UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

IL PROBLEMA DEL SUDOKU: FORMALIZZAZIONE DEL PROBLEMA E RASSEGNA DEI PRINCIPALI ALGORITMI RISOLUTIVI CON ANALISI DI COMPLESSITA'

SUDOKU PROBLEM: FORMALIZATION AND STUDY OF THE MAIN ALGORITHMS WITH COMPLEXITY ANALYSIS

FRANCESCO BELLEZZA

Relatrice: *Maria Cecilia Verri*

Anno Accademico 2021-2022

INDICE

ELENCO DELLE FIGURE

1

INTRODUZIONE

1.1 SUDOKU

Il sudoku è un gioco di logica costituito da una griglia 9x9 e da 9 sottogriglie 3x3. In questo schema, sono inseriti inizialmente numeri da 1 a 9 in alcune celle e, lo scopo del risolutore è quello di completare l'inserimento in modo tale che:

- ogni casella contenga solo numeri da 1 a 9
- in ogni riga, colonna e riquadro 3x3 non siano presenti due numeri uguali

In figura 1 un esempio di sudoku: le linee in grassetto delimitano i riquadri 3x3. Il gioco può essere esteso ad una griglia *nxn* e la sua

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figura 1.: Esempio di sudoku, schema preso da `it.wikipedia.org/wiki/Sudoku`

risoluzione è un problema NP[1]: verificare la correttezza della soluzione è possibile in tempo polinomiale, ma ad oggi non è conosciuta una strategia polinomiale per risolvere un sudoku di grandezza generica nxn. Inoltre sappiamo che la risoluzione di un sudoku nxn è un problema NP-completo[11]. Un problema P viene considerato NP-completo se e solo se

1 Non determinisco polinomiale: esiste quindi un algoritmo in grado di risolvere il problema utilizzando una macchina non deterministica in tempo polinomiale

P appartiene ad NP e, per ogni problema P' appartenente a NP, esiste una riduzione polinomiale da P' a P. La riduzione polinomiale è una funzione che consente (in tempo polinomiale) di convertire le istanze del problema P' in istanze del problema P. In pratica se P è NP-completo, è possibile, preso un qualunque elemento P' dell'insieme NP, trasformare le istanze di P' in P. Questo ci dice che se fosse nota una strategia polinomiale per risolvere un qualunque schema *nxn*, avremmo dimostrato che P=NP (in quanto il sudoku nxn è NP-completo), risolvendo uno dei problemi rimasti insoluti della complessità computazionale[10].

1.2 ALGORITMI ANALIZZATI

Il sudoku può essere considerato come una serie di condizioni logiche da soddisfare e, di conseguenza, è possibile ridurlo ad una espressione booleana da risolvere con un risolutore SAT[2] [1]. Sarà possibile poi, sfruttando le celle con valori già definiti all'interno della griglia, diminuire il numero di clausole e letterali presenti, in modo da rendere l'algoritmo ancora più veloce. Più numeri sono presenti come indizi, maggiore è la riduzione della grandezza della formula. Un altro approccio interessante è quello che tratta il sudoku come un problema di exact cover [2]: viene utilizzata una matrice binaria dove ogni colonna descrive una condizione da soddisfare del sudoku. Sarà necessario soddisfare ogni colonna, quindi la matrice dovrà contenere uno e un solo 1 per ogni colonna, creando così un exact cover di quest'ultima. (Tale risultato si ottiene mediante la scelta di opportune righe della matrice). L'algoritmo è di tipo ricorsivo e, la peculiarità principale, consiste nello sfruttare i Dancing Link di Knuth [2] per avere una implementazione molto efficiente. L'ultimo metodo che verrà approfondito consiste nell'utilizzo dell'algoritmo del simplesso per minimizzare una funzione obiettivo, chiamato $\beta - \mathtt{hill}$ climbing [3]. Inoltre, verranno spiegate alcune tecniche utilizzate dai sudokisti per facilitare la risoluzione di schemi particolarmente complessi. [6] Queste strategie potrebbero essere utilizzate prima di ogni algoritmo di risoluzione del sudoku per renderlo più veloce. E' stato implementato un algoritmo ottimizzato per la risoluzione di sudoku che alterna la forza bruta con alcune delle tecniche più semplici sfruttate dai risolutori umani di sudoku.

2 Il risolutore SAT ha lo scopo di trovare, se esiste, un assegnamento delle variabili nell'espressione booleana in forma CNF (congiunzione di clausole, dove una clausola consiste in un letterale o una disgiunzione di letterali) in modo tale da soddisfare l'espressione. Il problema SAT è un problema NP-completo.

2

RIDUZIONE SUDOKU A SAT

2.1 IMPOSTAZIONE DELLA FORMULA IN CNF

Il sudoku intrinsecamente definisce una serie di condizioni logiche per essere risolto. Se venisse effettuata la congiunzione di tali condizioni, allora l'assegnamento che soddisferà la formula così ottenuta, darà la soluzione dello schema.
Nel caso delle espressioni booleane, le variabili possono assumere solo due valori: 0 o 1. Ciascuna cella però può assumere 9 valori differenti da 1 a 9. Per risolvere questo problema, creiamo una variabile (r,c,n): se nella riga r e colonna c è presente il valore n, allora (r,c,n) vale 1, altrimenti vale 0: quindi ad ogni cella corrispondono 9 variabili.
Iniziamo definendo le celle che assumono già un valore all'inizio della risoluzione: verranno inserite in una congiunzione che le contiene tutte. In questo modo, se nello schema è presente un 1 nella prima riga e prima colonna e un 2 nella prima riga e seconda colonna, inseriamo nella formula la seguente congiunzione: $(1,1,1) \wedge (1,2,2)$
In questo modo il sudoku può essere risolto se e solo se queste due variabili assumono valore 1. Per il conteggio delle clausole nella formula finale, tali elementi non vengono considerati, in quanto solitamente il numero di indizi iniziali è variabile in base alla difficoltà dello schema.
Affinché il sudoku sia corretto, ogni casella deve contenere almeno un numero da 1 a 9. Tale condizione deve valere per ogni cella, quindi sarà necessario effettuare una congiunzione di tutte le variabili che esprimono il valore delle celle. Questo è possibile facendo variare rispettivamente l'indice di riga e di colonna da 1 a 9. Si crea quindi una congiunzione delle possibili righe che vanno da 1 a 9 di una congiunzione delle possibili colonne che vanno da 1 a 9, ricoprendo così tutte le 81 caselle dello schema. Ognuna di queste caselle contiene almeno un numero da 1 a 9:

in logica, l'almeno uno si traduce con una disgiunzione di tutti i possibili valori che la cella può assumere. La formula ottenuta è la seguente:

$$\bigwedge_{r=1}^{9}\bigwedge_{c=1}^{9}\bigvee_{n=1}^{9}(r,c,n) \tag{2.1}$$

Sono presenti 9x9 clausole con nove letterali ciascuna, per un totale di 729 variabili booleane.
Il sudoku è composto però anche da righe, colonne e riquadri. In uno schema, ogni numero deve apparire almeno una volta sia nelle righe, sia nelle colonne e sia nei riquadri. Analizziamo il caso delle righe e, per chiarezza espositiva, controlliamo che nella prima riga ci sia in ogni casella almeno un numero. Quindi, scorriamo la prima riga e controlliamo se è presente il numero uno almeno una volta. Se non fosse presente, tale formula ritornerebbe falso, che è il risultato atteso. Ripeto tale operazione per gli altri numeri, fino a 9. Quindi, in una riga, devo controllare che ciascun numero compaia almeno una volta, facendo variare l'indice di colonna. Considerando che tale condizione deve valere per tutte le righe, bisognerà effettuare una congiunzione per tutti gli indici r che vanno da 1 a 9. Il risultato è scritto nella formula sottostante:

$$\bigwedge_{r=1}^{9}\bigwedge_{n=1}^{9}\bigvee_{c=1}^{9}(r,c,n) \tag{2.2}$$

Anche in questo caso, otteniamo 81 clausole con 729 variabili booleane.
Continuando il filo logico che è stato seguito, adesso vi è la necessità di controllare che ogni numero compaia almeno una volta all'interno di ogni colonna. La logica è la stessa di quella descritta per le righe, solo che, in questo caso, sono gli indici di riga quelli su cui si itera per il controllo di ogni colonna:

$$\bigwedge_{c=1}^{9}\bigwedge_{n=1}^{9}\bigvee_{r=1}^{9}(r,c,n) \tag{2.3}$$

Questa espressione ha sempre 81 clausole da 9 letterali ciascuna.
Controllare che ogni numero compaia almeno una volta in ogni riquadro 3x3 richiede del lavoro extra. Infatti, useremo due indici supplementari, i e j, che servono per identificare il riquadro sotto analisi. Se gli indici i e j sono rispettivamente 0 e 0, il riquadro da analizzare è il primo in alto a sinistra, mentre se sono 0 e 1 il riquadro da analizzare è il secondo partendo in alto da sinistra e così via, fino ad arrivare ai valori 2 e 2 che

rappresentano il riquadro in basso a destra. Gli indici di riga e di colonna in questo caso variano solamente da 1 a 3, in quanto ciascun riquadro contiene 3 righe e 3 colonne. Come in precedenza, per iterare sul riquadro utilizzeremo una congiunzione di tutte le caselle della sottogriglia 3x3 e la incapsuliamo in una serie di disgiunzioni per ogni numero che va da 1 a 9. Così controlliamo che ogni cella di ogni riquadro ha almeno un numero che va da 1 a 9.

$$\bigvee_{n=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{r=1}^{3} \bigwedge_{c=1}^{3} (3i + r, 3j + c, n) \tag{2.4}$$

Quest'ultima formula, seppur abbia una scrittura più estesa rispetto alle precedenti, ha 81 clausole (3x3x3x3) da 9 letterali ciascuna. L'indice *i* e *j* viene moltiplicato per 3 prima di sommarci rispettivamente la r-esima riga del riquadro e la c-esima colonna perché in questo modo si riesce ad accedere alla effettiva cella del riquadro selezionato. Nel caso fossimo nel riquadro centrale (con indici *i* e *j* entrambi uguali ad 1), gli elementi analizzati previsti sarebbero: 4,5,6 sia per le righe sia per le colonne. Analizziamo la prima iterazione per gli indici *i* e *j* entrambi uguali a 1. L'indice *r* va da 1 a 3, così come l'indice *c*. Le colonne della matrice 9x9 analizzate da *c* sono: 3**j*+*c*. Per *c*=1 tale espressione vale 4, per *c*=2 vale 5 e per *c*=3 vale 6, come desiderato. Per le righe il calcolo è analogo, solo che invece di *j* e *c* avremo gli indici *i* e *r*. Questi valori identificano tutte le possibili celle del riquadro centrale. (Prendendo come riferimento le posizioni della matrice 9x9). In figura 2 vengono evidenziate le posizioni riga, colonna del riquadro centrale.
I controlli che abbiamo definito non bastano a definire la correttezza del sudoku: è necessario verificare anche che in ogni cella, riga, colonna e riquadro ogni numero sia presente al massimo una volta. Incominciamo descrivendo il controllo che viene inserito per le celle: come in precedenza, questa condizione deve valere per ogni quadratino 1x1; quindi bisogna creare una serie di congiunzioni per esaminare in completezza tutto lo schema. Per controllare che sia presente al più un numero per ogni casella, effettueremo una serie di confronti in modo simile a come opererebbe un selection sort. Prendiamo come esempio la prima cella in alto a sinistra. Fissiamo quindi l'indice di riga e di colonna a 1 nella serie di congiunzioni necessarie per ciclare l'intero schema. Dobbiamo introdurre una condizione che ritorni falso nel caso in cui ci sia un assegnamento che imposta il valore di (1,1,n) e (1,1,N) a 1 (con n e N contenenti cifre differenti). La formula richiesta è una disgiunzione di queste due celle

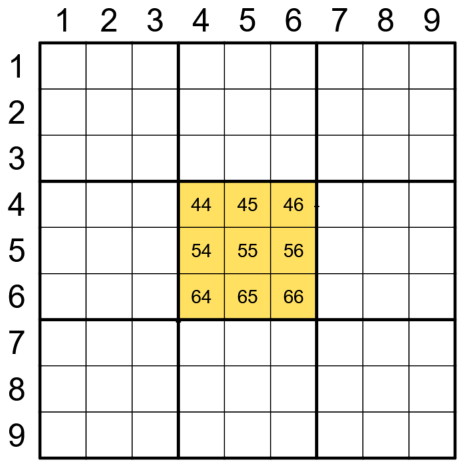

Figura 2.: Le celle evidenziate in giallo indicano il riquadro centrale. Ai bordi dello schema sono presenti gli indici di posizione relativi. In questo caso gli indici di posizione sia per le righe che per le colonne sono 4,5,6.

