KB sono vere, anche α è vera. Formalmente, un'assegnazione di variabili m si dice modello di α se α è vera in m; l'insieme dei modelli di α si dice M(α). La conseguenza logica si dice anche *entailment* e si indica: KB |= α ; condizione necessaria e sufficiente per l'entailment è la seguente: KB |= α se e solo se M(KB) è incluso in M(α). A parole, questo vuol dire che l'insieme dei modelli di KB è un insieme incluso nell'insieme dei modelli di α, ovvero le proposizioni di KB sono più "restrittive" rispetto ad α. Ad esempio, indicando con L la proposizione "oggi è lunedì" e con P la proposizione "oggi piove", se KB = L ∩ P ed α = P, risulta chiaro che KB |= α perché se è lunedì e piove, allora è vero che piove; inoltre, si ha che M(KB) = {(giorno=lunedì, meteo=piovoso)}, mentre M(α) = {(giorno=lunedì, meteo=piovoso), (giorno=martedì, meteo=piovoso) …} quindi è chiara la condizione che lega l'entailment all'insieme dei modelli.

4.2) Effettuare la dimostrazione di una proposizione logica tramite *model checking* vuol dire costruire una tabella di verità, enumerando tutti i possibili mondi; di questi mondi, si considerano quelli che sono modelli per la base di conoscenza KB, ovvero le righe a cui è associato un valore True e si dimostra una proposizione logica α come vera se essa è vera per tutti i mondi in cui KB è vera. Il problema di questo approccio è che la verifica della veridicità di α è una procedura di complessità temporale esponenziale. Vediamo comunque un esempio di come funziona:
Supponiamo KB := (P1∩⌐P2) U (⌐P1∩P2)

P1	P2	KB
F	F	F
F	T	T
T	F	T
T	T	F

Evidentemente KB è una xor. Mi chiedo se P3 = P1∩P2 consegue logicamente da KB. Nei modelli di KB, si ha (P1, P2) = {(F,T), (T,F)}, e per nessuno di essi P3 è vera, quindi P3 non consegue logicamente da KB e poiché non è vera per neanche un modello di KB, si dice insoddisfacibile.

Mi chiedo se P4 = ⌐P1 consegue logicamente da KB. P4 è vera quando (P1, P2) = (F,T) ma è falsa quando (P1, P2) = (T,F), quindi, dato che non è vera per tutti i modelli di KB, P4 non consegue logicamente da KB; in questo caso, si dice che P4 è soddisfacibile.
Mi chiedo se P5 = P1 U P2 consegue logicamente da KB. La risposta in questo caso è sì, e P5 si dice valida.

4.3) Il problema che ci si pone quando si parla di deducibilità logica è analogo a quello che ci si pone quando si parla di conseguenza logica, con delle piccole differenze: dedurre la veridicità di una proposizione logica α a partire da un insieme di proposizioni KB. Mentre la conseguenza logica, ovvero l'entailment, è definito mediante modelli e quindi è basato sulla teoria degli insiemi, la deducibilità logica tratta il problema di trovare α tra le conseguenza di KB, che è come trovare un ago in un pagliaio. Si scrive: KB $|-_i$ α e questo vuol dire che α può essere derivata (dedotta) da KB mediante la procedura i; la procedura i si dice inferenza. Notare che tra deducibilità e conseguenza logica valgono relazioni di consistenza, cioè KB $|-_i$ α => KB |= α, e di completezza, cioè KB |= α => KB $|-_i$ α; quindi deducibilità e conseguenza logica sono equivalenti. La differenza fondamentale è che l'inferenza fornisce una soluzione algoritmica (non enumerativa) al problema di determinare se α consegue da KB.

4.4) Per arrivare alla regola di risoluzione, bisogna dare delle definizioni. Innanzitutto, quella di equivalenza logica: α≡β se e solo se α |= β e β |= α, ovvero se ogni modello di α è modello di β e viceversa. A tal punto, come leggi di equivalenza logica ci interessano fondamentalmente le relazioni di De Morgan:
⌐(a ∩ b) ≡ (⌐a U ⌐b)
⌐(a U b) ≡ (⌐a ∩ ⌐b)
Mediante tali relazioni, insieme a qualche altra legge, è possibile portare qualsiasi proposizione logica in CNF (forma normale congiuntiva – una AND di OR). Altre definizioni: una proposizione si dice valida se è vera in tutti i modelli, si dice soddisfacibile se è vera per alcuni modelli ma non per tutti ed insoddisfacibile se non è vera per alcun modello. La validità è connessa all'inferenza mediante il teorema di deduzione:
KB |= α se e solo se (KB => α) è valida
Tuttavia, per dimostrare la validità è necessario provare tutti i modelli, quindi per ottenere una dimostrazione potenzialmente più veloce usiamo una prova per assurdo:
KB |= α se e solo se (KB ∩ ⌐α) è insoddisfacibile
Per costruire la regola di risoluzione, partiamo dal *modus ponens*:

$$\frac{A\,,A \rightarrow B}{B}\quad ,$$

che significa: se A è vera, e A => B è vera, allora deduciamo B.
Ma: A => B ≡ ⌐A U B, quindi posso riscrivere il modus ponens:

$$\frac{A\,,\neg A\,U\,B}{B}$$

Notare che A e ⌐A sono complementari: in generale, al denominatore, dove ci sono le clausole dedotte, si scrivere la disgiunzione della coppia di clausole al numeratore eliminando i letterali complementari (uno per clausola):

$$\frac{\neg b_j U\, a_1 U \ldots U\, a_n, \neg a_i U\, b_1 U \ldots U\, b_m}{a_1 U \ldots U\, a_{i-1} U\, a_{i+1} U \ldots U\, a_n U\, b_1 U \ldots U\, b_{j-1} U\, b_{j+1} U \ldots U\, b_m}$$

E’ questo il motivo per cui uso le relazioni di De Morgan per portare KB ed α in CNF. Il “modus ponens generalizzato” appena visto è proprio la regola di risoluzione. A tal punto, l’algoritmo da implementare consiste nell’iterare la regola di risoluzione considerando coppie di clausole prese dall’insieme di proposizioni KB ∩ ⌐α. Se trovo due clausole complementari, e quindi ottengo una clausola vuota al denominatore, mi fermo perché vorrà dire che KB ∩ ⌐α è insoddisfacibile e quindi posso dedurre che KB |= α, altrimenti aggiungo le clausole dedotte ad un nuovo set di clausole. Se, dopo aver provato a due a due tutte le possibili coppie, trovo che questo nuovo set è incluso nel set iniziale, allora vuol dire che non ho dedotto niente di nuovo e quindi mi fermo, concludendo che α non è conseguenza logica di KB, altrimenti ripeto l’algoritmo usando il nuovo set di clausole.
In pseudocodice:

```
def dimostrazione(KB, a):
    clausole = CNF(set(KB and not a))
    deduzioni = set()
    while True:
        for Ci, Cj in clausole:
            deduzione = risoluzione(Ci, Cj)
            # notare che la deduzione True non è vuota
            if deduzione == clausola_vuota:
                return True
            # non ha senso aggiungere la deduzione True perché è una tautologia
            if deduzione is not True:
                deduzioni.add(deduzione)
        if deduzioni.subset(clausole):
            return False
        clausole += deduzioni
        deduzioni = set()
```

4.5) Supponiamo di porci un problema di pura logica e di voler fare delle inferenza su di esso in logica proposizionale. Il testo del problema è il seguente:

“Quattro uomini sono indiziati di omicidio. Alle domande della polizia rispondono:
Antonio: Ho visto Carlo e Dario sul luogo del delitto, quindi uno di loro è l’assassino.
Bernardo: Non sono stato io.
Carlo: E’ stato Dario, l’ho visto sparare.
Dario: Giuro che è stato Bernardo, l’ho visto mentre fuggiva.
Se uno solo di loro ha mentito, chi è veramente il colpevole?”