-entrambe negate-. In questo caso, tale espressione booleana ritorna falso se e solo se sono presenti due valori diversi nella cella.
Per controllare tutti i possibili valori, prima viene confrontato il valore 1 con il valore 2,3,...9 , poi la cifra 2 con la cifra 3,4,...9 e così via (il 2 non si confronta nuovamente con il numero 1, perché è stato già paragonato nel ciclo precedente). La formula ottenuta è la seguente:

$$\bigwedge_{r=1}^{9}\bigwedge_{c=1}^{9}\bigwedge_{n_i=1}^{8}\bigvee_{n_j=v_i+1}^{9} \neg(r,c,n_i)\vee\neg(r,c,n_j) \tag{2.5}$$

Essa ha 9x9x36 clausole binarie, per un totale di 2916 clausole.
La logica per le colonne e per le righe è simile, l'unica differenza riguarda gli indici per controllare che non vi siano presenti ripetizioni: se si vuole verificare che in ogni riga non compaia più volte lo stesso elemento, le celle vengono confrontate incrementando l'indice di colonna, viceversa per le colonne si cicla sull'indice di riga.
Per le righe:

$$\bigwedge_{r=1}^{9}\bigwedge_{n=1}^{9}\bigwedge_{c_i=1}^{8}\bigvee_{c_j=c_i+1}^{9} \neg(r,c_i,n)\vee\neg(r,c_j,n) \tag{2.6}$$

Per le colonne:

$$\bigwedge_{c=1}^{9}\bigwedge_{n=1}^{9}\bigwedge_{r_i=1}^{8}\bigvee_{r_j=v_i+1}^{9}\neg(r_i,c,n)\vee\neg(r_j,c,n) \tag{2.7}$$

In ognuna di queste due espressioni sono presenti 2916 clausole binarie. Anche in questo caso, la logica per i riquadri è più complessa. La formula, per chiarezza di esposizione, viene divisa in due parti: l'espressione finale è ottenuta dalla congiunzione delle due formule: la prima contiene (9x3x3x3x3=729) clausole binarie, la seconda (729x3 = 2187).

$$\bigwedge_{n=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{r=1}^{3}\bigwedge_{c=1}^{3}\bigwedge_{k=c+1}^{3}\neg(3i+r,3j+c,n)\vee\neg(3i+r,3j+k,n) \tag{2.8}$$

Come in precedenza *i* e *j* selezionano il riquadro da esaminare. Fissata una cella, si controlla che nella stessa riga di quest'ultima non ce ne sia un'altra con lo stesso valore, effettuando una serie di confronti nella stessa modalità che è stata precedentemente descritta. Questo però, non basta: bisognerebbe controllare anche che tale numero nella cella non compaia in altre celle nel riquadro: questo è il compito della seconda formula, che confronta l'elemento selezionato con tutti i rimanenti che nono sono stati confrontati precedentemente (tutti gli elementi quindi che si trovano nelle righe inferiori).

$$\bigwedge_{n=1}^{9}\bigwedge_{i=0}^{2}\bigwedge_{j=0}^{2}\bigwedge_{r=1}^{3}\bigwedge_{c=1}^{3}\bigwedge_{k=c+1}^{3}\bigwedge_{l=1}^{3}\neg(3i+r,3j+c,n)\vee\neg(3i+k,3j+l,n) \tag{2.9}$$

La congiunzione di tutte le espressioni booleane che abbiamo descritto, descrive correttamente il problema del sudoku. Tale formula è composta da 11,988 clausole. Esiste anche una forma più compatta, che contiene meno clausole rispetto a quella descritta qui sopra. Per controllare la validità di un sudoku è sufficiente verificare che in ogni cella ci sia almeno un numero da 1 a 9 e che in ogni riquadro, riga e colonna non si ripetano numeri. La congiunzione di queste formule viene chiamato "Minimal Encoding", ed esso contiene 8,829 clausole, mentre l'espressione con 11,988 clausole prende nome di "Extended Encoding".
Nell'articolo [1] viene effettuato un confronto tra minimal encoding ed extended encoding in termini di prestazioni: è immediato osservare che l'extended encoding risulta essere più performante rispetto ad un minimal encoding. Purtroppo è presente ancora un problema: entrambi

gli algoritmi, di fronte ad uno schema di dimensione 81x81, vanno in stack overflow, in quanto la formula (con opportuni accorgimenti sugli indici rispetto a come la abbiamo descritta) risulterebbe troppo grande. Per velocizzare l'algoritmo, è possibile sfruttare le celle che contengono già dei numeri all'inizio della risoluzione per alleggerire la grandezza della formula.

2.2 OTTIMIZZAZIONE DELLA FORMULA SAT

Per rendere l'espressione booleana più compatta, è possibile sfruttare degli operatori di riduzione che utilizzano le celle con un numero già presente[1]. Infatti, in maniera implicita, tali caselle escludono alcune configurazioni che non si possono mai avverare - o che si presentano in maniera forzata - : rimuovere queste clausole o letterali rappresentanti di questa situazione spetta ad alcuni operatori che in seguito verranno definiti. Le celle che contengono già un valore prima della risoluzione dello schema, le inseriamo in un insieme chiamato V^+: in questo caso *(r,c,v)* relativo alla casella avrà valore 1. Inoltre sappiamo che non può essere inserito un numero se è già presente nella stessa riga, colonna, riquadro o cella. Costruiamo un insieme con questi elementi non validi e chiamiamolo V^-. Queste variabili assumono valore 0 e vengono inserite all'interno dell'insieme mediante 4 predicati. [1]:

1. stessaCella(r,c,v,r',c',v')
2. stessaRiga(r,c,v,r',c',v')
3. stessaColonna(r,c,v,r',c',v')
4. stessoRiquadro(r,c,v,r',c',v')

Gli elementi *r',c',v'* rappresentano gli elementi appartenenti all'insieme V^+. Per ognuno di questi elementi, confronto con tutte le possibili combinazioni di *r,c,v*. Se il predicato ritorna vero, allora la variabile corrispondente viene inserita all'interno dell'insieme V^-.

$$\mathrm{stessaCella\ ritorna\ vero} \Leftrightarrow (r = r') \wedge (c = c') \wedge (v \neq v')$$

In questo modo escludiamo dalla formula tutte le celle che sono state già inserite - se nella posizione *(r,c)* è già presente il valore v, non può esserci assegnato un altro valore -. Ragionamento simile per stessaRiga e stessaColonna, solo che la diseguaglianza riguarda rispettivamente l'indice di colonna e l'indice di riga. Per i riquadri, il problema maggiore consiste

nell'individuare il riquadro di appartenenza in base alla posizione indicata dagli indici *(r,c)* e *(r',c')*. Per risolvere questo problema dobbiamo stabilire come identificare un riquadro: esso si riconosce dalla posizione della cella in alto a sinistra (nel caso del primo riquadro tale cella è la numero (1,1), nel caso del secondo è (1,4)) e, in base a *(r,c)* riusciamo a identificare tale casella. La formula usata per ottenere il quadrato in alto a sinistra della sottomatrice 3x3 è la seguente, adoperando la sintassi Java:

```
(i <= 3) ? 1 : (i<=6) ? 4 : 7;
```

Dove i sarà in una iterazione l'indice di riga, mentre nell'altra è l'indice di colonna. Ora bisogna preoccuparsi di come individuare gli elementi ripetuti in un riquadro. Vengono controllate le variabili con $(r,c) \neq (r',c')$ e con $v = v'$. Per capire se sono nello stesso riquadro è sufficiente controllare che l'indice della riga relativa al riquadro sia minore di r e r' sia minore dell'indice del riquadro relativo alla riga + 3. In formula:

$$(r_{riquadro} \leqslant r) \wedge (r' \leqslant r_{riquadro} + 3).$$

Per le colonne si adotta il medesimo ragionamento e queste due condizioni devono essere entrambe rispettate affinché essa sia una configurazione impossibile. Gli operatori che effettuano la riduzione ci consentono di soddisfare in maniera parziale la formula e, per ogni iterazione, danno la possibilità al risolutore SAT di impiegare meno energie del necessario. Esistono due tipi di operatore: quello che rimuove le clausole che sono sicuramente soddisfatte - o non - ($\Downarrow$) e quello che rimuove i letterali che sicuramente non soddisfano la clausola ($\downarrow$). Entrambi lavorano sia sull'insieme V^+ sia sull'insieme V^-.
Qui sotto verranno descritti in maniera più formale : [1]

$$\phi \Downarrow V^+ = \{C \in \phi \mid \neg\exists L \in C : (L = x \wedge x \in V^+)\}$$
$$\phi \Downarrow V^- = \{C \in \phi \mid \neg\exists L \in C : (L = \neg(x \wedge x) \mid x \in V^-)\}$$
$$C \downarrow V^- = \{L \in C \mid \neg(L = x \Rightarrow x \in V^-)\}$$

Le prime due formule rimuovono la clausola da ϕ se e solo se L ritorna vero, mentre la terza rimuove i letterali se e solo se L ritorna falso. L'idea alla base della prima riduzione è quella di rimuovere tutte le clausole che contengono una cella presente in V^+. Infatti, siccome una clausola è una disgiunzione di letterali, è sufficiente che un singolo elemento abbia valore di verità 1 per soddisfare quest'ultima. (Nel nostro caso, l'elemento in V^+ sappiamo che assume valore 1 per quanto detto prima). La seconda riduzione verrà utilizzata per rimuovere le clausole relative alla condizione

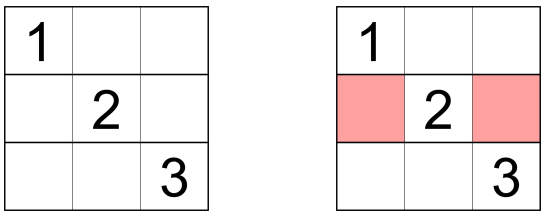

Figura 3.: Il 2 non può ripetersi nella riga 2. Quindi, la condizione di non ripetizione del 2 è già rispettata e le clausole relative possono essere eliminate. Il 2 non può essere inserito nelle celle in rosso

di non ripetizione dei numeri da 1 a 9 in ogni riga colonna e riquadro (insieme alla condizione che in ogni cella è presente uno e un solo numero). Per semplicità di spiegazione, tratteremo il caso del sudoku 3x3 in figura 3 e, nello specifico, analizziamo il comportamento di questa riduzione nel caso del numero 2 presente nella seconda riga. (E' evidente che in questo sudoku non esiste una condizione sui riquadri, ma esula dal nostro interesse, in quanto analizziamo una informazione diversa). La congiunzione relativa alla non ripetizione del numero 2 nella seconda riga è la seguente: (vedi formula 2.6) $((\neg(2,1,2) \vee \neg(2,2,2)) \wedge (\neg(2,1,2) \vee \neg(2,3,2)))$
Sappiamo che in V^- sono presenti le variabili (2,1,2) e (2,3,2): entrambe assumono quindi valore 0. Guardando entrambe le clausole notiamo che sono già soddisfatte entrambe grazie a (2,1,2) e (2,3,2), quindi possono essere rimosse.
Se impostassimo ad 1 tali variabili, avremmo una configurazione insoddisfacibile (rappresentata dalla negazione della congiunzione della variabile con se stessa all'interno della seconda formula di riduzione $\neg(x \wedge x)$).
L'ultimo operatore lavora su una singola clausola. Sappiamo che i letterali presenti in V^- rappresentano celle che sicuramente non fanno parte della soluzione, quindi, possiamo rimuovere queste variabili dalle clausole che determinano la condizione dello schema riguardante la presenza di almeno un numero per ogni cella, e la verifica che ogni numero appaia almeno una volta in ogni riga colonna e riquadro.
Un modo intelligente per rimuovere tali letterali è inserire un controllo per quest'ultimi che ritorni sempre falso, così l'operatore esclude queste variabili dalla formula. $\neg(L = x \Rightarrow x)$ ritorna sempre falso, quindi x verrà rimossa (con $x \in V^-$). Tale operatore va eseguito su tutte le clausole. Con questi operatori è possibile ridurre il peso della formula in media di 12 o 79 volte [1], permettendo anche di risolvere schemi 81x81 senza entrare nella condizione di stack overflow.

3

ALGORTIMO X PER LA RISOLUZIONE DEI SUDOKU

3.1 IDEA DELL'ALGORITMO: PROBLEMA DI EXACT COVER

L'algoritmo X è un algoritmo di backtracking che sfrutta una struttura dati definita da Knuth [2] per risolvere un problema di exact cover. La struttura di tale matrice per rappresentare il sudoku viene esplicitata nelle sezioni successive. Adesso ci interessa comprendere la natura dell'algoritmo X per una generica matrice binaria, che ha lo scopo di trovare un sottoinsieme di righe della matrice in modo tale che sia presente uno e un solo uno per ogni colonna. Per ottenere questo risultato, vengono effettuati i seguenti passaggi: [4]

1:Se la matrice A non contiene colonne, allora siamo arrivati alla soluzione desiderata.
2:Se la matrice A contiene almeno una colonna con soli zeri, allora non esiste un ricoprimento di tale matrice.
3:Altrimenti selezioniamo in maniera deterministica una colonna c
4:selezioniamo una riga tale che l'elemento in posizione (r,c) = 1 (in modo non deterministico)
5:inseriamo la riga r nella soluzione parziale
6:per ogni colonna j tale che (r,j) contiene 1
per ogni riga i<>r tale che in (i,j) e' presente un uno: rimuoviamo la riga i dalla matrice A
7:rimuoviamo la colonna j dalla matrice A
8:alla fine di entrambi i cicli, rimuoviamo la riga r dalla matrice A
9:Ripeti l'algoritmo sulla matrice ridotta A.

Riassumendo, selezioniamo una colonna e proviamo a scegliere una riga che in corrispondenza della colonna ha il numero 1. Proviamo ad inserire tale riga nella soluzione parziale (la riga scelta coprirà sicuramente la colonna c). I prossimi due passaggi descritti sopra, servono per rimuo-

vere le colonne che sono state ricoperte a causa della riga selezionata e rimuovere le righe che se aggiunte alla matrice porterebbero ad avere una colonna con due numeri 1. La riga viene scelta in maniera non deterministica, in questo modo, se esiste un ricoprimento della matrice, verrà scelta la riga che porterà a risolvere il problema di exact cover: ed è per questo motivo che possiamo dire che, qualora sia presente una colonna con soli zeri non esiste un ricoprimento di tale matrice. Esiste quindi la possibilità che, partendo da una certa riga, quest'ultima non ci conduca alla soluzione desiderata: siccome abbiamo adoperato una strategia per la selezione della riga non deterministica, in questo caso significa che la matrice non ha effettivamente un ricoprimento. Per implementare tale algoritmo, bisognerà poi implementare una logica per scegliere la riga (rendendo tale scelta deterministica invece che non deterministica) e, se la riga selezionata portasse ad una configurazione errata, ripristinare una configurazione precedente. In questo caso, la matrice non avrà ricoprimenti se e solo se sono state esaminate tutte le possibili scelte di righe della matrice.
Per chiarire il funzionamento dell'algoritmo, verrà illustrato un esempio. Per la selezione della riga all'interno dell'esempio la scelta è casuale. Sia la matrice A così definita:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{3.1}$$

Proviamo ad eseguire l'algoritmo precedentemente definito. Scegliamo la prima colonna. Adesso dobbiamo selezionare una riga che contiene un uno in corrispondenza della colonna scelta. Possiamo scegliere in questo caso la prima o la quarta o la sesta riga. Scegliamo la prima riga e la aggiungiamo alla soluzione parziale. Successivamente si eliminano tutte le righe che hanno almeno un uno nella stessa colonna della prima riga. (E

si eliminano anche le colonne che contengono un uno in corrispondenza della riga scelta).

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{3.2}$$

Rimuoviamo le righe che contengono gli elementi in rosso e le colonne che hanno i numeri 1 in verde.

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{3.3}$$

Adesso selezioniamo la prima colonna e la seconda riga.

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{3.4}$$

Utilizzando la logica di prima, otteniamo la seguente matrice.

$$\begin{pmatrix} 0 \end{pmatrix} \tag{3.5}$$

Siamo in un vicolo cieco: questo perché nella matrice ridotta è presente una colonna con soli zeri. Infatti, se selezioniamo la prima e la seconda riga della matrice di partenza, è facile osservare che non è presente una riga da selezionare per avere un uno nell'ultima colonna: tutte le possibili righe che possono essere scelte con un uno nell'ultima colonna poterebbero ad avere una colonna con più numeri 1. Quindi ripristiniamo la matrice precedente e selezioniamo invece la terza riga (quinta per la matrice di partenza).

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{3.6}$$

Se rimuoviamo le righe con elementi rossi e le colonne con gli elementi in verde, otteniamo la matrice vuota: il ricoprimento si ottiene selezionando la prima e la quinta riga.

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \tag{3.7}$$

Guardiamo nel caso in cui avessimo scelto all'inizio la quarta riga cosa sarebbe successo.

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \tag{3.8}$$

Rimuovendo le colonne e le righe come prima:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.9}$$

Se reiteriamo altre due volte l'algoritmo vediamo che la matrice di ricoprimento da noi trovata è quella che seleziona la quarta, la seconda e terza riga della matrice di partenza.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \tag{3.10}$$

Generalmente in una matrice binaria possono esserci uno o più ricoprimenti, oppure nessuno: nel caso del sudoku, se lo schema è corretto, l'exact cover dovrebbe essere uno e uno solo. (Altrimenti il sudoku avrebbe più soluzioni).
Il sudoku verrà formalizzato in una matrice binaria, dove ogni riga rappresenta la configurazione *(r,c,n)* di cui si è parlato nell'algoritmo SAT. Le colonne andranno a specificare il tipo di condizioni che devono essere rispettate: una scelta di una determinata riga, quindi, implica l'inserimento di un numero all'interno del sudoku. L'aggiunta dell'elemento avrà una ripercussione sia sulle righe che possono essere selezionate (i numeri cioè che possono essere inseriti), sia sulle condizioni da rispettare (ovvero

le colonne). Seguendo la logica dell'algoritmo quindi, si sceglie prima la condizione (ad esempio la condizione che riguarda la posizione del numero 1 nel riquadro 1). Si sceglie poi la riga tra quelle possibili (ovvero si decide in quale cella nel primo riquadro si inserisce l'1) e si effettua il ricoprimento. Si re-itera questo procedimento fino a quando non viene ricoperta completamente la matrice oppure finiamo in un vicolo cieco e ripristiniamo la configurazione precedente. Se dopo la scelta di una riga, ci ritroviamo con una colonna con soli zeri, significa che una condizione del sudoku non può essere soddisfatta: quindi si ripristina una configurazione precedente. Se il sudoku non ha soluzioni, allora il metodo fallisce dopo aver esplorato tutte le possibilità.
Qui sotto viene fornito un esempio di un sudoku 2x2 per far comprendere al lettore la logica di questo algoritmo applicato al gioco logico. In questo caso non è presente la condizione riguardante il riquadro. Le colonne r1c1, ..., r2c2 indicano le condizioni relative alla celle corrispondenti - simulando un assegnamento alla casella, in base alla riga selezionata -, le colonne rY.Z indica la condizione di non ripetizione del numero Z nella riga Y, mentre cY.Z segnala quella di non ripetizione della cifra Z nella colonna Y. Prendiamo il seguente schema come esempio, dove lo zero indica un numero che non è stato ancora inserito. A sinistra lo schema iniziale, a destra la soluzione attesa.

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

Qui sotto riportiamo la matrice riguardante un sudoku generico 2x2:

	r1c1	r1c2	r2c1	r2c2	r1.1	r1.2	r2.1	r2.2	c1.1	c1.2	c2.1	c2.2
(1,1,1)	1	0	0	0	1	0	0	0	1	0	0	0
(1,1,2)	1	0	0	0	0	1	0	0	0	1	0	0
(1,2,1)	0	1	0	0	1	0	0	0	0	0	1	0
(1,2,2)	0	1	0	0	0	1	0	0	0	0	0	1
(2,1,1)	0	0	1	0	0	0	1	0	1	0	0	0
(2,1,2)	0	0	1	0	0	0	0	1	0	1	0	0
(2,2,1)	0	0	0	1	0	0	1	0	0	0	1	0
(2,2,2)	0	0	0	1	0	0	0	1	0	0	0	1

Selezioniamo la riga dell'elemento che fa già parte della soluzione ed effettuiamo un ricoprimento, rimuovendo le colonne che contengono un uno in corrispondenza degli 1 della riga selezionata (evidenziati in rosso nella tabella sottostante) e eliminando le righe che contengono un uno in

tale posizione. Quindi cancelliamo le righe in rosso e le colonne in rosso. Il colore verde evidenzia la riga selezionata e gli 1 presenti.

	r1c1	r1c2	r2c1	r2c2	r1.1	r1.2	r2.1	r2.2	c1.1	c1.2	c2.1	c2.2
(1,1,1)	1	0	0	0	1	0	0	0	1	0	0	0
(1,1,2)	1	0	0	0	0	1	0	0	0	1	0	0
(1,2,1)	0	1	0	0	1	0	0	0	0	0	1	0
(1,2,2)	0	1	0	0	0	1	0	0	0	0	0	1
(2,1,1)	0	0	1	0	0	0	1	0	1	0	0	0
(2,1,2)	0	0	1	0	0	0	0	1	0	1	0	0
(2,2,1)	0	0	0	1	0	0	1	0	0	0	1	0
(2,2,2)	0	0	0	1	0	0	0	1	0	0	0	1

	r1c2	r2c1	r2c2	r1.2	r2.1	r2.2	c1.2	c2.1	c2.2
(1,2,2)	1	0	0	1	0	0	0	0	1
(2,1,2)	0	1	0	0	0	1	1	0	0
(2,2,1)	0	0	1	0	1	0	0	1	0
(2,2,2)	0	0	1	0	0	1	0	0	1

Adesso possiamo iniziare con l'applicazione dell'algoritmo. Selezioniamo la terza colonna e proviamo a scegliere la riga che contiene la terna (2,2,2).

	r1c2	r2c1	r2c2	r1.2	r2.1	r2.2	c1.2	c2.1	c2.2
(1,2,2)	1	0	0	1	0	0	0	0	1
(2,1,2)	0	1	0	0	0	1	1	0	0
(2,2,1)	0	0	1	0	1	0	0	1	0
(2,2,2)	0	0	1	0	0	1	0	0	1

Usando le stesse operazioni di prima, otterremo:

r1c2	r2c2	r1.2	r2.1	c1.2	c2.1

Non è più possibile selezionare altre righe, nonostante non siano state ricoperte tutte le colonne (evidenziato dal fatto che non tutte le colonne sono state selezionate). Bisogna ripristinare una configurazione precedente. Adesso selezioniamo (2,2,1) invece che (2,2,2) e vediamo cosa succede:

	r1c2	r2c1	r2c2	r1.2	r2.1	r2.2	c1.2	c2.1	c2.2
(1,2,2)	1	0	0	1	0	0	0	0	1
(2,1,2)	0	1	0	0	0	1	1	0	0
(2,2,1)	0	0	1	0	1	0	0	1	0
(2,2,2)	0	0	1	0	0	1	0	0	1

Procedendo come prima, otteniamo:

	r1c2	r2c1	r1.2	r2.2	c1.2	c2.2
(1,2,2)	1	0	1	0	0	1
(2,1,2)	0	1	0	1	1	0

Selezionando queste altre due righe ottengo la soluzione del sudoku desiderata (la matrice diventa vuota e sono state selezionate tutte le colonne).