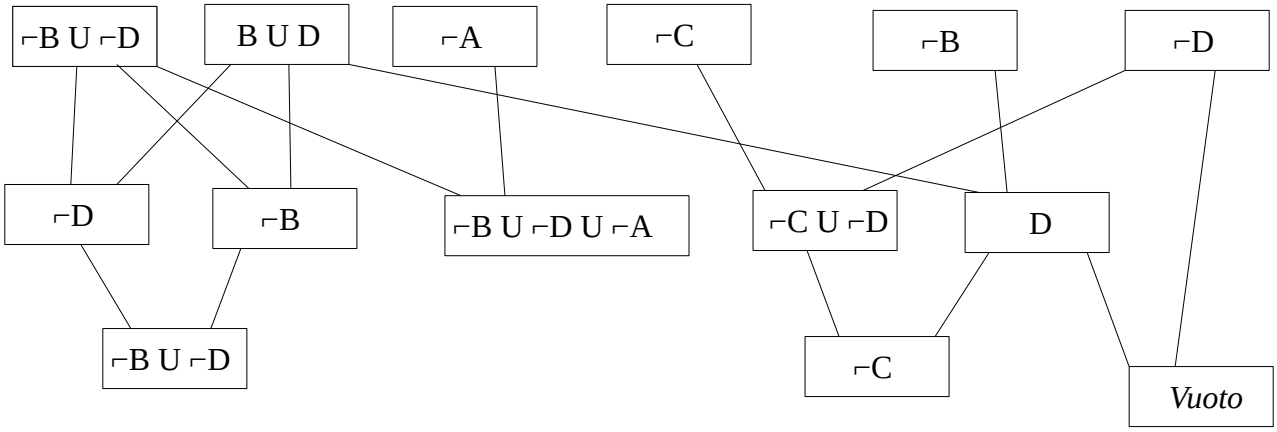
Bisogna tradurlo in logica proposizionale. Chiamiamo A=”Antonio è il colpevole”, quindi analogamente definiamo le proposizioni B, C e D. Poi chiamiamo con P_x le risposte degli indiziati alla polizia:
$P_A = (C \cap \neg D) \cup (\neg C \cap D)$
$P_B = \neg B$
$P_C = D$
$P_D = B$
Il fatto che solo uno di loro ha mentito è molto conciso da dire in italiano ma per esprimerlo in logica proposizionale dovremmo considerare l’or esclusivo tra le possibili terne vere, ed in totale abbiamo quattro possibili distinte terne vere. Inoltre poi bisogna convertire questo or esclusivo in CNF, quindi si otterrebbero N = 24 – 4 = 20 clausole. Però in realtà:
$P_A \cap P_B \cap P_C = \neg B \cap D$
$P_A \cap P_B \cap P_D = \emptyset$
$P_A \cap P_C \cap P_D = D \cap B$

$P_B \cap P_C \cap P_D = \emptyset$

Così facendo sono state eliminate ben due proposizioni, mentre la terza risulta contraddittoria perché afferma che sono stati D e B insieme i colpevoli; in effetti, questa è una “regola non scritta”, ovvero non è specificato che il colpevole sia solo uno. Ammettendo questa possibilità (in caso contrario sapremmo già chi è il colpevole) otteniamo, facendo i conti, la seguente KB:
KB = (⌐B U ⌐D) ∩ (B U D)
che è la xor tra B e D espressa in CNF; otteniamo alla fine che il colpevole è solo uno, ma abbiamo incertezza su chi è dei due. Comunque, dato che sono presenti anche A e C e si possono fare inferenze su di loro, bisogna aggiungerli alla KB:
KB = (⌐B U ⌐D) ∩ (B U D) ∩ ⌐A ∩ ⌐C

Poniamoci quindi la seguente domanda: α = B U D

Si ha: ⌐α = ⌐(B U D) = ⌐B ∩ ⌐D



In questa vista dei passi effettuati dall’algoritmo sono riportarti solo alcuni passi del “primo livello di deduzione” ed ancora meno passi del “secondo livello di deduzione”, comunque sono abbastanza per: vedere che alcuni passi del primo livello non porteranno a deduzioni utili all’inferenza che si sta facendo, ad esempio quelli con ⌐A e ⌐C ; vedere che al secondo livello ci sono delle ridondanze, cioè vengono dedotte delle cose che si sapevano già, e queste poi, nel caso in cui si cerchi di dimostrare una α che non è conseguenza logica di KB, ad un certo punto saranno le sole proposizioni ad essere dedotte e quindi si ricadrà nel caso in cui l’insieme delle deduzioni è incluso nell’insieme complessivo delle clausole. Sono state chiaramente anche incluse le combinazioni di clausole necessarie per dimostrare l’insoddisfacibilità di KB ∩ ⌐α del caso in esame, il che è giusto perché vuol dire che α è vera: la KB considera vera la xor tra B e D, quindi è giusto che sia vera la or tra B e D. Se lo si vuole pensare con il model checking, per ogni riga della tabella di verità in cui KB è vera, α risulta vera.