3.2 DLX: STRUTTURA DATI

Questa struttura dati è una matrice dove ogni elemento che ha un 1 al suo interno possiede 4 puntatori: uno che punta alla prima cella adiacente che contiene un uno per ogni possibile direzione: Up (U), Down (D), Left (L), Right (R). Se esaminiamo un elemento che si trova sull'ultima colonna, il puntatore R punta alla cella sulla stessa riga sulla prima colonna. Invece, se analizziamo una casella sulla prima colonna, il puntatore L riferisce l'elemento della stessa riga sull'ultima colonna. Similmente per quanto riguarda gli elementi della prima e ultima riga nelle direzioni Up e Down, ottenendo così una sorta di struttura circolare. Inoltre, se una casella che contiene un uno non indirizza nessun altro elemento, il puntatore riferisce se stesso. L'elemento nella riga più in alto - con l'indice di riga più basso - che contiene un uno e la casella più in basso - con l'indice di riga più alto -, punta anche ad un oggetto che identifica la colonna relativa della matrice. L'oggetto colonna possiede due campi: il campo che indica la dimensione della colonna, ovvero quanti uni sono presenti nella relativa colonna, e il nome della colonna - che è arbitrario - .
La cosa interessante di questa struttura è che è circolare, dove ogni elemento punta ed è riferito da un altro. Inoltre, se noi effettuiamo una serie di cancellazioni, se eseguiamo gli inserimenti in ordine inverso, allora possiamo recuperare la configurazione precedente e questo è perfetto per algoritmi di backtracking, come nel nostro caso. Prendiamo ad

esempio una struttura A dove ogni elemento punta all'elemento successivo a sinistra e a destra in maniera circolare. Definiamo le operazioni di cancellazione e di inserimento, considerando che L(X) e R(X) sono rispettivamente l'elemento a sinistra e a destra del nodo X[2]. Ogni nodo X rappresenta un elemento della matrice che contiene un 1.

$$L(R(X)) \leftarrow L(X) \quad R(L(X)) \leftarrow R(X) \qquad \text{(cancellazione)}$$

Non rimuoviamo così l'informazione presente in X; spostiamo semplicemente i puntatori che non riferiscono più X: risulterà non essere più presente nella struttura anche se in memoria è ancora presente. Questo ci consente di ripristinare l'informazione X modificando solamente due puntatori: [2]

$$L(R(X)) \leftarrow X \quad R(L(X)) \leftarrow X \qquad \text{(ripristino)}$$

Così ripristiniamo il dato precedentemente cancellato logicamente. Come detto in precedenza, è importante mantenere l'ordine inverso: se effettuiamo n cancellazioni, si riesce a ripristinare la configurazione iniziale se e solo se gli inserimenti avvengono in ordine contrario rispetto alle rimozioni. Per identificare le varie colonne, ognuna ha un oggetto identificatore che punta al primo uno presente all'interno della colonna. Oltre a questo, esiste un altro elemento che invece riporta le dimensioni di quest'ultima, dove per dimensioni intendiamo il numero di 1 presenti in questo momento in suddetta colonna. (Lo chiameremo Size, abbrevviato in S). Quindi, per sfruttare l'algortimo X, non viene utilizzata una semplice matrice. La particolarità di questa struttura è che ogni elemento della matrice che contiene un uno, punta all'oggetto più vicino che contiene un 1 in tutte e quattro le direzioni possibili. Se l'elemento è quello più vicino alla colonna, punta anche all'oggetto che identifica la colonna.

3.3 PSEUDOCODICE DELL'ALGORITMO X

Per implementare tale algoritmo, ci sono due operazioni che rappresentano il fulcro: quella di cover e quella di uncover. Nell'operazione di cover si cerca di rimuovere gli elementi che sono stati già esplorati e rimuovere le righe e o colonne che non rispettano la condizione del problema, in base alle colonne e righe che selezioniamo iterazione per iterazione, mentre l'operazione di uncover serve per ripristinare una configurazione precedente, e viene utilizzata se l'algoritmo si accorge che le scelte effettuate non portano ad una copertura di tale matrice. Riportiamo lo pseudocodice [2]:

```
cover(c){
   L(R(c)) = L(c)
   R(L(c)) = R(c)
   i = D(c)
   while(i not equal c){
      j = R(i)
      while(j not equal i){
         U(D(j)) = U(j)
         D(U(j)) = D(j)
         S(C(j)) = S(C(j)) - 1
         j = R(j)
      }
      i = D(i)
   }
}
```

Il paramtero c indica la colonna che è stata selezionata. Ricordiamo che ogni elemento punta agli elementi che contengono uno nelle quattro direzioni possibili. Quindi questo pseudocodice esegue i seguenti passaggi:

1. Con le prime due istruzioni, i vicini orizzontali non riferiscono più la colonna *c* (perché appunto è stata scelta e fa parte del ricoprimento).

2. Selezioniamo il primo elemento sottostante alla colonna c che contiene un uno

3. Siccome lavoriamo in una struttura circolare, quando l'elemento *i* che scorre tutte le righe inerenti alla nostra scelta è uguale a *c*, significa che abbiamo analizzato tutto quello che era necessario, (si sta quindi scorrendo la colonna c dall'alto verso il basso)

4. altrimenti, scorriamo la riga scelta verso destra, analizzando elemento per elemento, fino a quando non torniamo alla posizione *i* da cui siamo partiti (anche in questo caso sfruttiamo la circolarità)

5. per ogni elemento *j*, tale elemento non verrà più riferito dalla cella successivamente soprastante e sottostante e la dimensione della colonna diminuisce di 1.

6. Il processo finirà quando verranno analizzate tutte le righe prese in esame.

Per quanto riguarda la funzionalità di uncover, qui sotto viene riportato lo pseudocodice[2]:

```
uncover(c){
   i = U(c)
   while(i not equal c){
      j = L(i)
      while(j not equal i){
         S(C(j)) = S(C(j) + 1
         U(D(j)) = j
         D(U(j)) = j
         j = L(j)
      }
   }
   L(R(c) = c
   R(L(c) = c
}
```

Qui sfruttiamo l'operazione di undo che è stata precedentemente definita: infatti è possibile notare che, mentre si effettuava la fase di ricoprimento, gli elementi venivano rimossi seguendo l'ordine basso-destra, il re-inserimento di questi elementi viene fatto nel modo inverso (alto, sinistra). Le due operazioni nel ciclo while interno servono per ripristinare il riferimento all'elemento utilizzato, mentre i due assegnamenti finali ripristinano la presenza della colonna nella matrice che è nuovamente riferita dalle colonne adiacenti. Adesso bisogna costruire un metodo che sfrutti entrambi gli operatori: è facile intuire che l'approccio più semplice da adoperare è di tipo ricorsivo. Effettuo ricoprimenti parziali a cascata richiamando il metodo e, qualora non si ottenesse il risultato atteso, si effettua la fase di uncover e si prova ad effettuare decisioni diverse che, prima o poi, porteranno ad un ricoprimento della matrice. (Questo è vero se è presente almeno un ricoprimento). Verrà riportato lo pseudocodice che descrive il comportamento descritto sopra[2]:

```
search(h,k,s){
   if(R(h) = h){
      solution founded
      return
   }else{
      c = choose_a_column(h)
      r = D(c)
      while (r not equal c){
```

```
            add r to partial solution s
            j = R(j)
            while(j not equal r){
               cover(C(j))
               j = R(j)
            }
            search(h,k+1,s)
            r = remove the k-element to partial solution s
            c = C(r)
            j = L(r)
            while(j not equal r){
               uncover(C(j))
               j = L(j)
            }
            r = D(r)
         }
         uncover(c)
      }
   }
```

Nel codice qua sopra, nel primo ciclo for si scorre la riga selezionata (in questo caso si sceglie la prima disponibile della colonna). Poi, per ogni elemento che contiene un uno della riga si effettua l'operazione di cover. Successivamente si richiama ricorsivamente la funzione, passando *k+1* come indice del prossimo elemento da aggiungere alla soluzione parziale *s*. Se la chiamata *search(h,k+1,s)* non andasse a buon fine, allora bisognerà recuperare una configurazione precedente: quindi si procede alla fase di uncover per ripristinare la riga che era stata selezionata precedentemente ed infine anche la colonna. La variabile *h* contiene la testa della struttura della matrice: *h* punta alla prima e all'ultima colonna della matrice. *K* indica la posizione dell'elemento da aggiungere alla soluzione parziale *s*. In questo pseudocodice si può notare che, scelta la colonna, viene selezionata la prima riga inerente a suddetta colonna. Per selezionare l'elemento c esistono due criteri principali: o scegliere la prima colonna disponibile, oppure scegliere la colonna con un numero minore di 1 presenti. Quest'ultima è una scelta intelligente, in quanto, qualora avessimo effettuato una scelta sbagliata all'inizio, ci saranno solo poche altre possibilità disponibili. Se invece partissimo paradossalmente con una colonna con un grande numero di 1 presenti, allora se dovessimo ripartire ogni volta al punto di partenza perché abbiamo sbagliato elemento iniziale, ci sarebbe un grande overhead

in termine di tempo che poteva essere evitato se si fosse effettuata la decisione sopra descritta. Inoltre, nel caso sia presente una colonna con soli zeri, verrebbe selezionata subito e non si entrerebbe nel ciclo while più esterno, procedendo subito alla fase di uncover della colonna. Si ripristinerà poi la configurazione precedente. Il metodo search, alla prima chiamata, ha come parametri h,0, e s. La soluzione parziale inizialmente non contiene elementi.

3.4 DESCRIZIONE DEL SUDOKU

Ogni cella del sudoku può assumere 9 diversi valori (le celle sono 9x9 per un totale di 729 possibili configurazioni). La matrice binaria rappresentante questo tipo di problema è rappresentata dividendo le colonne per condizione. Bisogna rispettare 4 condizioni principali:

1. In ogni cella è presente un solo numero
2. In ogni riga sono presenti i numeri da 1 a 9 senza ripetizioni
3. In ogni colonna sono presenti i numeri da 1 a 9 senza ripetizioni
4. In ogni riquadro sono presenti i numeri da 1 a 9 senza ripetizioni

Le righe di tale matrice identificano il numero in cui viene posizionato l'elemento. Saranno disposte nel seguente modo (partendo dall'alto in basso):

- nella cella in posizione (1,1) è posizionato il numero 1
- nella cella in posizione (1,1) è posizionato il numero 2
  ...
- nella cella in posizione (1,2) è presente il numero 1
  ...
- nella cella in posizione (9,9) è presente il numero 9

Quindi in totale avremo una matrice con 729 righe. Le possibilità per condizione sono 81 (siccome il sudoku è formato da 81 celle), per un totale di 324 colonne. Descriviamo come costruire la matrice mediante tali condizioni.

3.4.1 *Condizione di cella*

In ogni cella può essere presente solamente un numero, quindi è necessario costruire una area della matrice in modo tale che, una volta selezionata la prima cella in alto a sinistra e una volta selezionato il numero che va inserito, bisogna escludere gli altri numeri possibili in quella cella. Lo stesso vale per tutte le altre 81 caselle del sudoku. Avremo quindi una parte della matrice in questo modo: (rXcY.Z indica che nella cella in posizione [X,Y] è presente il numero Z)

	r1c1	r1c2	...	r9c9
r1c1.1	1	0	...	0
r1c1.2	1	0	...	0
...	...	...	...	...
r1c1.9	1	0	...	0
r1c2.1	0	1	...	0
r1c2.2	0	1	...	0
...	...	...	...	...
r9c9.1	0	0	...	1
...	...	...	...	...
r9c9.9	0	0	...	1

Se selezioniamo la prima colonna e la prima riga, sicuramente, con la fase di copertura, escludiamo tutta la prima colonna e quindi tutti i possibili numeri che possono essere inseriti in quella casella. In pratica, con tale scelta, inseriamo l'uno all'interno della cella in posizione [1,1] e quindi, in quella casella, non possono essere presenti altri numeri. La matrice così impostata tiene conto di suddetta condizione.

3.4.2 *Condizione di riga*

Per le righe invece, se ad esempio analizziamo la prima riga, il numero uno può essere inserito in 9 posizioni differenti. Quindi, avremo questa porzione di tabella:

	r1.1	r1.2	...	r1.9	r2.1	...
r1c1.1	1	0	...	...	...	...
r1c1.2	0	1	...	...	...	...
...	...	...	...	...	...	...
r1c1.9	0	0	...	1	...	...
r1c2.1	1	0	...	0	0	...
...	...	...	...	...	...	...
r2c1.1	1	0	...	...	...	...
r2c1.2	0	1	0	...	...	...
...	...	...	...	...	...	...

In questo caso è evidente notare che se selezioniamo la casella [1,1] e ci inseriamo l'1, allora escludiamo tutti gli 1 dalla riga (r1c2.1 fino a r1c9.1 vengono esclusi perché si trovano nella stessa riga dove è già presente un 1)

3.4.3 *Condizione di colonna*

Per controllare le colonne, si ha una struttura di questo tipo:

	c1.1	c1.2	...	c1.9	c2.1	...
r1c1.1	1	0	...	...	...	...
r1c1.2	0	1	...	...	...	...
...	...	...	...	...	...	...
r1c1.9	0	0	...	1	...	...
r1c2.1	0	0	...	0	1	...
...	...	...	...	...	...	...
r2c1.1	1	0	...	...	...	...
r2c1.2	0	1	...	...	...	...
...	...	...	...	...	...	...
r9c1.1	1	0	...	...	...	...
...	...	...	...	...	...	...

Come per la condizione di riga, i numeri uno nella matrice vengono disposti sulla diagonale: in questo caso però la diagonale è formata da 81 elementi uguali ad 1, mentre nel caso della riga la diagonale è

composta da 9 elementi (questo perché, ad ogni incremento di riga del sudoku, si ricomincia a inserire gli 1 a partire dalla prima colonna della matrice binaria). E' immediato osservare che sulla tabella così composta, se inseriamo nel sudoku l'elemento 1 in posizione [1,1], allora, tramite la procedura di ricoprimento, verrà esclusa la colonna che contiene un un uno sulla riga r2c1.2: e così escluderemo anche gli altri uni nella medesima colonna (fino a r9c1).

3.4.4 *Condizione di riquadro*

Per descrivere la condizione di riquadro tramite una matrice binaria, facciamo un esempio. Come si possono rappresentare tutte le possibili posizioni del numero 1 nel primo riquadro? Tale numero può andare nelle seguenti celle: $[r_1c_1, r_1c_2, r_1c_3, r_2c_1, r_2c_2, r_2c_3, r_3c_1, r_3c_2, r_3c_3]$. Quindi gli 1 verranno disposti in modo tale che, una volta selezionata la riga r_1c_1, allora verranno rimosse le righe $r_1c_2, r_1c_3, r_2c_1, r_2c_2, r_2c_3, r_3c_1, r_3c_2, r_3c_3$. Ogni colonna rappresenta l'inserimento di un numero X all'interno del riquadro Y. (Y.X) Riportiamo parte della condizione descritta sulla prima colonna:

	Riquadro1.1	Riquadro1.2	...	...	...	...
r1c1.1	1	0	...	...	...	...
r1c1.2	0	1	...	...	...	...
...	...	...	...	...	...	...
r1c2.1	1	0	...	...	...	...
r1c2.2	0	1	...	...	...	...
r1c2.3	0	0	...	...	...	...
...	...	...	...	...	...	...
r1c3.1	1	0	...	...	...	...
...	...	...	...	...	...	...
r3c3.1	1	0	...	...	...	...
...	...	...	...	...	...	...

Sulla seconda colonna si analizza sempre il primo riquadro, ma si verifica il posizionamento del 2.

3.5 CONCLUSIONE

La combinazione di queste tabelle crea il nostro problema del sudoku trasposto in una matrice binaria e, trovata copertura di tale struttura, è possibile risalire alla soluzione del sudoku.

	Cella	Riga	Colonna	Riquadro
r1c1.1	...	...	...	...
...	...	...	...	...
r9c9.9	...	...	...	...

Ovviamente alcune colonne e righe sono già ricoperte dai numeri iniziali, quindi bisogna effettuare una fase di scrematura prima di procedere all'algoritmo vero e proprio, inserendo già nella soluzione parziale le righe che rappresentano i numeri già inseriti. (Ed effettuando gli eventuali ricoprimenti). Però esiste un problema: dobbiamo tenere traccia degli elementi che andiamo a scegliere iterazione per iterazione, per poter poi risalire al problema del sudoku. Per risolvere questo incoveniente, è sufficiente che ogni elemento della matrice contenga al suo interno l'informazione del posizionamento dell'elemento all'interno dello schema. Una volta ottenuta una soluzione, possiamo riordinare la matrice in base alla posizione di riga: alla fine dell'algoritmo avremo una matrice con 81 righe che ci permetterà di completare l'inserimento dei numeri all'interno del sudoku.

4

RISOLUZIONE DEL SUDOKU CON FUNZIONE OBIETTIVO

Gli approcci dei metodi che abbiamo utilizzato in precedenza sfruttano il backtracking per raggiungere alla soluzione desiderata: se dopo aver analizzato tutte le possibili configurazioni non ci sono soluzioni, allora lo schema non è risolvibile. Esiste anche un altro approccio, che prevede l'utilizzo di una funzione obiettivo da minimizzare per rappresentare il problema in questione e, tramite appositi algoritmi e sfruttando degli operatori particolari, si cerca di avvicinarsi il più possibile alla soluzione ottima, passando attraverso dei minimi locali.

4.1 METODO BETA-HILL CLIMBING

Il metodo beta-hill climbing incomincia tramite una soluzione generata in maniera completamente casuale. Se minimizza la funzione obiettivo, allora la soluzione allo schema è stata trovata. Altrimenti, sfrutta due operatori per avvicinarsi alla soluzione: uno che ha il compito di esplorare soluzioni adiacenti a quella corrente (in un insieme di celle il valore viene incrementato o decrementato di uno), mentre l'altro ha lo scopo di randomizzare la soluzione generata (un numero che si è provato ad inserire viene rigenerato in maniera causale). Questi operatori hanno una determinata probabilità di entrare in azione, che è stabilita dal programmatore.
Si confronta successivamente la soluzione con la funzione obiettivo: se la soluzione trovata è minore rispetto a quella precedentemente descritta, allora si aggiorna la soluzione, altrimenti si continua ad iterare. La funzione obiettivo utilizzata è la seguente:[3]

$$\min z = \sum_{i=1}^{9} \left| \sum_{j=1}^{9} x_{ij} - 45 \right| + \sum_{j=1}^{9} \left| \sum_{i=1}^{9} x_{ij} - 45 \right| + \sum_{k=1}^{9} \left| \sum_{(l,m)\in B} x_{lm} - 45 \right| \quad (4.1)$$

Dove gli indici l ed m sono gli indici utilizzati per iterare i vari blocchi. La formula scritta qua sopra è in grado anche di rilevare la condizione di non ripetizione del sudoku. Questo perché, in maniera implicita, siccome il gioco deve contenere i numeri da 1 a 9 in ogni riquadro, riga e colonna, allora la somma dei numeri nella soluzione per ogni riga colonna e riquadro sarà sempre uguale a 45. Visto che sottraiamo 45 ad ogni area di interesse, la funzione obiettivo assume valore zero quando il posizionamento dei numeri nello schema è corretto.

5

ALGORITMO BRUTE FORCE CON TECNICHE LOGICHE

Questo tipo di algoritmo è il più intuitivo da implementare. Consiste nel rappresentare il sudoku come una matrice di celle. Ciascuna cella ha al suo interno un vettore di candidati, dove i candidati indicano i possibili numeri che possono essere inseriti in quella determinata casella, controllando le condizioni del sudoku. Nella nostra implementazione, ogni cella ha un attributo finalNumber: tale numero significa che nella griglia è stata effettivamente inserita tale cifra. Il nostro codice risolve un qualunque sudoku *nxn*.
Prima di addentrarci in alcuni dettagli implementativi, definiamo le tecniche che sono state utilizzate per risolvere lo schema:

1. Se in una cella è presente un solo candidato, allora inseriamo quel candidato nella soluzione finale.

2. Se in un settore (riga, colonna, riquadro), è presente solo una volta un candidato, allora inseriamo tale numero.

3. Se esiste una coppia, una terna, una n-esima configurazione di celle su uno stesso settore, che hanno gli stessi n candidati, allora dal settore di interesse possiamo rimuovere gli n candidati da tutte le altre celle

4. Se esiste una coppia o terna di candidati uguali presenti in un riquadro tutti su una stessa riga o su una stessa colonna, allora possiamo rimuovere tale candidato dalla riga e o colonna considerata.

Abbiamo costruito una classe astratta SudokuTechnique: ogni sua classe figlia dovrà implementare il metodo apply che implementerà una tecnica specifica di risoluzione in base al tipo dell'oggetto. Abbiamo quindi costruito quattro diverse classi, dove ciascuna di esse implementa una delle strategie che abbiamo precedentemente descritto.

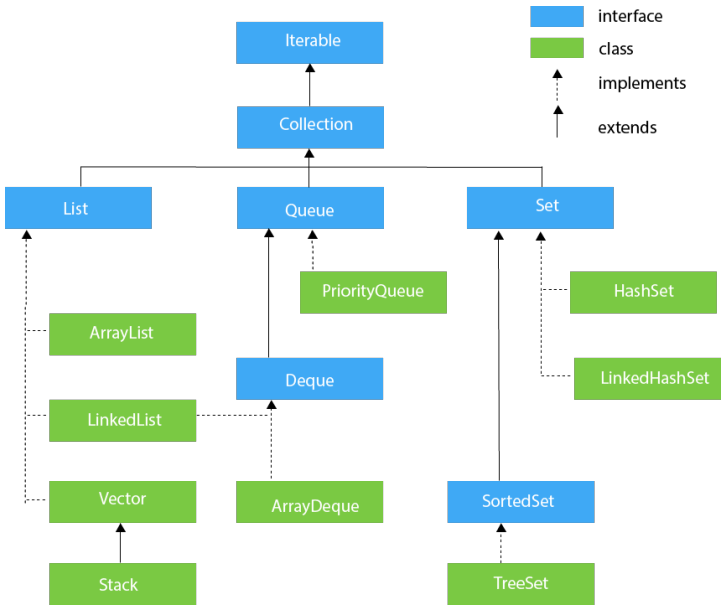

Figura 4.: Immagine presa da: `https://www.javatpoint.com/collections-in-java`

5.1 ALCUNE INFORMAZIONI RIGUARDO JAVA

In questa sezione verrà spiegato brevemente il linguaggio Java e vengono descritte alcune sue strutture che vengono adoperate nel codice qua sotto in parte presentato. [8]

5.1.1 *Collection e alcuni oggetti utilizzati*

In Java è presente una gerarchia di classi che riguardano le collezioni di oggetti. Tale gerarchia è mostrata dalla figura. 4 Noi abbiamo scelto di usare principalmente ArrayList per rappresentare vettori di dimensione variabile.

5.1.2 *Java Stream*

Le stream Java permettono di visitare gli elementi di una Collection di dati, specificando cosa vogliamo fare su quelle determinate informazioni, senza specificare il come. Permettono numerose operazioni, come l'operazione di filtraggio dei dati tramite il metodo filter che, grazie ad una espressione lambda passata come parametro, filtra determinati elementi della lista di informazioni. Le espressioni lambda sono, in sostanza, dei

metodi che possono essere passati come parametri a delle funzioni. Anch'essi come le funzioni hanno dei valori di ritorno e in Java ne esistono di diversi tipi.
Su una stream è possibile mappare le informazioni, inserendole in un oggetto di tipo Map, dove ogni elemento di tipo Entry è rappresentato dalla coppia chiave-valore. Solitamente, nel codice da noi utilizzato, la chiave indica il numero del candidato mentre il valore rappresenta quante volte tale candidato è apparso nella Map. Per raccogliere le informazioni in una Map raggruppate per un certo dato, è necessario richiamare su uno stream il metodo collect e passargli come parametro Collectors.groupingBy dove, a sua volta, come parametro gli si passa il fattore di raggruppamento. Metodi come findFirst ci consentono di ritornare il primo oggetto della stream, se è presente. Nel caso non fosse presente, tale metodo ritorna un Empty Optional (viene utilizzato questo stratagemma per evitare in parte il NullPointerException). Si usa il metodo get per ottenere l'oggetto incapsulato dentro l'Optional desiderato.

5.2 CELLE CON UN SOLO CANDIDATO

E' facile constatare che se è possibile inserire un solo numero all'interno di una cella, nella soluzione finale sarà presente quella cifra in maniera forzata.

```
@Override
public void apply(Sudoku sudoku) {
  for (Cell c : sudoku) {
    if (c.onlyOneCandidateIsPresent()) {
      int finalNumber = c.getCandidates().get(0);
      sudoku.addNumberToSudokuSolution(finalNumber, c);
      removeAllUselessCandidates(sudoku);
    }
  }
}
```

Questo metodo implementa tale logica, inserendo i numeri nelle celle che contengono un solo candidato.
RemoveAllUselessCandidate serve per rimuovere tutti i candidati che non possono essere inseriti in alcune celle dello schema a causa dei numeri piazzati all'interno della griglia. Tale metodo viene richiamato dopo la creazione del sudoku (in quanto già sono presenti alcuni elementi

all'interno della matrice 9x9). La logica è semplice: scorro tutte le celle, e se è stato settato il numero in tale cella (per rappresentare questa situazione adoperiamo l'attributo finalNumber della classe che identifica la cella) allora mi preoccupo di rimuovere il candidato di quella cella in ogni riga, colonna e riquadro in modo tale da rispettare la non ripetizione dei numeri inseriti nello schema.

5.3 NUMERI OBBLIGATI

Un numero si dice obbligato se e solo se, all'interno di una determinata riga, colonna e riquadro, un numero si può inserire solamente in una determinata cella.

```
@Override
public void apply(Sudoku sudoku) {
  int dimension = sudoku.getDimension();
  for (int j = 0; j < dimension; j++) {
    ArrayList<Cell> cellsRow = sudoku.getCells().get(j);
    List<Integer> numbersUnique = findUniqueNumbers(cellsRow);
    putAllUniqueNumbersInRow(numbersUnique, j, sudoku);
    ArrayList<Cell> cellsColumn =
        SudokuUtilities.getCellsColumn(sudoku,j);
    numbersUnique = findUniqueNumbers(cellsColumn);
    putAllUniqueNumbersInColumn(numbersUnique, j, sudoku);
    findAndInsertAllBoxUniqueNumbers(j, sudoku);
  }
}
```

Questa è una possibile implementazione Java di tale strategia. Come vettore di dimensioni variabili abbiamo deciso di sfruttare l'ArrayList di Java.
FindUniqueNumbers ha il compito di trovare tutti i numeri che sono unici in una determinata lista di elementi (riga, colonna e riquadro). Per analizzare righe, colonne e riquadri utilizziamo delle strutture di appoggio (ArrayList di Cell), rispettivamente: cellsRows, cellsColumn e nel metodo insertAllBoxUniqueNumbers cellsBox. Dopo aver trovato i numeri unici, quest'ultimi poi vengono inseriti nello schema dai metodi putAllUniqueNumbersInRow e putAllUniqueNumbersInColumn. Per quanto riguarda i riquadri, siccome la logica è maggiormente complessa, allora viene tutto svolto nel metodo a parte findAndInsertAllBoxUnique-Numbers.

Per poter trovare i numeri unici, viene sfruttata una stream che viene qui sotto riportata:

```
private List<Integer> findUniqueNumbers(ArrayList<Cell> array) {
     return array.stream().filter((c) ->
         c.isNotCellAlreadyOccupied())
          .flatMap((c) -> c.getCandidates().stream())
          .collect(Collectors.groupingBy(Function.identity(),
              Collectors.counting())).entrySet().stream()
          .filter(c -> c.getValue() == 1).map(c ->
              c.getKey()).collect(Collectors.toList());
   }
```

Questo metodo prende tutte le celle che non sono occupate, prende tutti i candidati presenti e li inserisce in un unico stream con flatMap, poi successivamente li raggruppa per numero. Vengono selezionati solo gli elementi che sono contati una sola volta, ovvero i candidati unici. Per quanto riguarda i riquadri invece:

```
private void findAndInsertAllBoxUniqueNumbers(int box, Sudoku
    sudoku) {
     int[][] boxindexes = getBoxManagement()
     .getBoxIndexes(BoxManagement.boxes[box][0],
         BoxManagement.boxes[box][1]);
     int[] irows = boxindexes[0];
     int[] jcolumns = boxindexes[1];
     ArrayList<Cell> cellsBox =
         getBoxManagement().getCellsBox(sudoku, irows, jcolumns);
     List<Integer> numbersUnique = findUniqueNumbers(cellsBox);
     insertUniqueNumbersInBox(sudoku, irows, jcolumns,
         numbersUnique);
}
private void insertUniqueNumbersInBox(Sudoku sudoku, int[] irows,
    int[] jcolumns, List<Integer> numbersUnique) {
     for (int z = 0; z < numbersUnique.size(); z++) {
       int finalNumber = numbersUnique.get(z);
       for (int i = 0; i < 3; i++) {
         for (int j = 0; j < 3; j++) {
           Cell cell =
               sudoku.getCells().get(irows[i]).get(jcolumns[j]);
           if (cell.getCandidates().contains(finalNumber)) {
             sudoku.addNumberToSudokuSolution(finalNumber, cell);
             removeAllUselessCandidates();
```

```
                return;
            }
          }
        }
      }
    }
```

Il parametro j indica il j-esimo riquadro da analizzare (numerati da 0 a 8, seguono ordine sinistra-destra, alto-basso).
E' necessario avere un metodo per poter esplorare tutte le celle del riquadro. Per farlo ci serviamo di due array che vengono ritornati dal metodo getBoxIndexes. Questi vettori contengono al loro interno tutte le posizioni delle celle relative al riquadro che stiamo analizzando. Se stessimo analizzando il secondo settore, avremo due vettori con rispettivamente i valori 0,1,2 e 3,4,5. Per poter iterare su questa matrice 3x3, sarà sufficiente avere un doppio ciclo for annidato con gli indici i e j che vanno da 0 a 2. Per prelevare poi la posizione effettiva della casella sarà sufficiente prendere il valore i-esimo del primo vettore per comprendere la posizione di riga, j-esimo del secondo vettore per comprendere la posizione della colonna. La classe BoxManagement ha il compito di implementare questo tipo di metodo.
La logica poi è la stessa dei metodi precedenti: trovo gli elementi unici in una riga, colonna o riquadro e li inserisco. (Operazione che viene svolta dal metodo insertUniqueNumbersInBox).

5.4 COPPIE, TERNE, POKER, N-UPLE DI CANDIDATI

A differenza delle due precedenti, non consente di inserire direttamente un numero, bensì di escludere determinati candidati da alcune celle dello schema. Se in una riga, colonna o riquadro, è presente una coppia, una terna o un poker di candidati ripetuti in rispettivamente due, tre e quattro celle, allora possiamo escludere tutti questi candidati dalle altre caselle che si trovano nella stessa riga, colonna o riquadro. In realtà è possibile applicare la logica anche per n-celle e n candidati, ma le situazioni più frequenti nella risoluzione del sudoku sono quelle fino a quattro celle. Qui di seguito viene riportato un esempio per aiutare il lettore a comprendere tale strategia. L'esempio è stato preso dal sito di Andrew Stuart [6].
Analizziamo la figura 5: i numeri in piccolo rappresentano gli unici candidati che possono essere inseriti in quella cella. Si può constatare che nella prima riga e primo riquadro è presente una coppia di celle, quelle

evidenziate in verde, che contengono gli stessi due candidati. Quindi, sia nel primo riquadro, sia nella prima riga, non possono essere inseriti i candidati 1 e 6 se non nelle due celle evidenziate in verde. I candidati in rosso possono essere rimossi.
Per trovare questo tipo di configurazione, si analizzano riquadri, righe e colonne e si conta, sfruttando una HashMap, quante celle con gli stessi candidati ci sono: se in un determinato settore d'interesse ci sono n celle con n candidati in comune, allora in tutte le altre 9-n celle del settore possono essere rimossi gli n candidati. Qui sotto viene riportato il codice che consente di trovare le celle che rispondono alla condizione sopra descritta e che rimuove i candidati dallo schema. L'indice i rappresenta il numero del riquadro, o della riga o della colonna dove dobbiamo eliminare i candidati. Il metodo generateCellMap crea un Hash Map che conta quante celle con lo stesso numero di candidati sono presenti[7].

```
private void findNTuple(Sudoku sudoku, int i, Cell[] cellsApp,
    String removeWhere) {
     Map<Cell, Integer> cellAndCount = generateCellMap(cellsApp);
     Set<Entry<Cell, Integer>> entrySet = cellAndCount.entrySet();
     for (Entry<Cell, Integer> entry : entrySet) {
       if (entry.getValue() > 1) {
         if (entry.getKey().getCandidates().size() ==
             entry.getValue()) {
           removeAllCandidatesFrom(sudoku, i, removeWhere,
               entry.getKey().getCandidates());
         }
       }
     }
  }
```

Removewhere è una stringa che consente al metodo removeAllCandidatesFrom di capire dove vanno rimossi quei candidati, se dalla riga, dalla colonna o dal riquadro i-esimo. Se sono presenti n celle con gli stessi n candidati, allora abbiamo trovato la condizione per sfruttare la tecnica.

5.5 COPPIE O TERNE CHE PUNTANO IN UNA DIREZIONE

Se all'interno di un riquadro, è presente un candidato che si può inserire solamente in una determinata riga o colonna, non ci interessa sapere dove verrà inserito il numero, in ogni caso il candidato che si trova sulla stessa riga o colonna ma riquadro differente non potrà essere inserito, perché

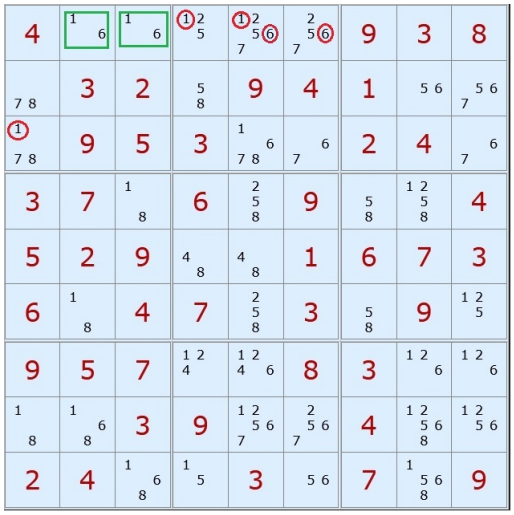

Figura 5.: Schema di esempio dove è possibile applicare la tecnica

altrimenti impedirebbe l'inserimento di tale candidato nel riquadro di partenza. Per capire meglio verrà visualizzato un esempio preso dal sito[6](Vedi figura 6). La classe PointingSudokuTechnique ridefinisce il metodo apply della classe padre SudokuTechnique per eseguire la tecnica precedentemente descritta.

```
 @Override
public void apply(Sudoku sudoku) {
  int dimension = sudoku.getDimension();
  for (int i = 0; i < dimension; i++) {
    for (int number = 1; number <= dimension; number++) {
      ArrayList<String> pos =
          positionsOfCandidateWhereInBox(sudoku,i, number);
      if (!pos.isEmpty() &&
          everyElementStartEndWithSameDigit(pos, true)) {
        removePairsRow(sudoku,i, number,
            getRowColumnCandidate(sudoku,i, number)[0]);
      } else if (!pos.isEmpty() &&
          everyElementStartEndWithSameDigit(pos, false)) {
        removePairsColumn(sudoku,i, number,
            getRowColumnCandidate(sudoku,i, number)[1]);
      }
    }
  }
}
```

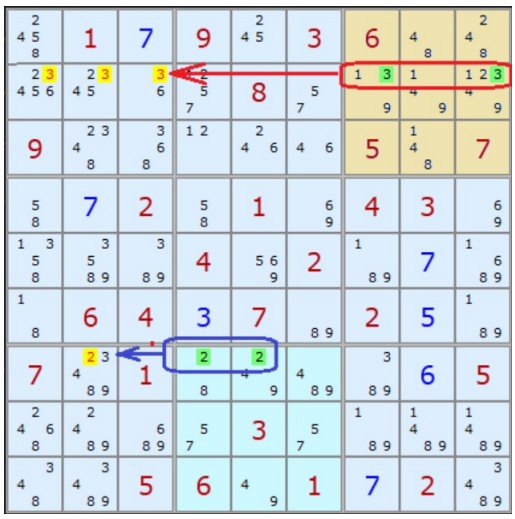

Figura 6.: Le celle con i numeri in verdi puntano verso le celle con i numeri in rosso: tali candidati possono essere eliminati

Per ogni riquadro i-esimo si controlla tutti i numeri da 1 a 9 presenti all'interno del riquadro. Se troviamo un candidato all'interno di un riquadro che si trova posizionato solo su celle che si trovano sulla stessa riga o sulla stessa colonna si può applicare la tecnica (il metodo positionOfCandidateWhereInBox ritorna una lista di stringhe che identificano la posizione del candidato number (scritta nel formato "rowcolumn"), mentre everyElementStartEndWithSameDigit controlla se si trova tale candidato su una sola riga del riquadro o su una sola colonna. Se al metodo everyElementStartEndWithSameDigit si passa come parametro il valore true allora si controlla la condizione di riga, se si passa false, si controlla quella di colonna. Il metodo getRowColumnCandidate serve per sapere il candidato analizzato su cui possiamo applicare la tecnica in quale riquadro si trova. Ritorna quindi un vettore bidimensionale riga-colonna. Quando rimuoviamo i candidati della riga prendiamo il primo elemento dell'array bidimensionale, quando invece li rimuoviamo dalla colonna, estraiamo il secondo elemento. Il metodo removePairsRow e removePairsColumn serve per rimuovere effettivamente i candidati che rispettano la condizione della tecnica. (Evitando di rimuovere anche quelli del settore di riferimento grazie al metodo getRowColumnCandidate).

5.6 ALGORITMO PROPOSTO

L'algoritmo proposto prova ad utilizzare queste tecniche un numero arbitrario di volte, deciso prima dell'esecuzione: se non si riesce a trovare la soluzione con quest'ultime, allora si procede a forzare lo schema. Per implementare questo tipo di meccanismo è necessario avere un metodo di backtracking: questo perché, se provassimo a inserire un numero ma ci accorgiamo che tale cifra rende lo schema non risolvibile, allora significa che abbiamo sbagliato un inserimento e dobbiamo provare un altro tipo di configurazione. L'algoritmo di forza bruta generico prevede di selezionare la prima cella e di inserire il primo numero disponibile dalla lista dei candidati (ordinati in ordine crescente di valore). Questo tipo di approccio però presenta un problema non indifferente: se nella soluzione finale, ad esempio, è presente una riga -inizialmente senza elementi- 9,8,7,6,5,4,3,2,1 tale risolutore impiega moltissimo tempo per risolvere questo schema. Questo perché, nella prima cella il risolutore prova prima ad inserire il numero 1, poi il 2 e così via: in ogni cella della riga quindi, prima di inserire il numero corretto, si inseriscono tutti gli altri numeri e, prima di verificare che una determinata cifra è effettivamente sbagliata, altri elementi potrebbero essere inseriti nelle celle successive -nel caso peggiore fino a riempire quasi tutto lo schema- e questo comporterebbe un grosso costo in termini temporali sull'algoritmo. Una accortezza potrebbe essere quella di iniziare a inserire numeri da quelle celle che hanno un minor numero di candidati, così da ridurre la possibilità di incorrere in questo tipo di scenario. Per implementare tale strategia, basta ordinare le celle del sudoku in ordine crescente per numero di possibili candidati inseribili e prendere il primo.
Per rendere ancora più performante l'algoritmo, dopo l'inserimento dell'elemento selezionato dalla cella con il minor numero di candidati possibili, inseriamo altri numeri a cascata e rimuoviamo candidati adoperando le tecniche che abbiamo visto nelle sottosezioni precedenti. Alla fine, stiamo facendo un qualcosa che è molto simile a quello che avrebbe fatto il risolutore SAT per il sudoku: anche in quel caso, quando si inseriva un numero nello schema, venivano forzati alcuni assegnamenti per rendere l'espressione soddisfacibile (per vedere spiegazione dell'utilizzo del risolutore SAT vedere appendice). Per applicare la tecnica di forza bruta è necessario creare una copia dello schema ogni volta che si posiziona un numero: questo perché bisogna fare un salvataggio della configurazione del sudoku precedente (nel caso avessimo inserito un numero errato). Per fare ciò, ogni classe di interesse implementa il metodo clone, così

che si possa effettivamente clonare l'istanza del sudoku. E' necessario salvare anche la scelta precedente riguardante una determinata cella: tale impostazione viene salvata nella variabile cellSettage. Così, se un numero inserito è sbagliato, si recupera la configurazione di cella precedente e si prova a inserire il primo numero disponibile. Altrimenti, se nessun numero va bene, ripristiniamo una configurazione di sudoku e di cella precedente.

5.7 ALCUNE ALTRE TECNICHE POSSIBILI

Qui sotto verranno riportate alcune tecniche che potrebbero essere aggiunte all'algoritmo proposto per renderlo ancora più veloce e performante. Esso si ispira a quello utilizzato dal sito [6].

5.7.1 *X-Wing*

La tecnica X-wing è una tecnica risolutiva molto intuitiva che può essere sfruttata per i sudoku più difficili. L'idea alla sua base è semplice: bisogna trovare un candidato[1] che compaia esattamente due volte in due righe, inoltre i due candidati si devono trovare nelle stesse colonne. Nella figura 7 abbiamo un esempio. Nello schema sono indicate solamente le posizioni possibili del 2 - dove non è presente un due, il due non può essere inserito -. La griglia è vuota per semplicità di esposizione. I due possono trovarsi solo in due punti nella seconda e penultima riga. Esaminiamo la seconda: il 2 potrebbe andare solamente nella prima o nell'ultima colonna. Cosa succederebbe se il due venisse inserito nel riquadro giallo in alto a sinistra? Siccome nella penultima riga il due può essere inserito solamente nella prima e nell'ultima colonna, dovrà essere inserito necessariamente in quest'ultima - altrimenti ci sarebbero due due nella prima colonna - . Ora verifichiamo cosa potrebbe succedere se il due venisse inserito nell'ultima colonna della seconda riga: in modo analogo, nella penultima riga, il 2 potrebbe andare solamente nella prima colonna. Queste sono le uniche due configurazioni possibili, in quanto abbiamo esaminato entrambe le posizione possibili del due nella seconda riga: o vale una o vale l'altra. In ogni caso, nelle posizioni in rosso, non può essere presente il 2. Questa tecnica funziona anche nel caso in cui

1 Ogni casella del sudoku ha una lista di candidati. Questi candidati rappresentano i numeri che possono essere inseriti all'interno della cella, rispettando nella configurazione attuale le condizioni del sudoku

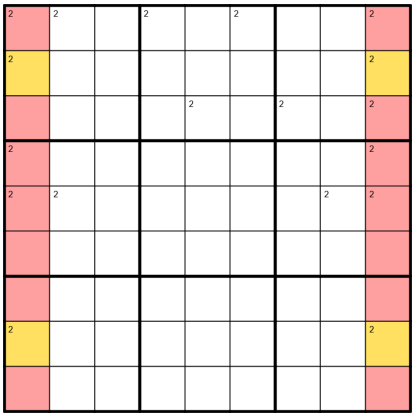

Figura 7.: I 2 contenuti nelle celle in rosso possono essere eliminati dallo schema

un candidato compaia esattamente due volte in due colonne diverse, a condizione che il candidato in questione si trovi sulle stesse righe. Il pattern viene chiamato così perché il candidato può andare in alto a sinistra e in basso a destra oppure in alto a destra e in basso a sinistra, formando una specie di X.

5.7.2 *Univocita'*

Questa tecnica si basa sul presupposto che lo schema che stiamo analizzando abbia una singola soluzione. Come nella sezione precedente, illustriamo la spiegazione con un esempio pratico. Nelle celle colorate sono inseriti gli unici numeri possibilmente inseribili dello schema. Cosa succederebbe se nella cella evidenziata in rosso non fosse inserito il 7? Avremmo una configurazione di 4 celle dove ciascuna di esse è l'angolo di una figura rettangolare - contenenti solo due possibili numeri -. Questa configurazione sembrerebbe non portare contraddizioni con le regole del sudoku. Invece il non inserimento del 7 nella cella in rosso, porterebbe questo schema ad avere due soluzioni. Infatti, scrivendo tali valori nella matrice 2x2 (rappresentante le celle colorate) otteniamo due configurazioni possibili:

$$\begin{bmatrix} 2 & 4 \\ 4 & 2 \end{bmatrix} \begin{bmatrix} 4 & 2 \\ 2 & 4 \end{bmatrix}$$

In ogni caso, lo schema non presenterebbe quindi contraddizioni, e questo possiamo dirlo perché tali elementi sono disposti in maniera rettangolare. Ogni schema dove sono presenti quattro celle con due

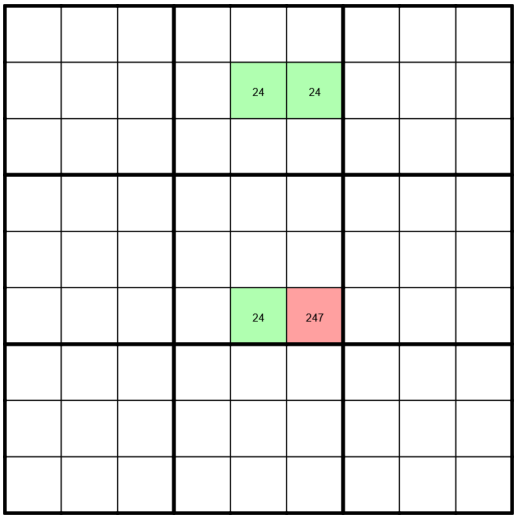

Figura 8.: Il 7 deve essere obbligatoriamente inserito nella cella in rosso, altrimenti lo schema ha due soluzioni

candidati in comune (disposte in maniera rettangolare), e tre di queste caselle hanno esattamente due candidati e l'altra ne contiene in più, è possibili rimuovere i due candidati in comune dalla cella che contiene più candidati (se il sudoku ha una soluzione unica, come dovrebbe essere).

BIBLIOGRAFIA

[1] Gihwon Kwon, Himanshu Jain - *Optimized CNF Encoding for Sudoku Puzzles* - Anno 2006 - (Citato nelle pagine 6, 11, 12, 13, e 14.)

[2] Mattias Harryson, Hjalmar Laestander - *Solving Sudoku efficiently with Dancing Links* - KTH Computer Science and Communication - Anno 2016 - (Citato nelle pagine 6, 15, 22, e 24.)

[3] Mohammed Azmi Al-Betar Mohammed A. Awadallah, Asaju La'aro Bolaji Basem O. Alijla - *ß-Hill Climbing algorithm for sudoku game* - Palestinian International Conference on Information and Communication Technology (PICICT) - Anno 2017 - (Citato nelle pagine 6 e 31.)

[4] `https://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X` - Data consultazione 15/09/2022 - (Citato a pagina 15.)

[5] `https://en.wikipedia.org/wiki/Exact_cover` - Data consultazione 15/09/2022 -

[6] Andrew Stuart - *Sudoku Solver by Andrew Stuart* - `https://www.sudokuwiki.org/sudoku.htm` - Data consultazione 15/09/2022 - (Citato nelle pagine 6, 38, 40, e 43.)

[7] `https://javarevisited.blogspot.com/2015/06/3-ways-to-find-duplicate-elements-in-array-java.html#axzz7cxbZjmBI` - Data consultazione 15/09/2022 - (Citato a pagina 39.)

[8] Cay S. Horstmann (edizione italiana a cura di Lorenzo Bettini) - *Java Per Impazienti* - (Citato a pagina 34.)

[9] Nieuwenhuis Robert, Oliveras Albert, Tinelli Cesar - Abstract DPLL and Abstract DPLL Modulo Theories - Proceedings Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning - Anno 2004 - Pagine 36–50 - (Citato a pagina 77.)

[10] Fortnow, Lance - The status of the P versus NP problem - Communications of the ACM - Anno 2009 - Volume 52 - Pagine 78-86 - Doi 10.1145/1562164.1562186 - (Citato a pagina 6.)

[11] Yato Takayuki, Seta Takahiro - Complexity and Completeness of Finding Another Solution and Its Application to Puzzles - IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences - E86-A - Anno 2003 - (Citato a pagina 5.)

A

ALCUNE IMPLEMENTAZIONI JAVA

A.1 ALGORITMO BRUTE-FORCE OTTIMIZZATO

```java
package sudoku_nxn;

import java.util.ArrayList;

public class BoxManagement {

  private int dimension;

  public BoxManagement(int dimension) {
    this.dimension = dimension;
  }

  public int[][] getBoxes() {
    int[][] boxes = new int[dimension][2];
    int sqrtDimension = (int) Math.sqrt(dimension);
    int rowElement = 0;
    int colElement = 0;
    for (int i = 0; i < dimension; i++) {
      boxes[i][0] = rowElement;
      boxes[i][1] = colElement;
      if (colElement + sqrtDimension >= dimension) {
        rowElement += 3;
        colElement = 0;
      } else {
        colElement += 3;
      }
    }
    return boxes;
  }
```

```
public int[][] getBoxIndexes(int row, int col) {
  int sqrtDimension = (int) Math.sqrt(dimension);
  int[][] positions = createPositionsArray(sqrtDimension);
  int[][] boxindexes = new int[2][sqrtDimension];
  boxindexes[0] = positions[row / sqrtDimension];
  boxindexes[1] = positions[col / sqrtDimension];
  return boxindexes;
}

private int[][] createPositionsArray(int sqrtDimension) {
  int[][] positions = new int[sqrtDimension][sqrtDimension];
  int array = 0;
  int posInArray = 0;
  int number = 0;
  while (number < dimension) {
    positions[array][posInArray] = number++;
    if (posInArray >= sqrtDimension - 1) {
      array++;
      posInArray = 0;
    } else {
      posInArray++;
    }
  }
  return positions;
}

public void removeBox(int[] irows, int[] jcolumns, int
    finalNumber, Sudoku sudoku) {
  int sqrtDimension = (int) Math.sqrt(sudoku.getDimension());
  for (int i = 0; i < sqrtDimension; i++) {
    for (int j = 0; j < sqrtDimension; j++) {
      sudoku.getCell(irows[i],
          jcolumns[j]).removeCandidates(finalNumber);
    }
  }

}

public ArrayList<Cell> getCellsBox(Sudoku sudoku, int[] irows,
    int[] jcolumns) {
  int sqrtDimension = (int) Math.sqrt(sudoku.getDimension());
```

```
    ArrayList<Cell> cellsBox = new ArrayList<>();
    for (int i = 0; i < sqrtDimension; i++) {
      for (int j = 0; j < sqrtDimension; j++) {
        cellsBox.add(sudoku.getCell(irows[i], jcolumns[j]));
      }
    }
    return cellsBox;
  }

}
package sudoku_nxn;

import java.util.ArrayList;

public class Cell implements Cloneable {

  private ArrayList<Integer> candidates;
  private int r;
  private int c;
  private int finalNumber;
  private int n;

  public Cell(int n) {
    candidates = new ArrayList<>();
    for (int i = 0; i < n; i++) {
      candidates.add(i + 1);
    }
    finalNumber = 0;
    this.n = n;
  }

  public Cell(int finalNumber, int n) {
    candidates = new ArrayList<>(n);
    this.finalNumber = finalNumber;
    this.n = n;
  }

  private Cell(ArrayList<Integer> candidates, int r, int c, int
      finalNumber, int n) {
    this.candidates = candidates;
    this.r = r;
    this.c = c;
```

```
    this.finalNumber = finalNumber;
    this.n = n;
  }

  public ArrayList<Integer> getCandidates() {
    return candidates;
  }

  public void removeCandidates(Integer... numbers) {
    for (int numb : numbers) {
      if (candidates.contains(numb)) {
        candidates.remove(Integer.valueOf(numb));
      }
    }

  }

  public void addCandidates(Integer... numbers) {
    for (int numb : numbers) {
      candidates.add(numb);
    }

  }

  public int getFinalNumber() {
    return finalNumber;
  }

  public void setFinalNumber(int finalNumber) {
    this.finalNumber = finalNumber;
    this.candidates.removeAll(SudokuUtilities.getAllCandidates(n));
  }

  public void setFinalNumberWithoutRemovingCandidates(int
      finalNumber) {
    this.finalNumber = finalNumber;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
```

```
    result = prime * result + ((candidates == null) ? 0 :
        candidates.hashCode());
    result = prime * result + finalNumber;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Cell other = (Cell) obj;
    if (candidates == null) {
      if (other.candidates != null)
        return false;
    } else if (!candidates.equals(other.candidates))
      return false;
    if (finalNumber != other.finalNumber)
      return false;
    return true;
}

@Override
public String toString() {
    return "Cell [candidates=" + candidates + ", r=" + r + ", c=" + c +
        ", finalNumber=" + finalNumber + "]";
}

public void removeCandidatesIf(ArrayList<Integer> cand2) {
    ArrayList<Integer> arrApp = new ArrayList<>(candidates);
    for (int numb : cand2) {
      if (candidates.contains(numb)) {
        arrApp.removeAll(cand2);
        if (!arrApp.isEmpty()) {
          candidates.remove(Integer.valueOf(numb));
        } else {
          arrApp.addAll(cand2);
        }
```

```
        }
      }
    }

    public int getC() {
      return c;
    }

    public void setC(int c) {
      this.c = c;
    }

    public int getR() {
      return r;
    }

    public void setR(int r) {
      this.r = r;
    }

    @Override
    public Cell clone() throws CloneNotSupportedException {
      ArrayList<Integer> cands = new ArrayList<>();
      for (Integer integ : candidates) {
        cands.add(Integer.valueOf(integ.intValue()));
      }
      return new Cell(cands, r, c, finalNumber, n);
    }

    public boolean onlyOneCandidateIsPresent() {
      return getCandidates().size() == 1;
    }

    public boolean isNotAlreadyOccupied() {
      return getFinalNumber() == 0;
    }

    public int getN() {
      return n;
    }
  }
  package sudoku_nxn;
```

```
import java.util.Comparator;

public class CellComparator implements Comparator<Cell> {

  @Override
  public int compare(Cell o1, Cell o2) {
    return o1.getCandidates().size() - o2.getCandidates().size();
  }

}
package sudoku_nxn;

import java.util.Stack;

public class Configuration {
  Stack<Sudoku> stackPreviousSudoku;
  Stack<Cell> stackCellPreviousSetting;

  public Configuration() {
    stackPreviousSudoku = new Stack<>();
    stackCellPreviousSetting = new Stack<>();
  }

  public void pushSudokuConfig(Sudoku sudoku) {
    stackPreviousSudoku.push(sudoku);
  }

  public void pushCellPreviousSetting(Cell cell) {
    stackCellPreviousSetting.push(cell);
  }

  public Sudoku popPreviousSudoku() {
    return stackPreviousSudoku.pop();
  }

  public Cell popPreviousCell() {
    return stackCellPreviousSetting.pop();
  }

  public boolean isPreviousSudokuEmpty() {
    return stackPreviousSudoku.isEmpty();
```

```
  }

}
package sudoku_nxn;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.stream.Collectors;

public class EasyCellSudokuTecnique extends SudokuTechnique {

  private List<Integer> findUniqueNumbers(ArrayList<Cell> array) {
    return array.stream().filter((c) ->
        c.isNotAlreadyOccupied()).flatMap((c) ->
        c.getCandidates().stream())
         .collect(Collectors.groupingBy(Function.identity(),
            Collectors.counting())).entrySet().stream()
         .filter(c -> c.getValue() == 1).map(c ->
            c.getKey()).collect(Collectors.toList());
  }

  private void findAndInsertAllBoxUniqueNumbers(int box, Sudoku
      sudoku) {
    BoxManagement boxManager = sudoku.getBoxManagement();
    int[][] boxes = boxManager.getBoxes();
    int[][] boxindexes =
        boxManager.getBoxIndexes(boxes[box][0],boxes[box][1]);
    int[] irows = boxindexes[0];
    int[] jcolumns = boxindexes[1];
    ArrayList<Cell> cellsBox = boxManager.getCellsBox(sudoku,
        irows, jcolumns);
    List<Integer> numbersUnique = findUniqueNumbers(cellsBox);
    insertUniqueNumbersInBox(sudoku, irows, jcolumns,
        numbersUnique);
  }

  private void insertUniqueNumbersInBox(Sudoku sudoku, int[] irows,
      int[] jcolumns, List<Integer> numbersUnique) {
    int sqrtDimension = (int) Math.sqrt(sudoku.getDimension());
    for (int z = 0; z < numbersUnique.size(); z++) {
      int finalNumber = numbersUnique.get(z);
```

```
    for (int i = 0; i < sqrtDimension; i++) {
      for (int j = 0; j < sqrtDimension; j++) {
        Cell cell = sudoku.getCell(irows[i],jcolumns[j]);
        if (cell.getCandidates().contains(finalNumber)) {
          sudoku.addNumberToSudokuSolution(finalNumber, cell);
          removeAllUselessCandidates(sudoku);
          return;
        }
      }
    }
  }
}

private void putAllUniqueNumbersInRow(List<Integer>
    numbersUnique, int j, Sudoku sudoku) {
  int dimension = sudoku.getDimension();
  for (int z = 0; z < numbersUnique.size(); z++) {
    int finalNumber = numbersUnique.get(z);
    for (int i = 0; i < dimension; i++) {
      Cell cell = sudoku.getCell(j, i);
      if (cell.getCandidates().contains(finalNumber)) {
        sudoku.addNumberToSudokuSolution(finalNumber, cell);
        removeAllUselessCandidates(sudoku);
        return;
      }

    }
  }
}

private void putAllUniqueNumbersInColumn(List<Integer>
    numbersUnique, int j, Sudoku sudoku) {
  int dimension = sudoku.getDimension();
  for (int z = 0; z < numbersUnique.size(); z++) {
    int finalNumber = numbersUnique.get(z);
    for (int i = 0; i < dimension; i++) {
      Cell cell = sudoku.getCell(i, j);
      if (cell.getCandidates().contains(finalNumber)) {
        sudoku.addNumberToSudokuSolution(finalNumber, cell);
        removeAllUselessCandidates(sudoku);
        return;
      }
```

```
        }

      }
    }

    @Override
    public void apply(Sudoku sudoku) {
      int dimension = sudoku.getDimension();
      for (int j = 0; j < dimension; j++) {
        ArrayList<Cell> cellsRow = sudoku.getCells().get(j);
        List<Integer> numbersUnique = findUniqueNumbers(cellsRow);
        putAllUniqueNumbersInRow(numbersUnique, j, sudoku);
        ArrayList<Cell> cellsColumn =
            SudokuUtilities.getCellsColumn(sudoku,j);
        numbersUnique = findUniqueNumbers(cellsColumn);
        putAllUniqueNumbersInColumn(numbersUnique, j, sudoku);
        findAndInsertAllBoxUniqueNumbers(j, sudoku);
      }
    }



}
package sudoku_nxn;

public class HiddenSingleSudokuTechnique extends SudokuTechnique {

    @Override
    public void apply(Sudoku sudoku) {
      for (Cell c : sudoku) {
        if (c.onlyOneCandidateIsPresent()) {
          int finalNumber = c.getCandidates().get(0);
          sudoku.addNumberToSudokuSolution(finalNumber, c);
          removeAllUselessCandidates(sudoku);
        }
      }

    }

}
package sudoku_nxn;
```

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.Map.Entry;

public class NTupleSudokuTechnique extends SudokuTechnique {

  @Override
  public void apply(Sudoku sudoku) {
    int dimension = sudoku.getDimension();
    for (int i = 0; i < dimension; i++) {
      ArrayList<Cell> cellsRow = sudoku.getCells().get(i);
      findNTuple(sudoku, i, cellsRow, "row");
      ArrayList<Cell> cellsColumn =
          SudokuUtilities.getCellsColumn(sudoku, i);
      findNTuple(sudoku, i, cellsColumn, "column");
      ArrayList<Cell> cellsBox = getCellsBox(sudoku, i);
      findNTuple(sudoku, i, cellsBox, "box");
    }
  }



  private ArrayList<Cell> getCellsBox(Sudoku sudoku, int box) {
    int sqrtDimension = (int) Math.sqrt(sudoku.getDimension());
    BoxManagement boxManager = sudoku.getBoxManagement();
    int[][] boxes = boxManager.getBoxes();
    int[][] indexes =
        boxManager.getBoxIndexes(boxes[box][0],boxes[box][1]);
    int[] irows = indexes[0];
    int[] jcolumns = indexes[1];
    ArrayList<Cell> cellsBox = new ArrayList<>();
    for (int i = 0; i < sqrtDimension; i++) {
      for (int j = 0; j < sqrtDimension; j++) {
        cellsBox.add(sudoku.getCell(irows[i], jcolumns[j]));
      }
    }
    return cellsBox;
  }
```

```
private void findNTuple(Sudoku sudoku, int i, ArrayList<Cell>
    cellsApp, String removeWhere) {
  Map<Cell, Integer> cellAndCount = generateCellMap(cellsApp,
      sudoku.getDimension());
  Set<Entry<Cell, Integer>> entrySet = cellAndCount.entrySet();
  for (Entry<Cell, Integer> entry : entrySet) {
    if (entry.getValue() > 1) {
      if (entry.getKey().getCandidates().size() ==
          entry.getValue()) {
        removeAllCandidatesFrom(sudoku, i, removeWhere,
            entry.getKey().getCandidates());
      }
    }
  }
}

private Map<Cell, Integer> generateCellMap(ArrayList<Cell>
    cellsApp, int dimension) {
  Map<Cell, Integer> cellAndCount = new HashMap<>();
  for (int j = 0; j < dimension; j++) {
    if (cellsApp.get(j).getFinalNumber() == 0) {
      Integer count = cellAndCount.get(cellsApp.get(j));
      if (count == null) {
        cellAndCount.put(cellsApp.get(j), 1);
      } else {
        cellAndCount.put(cellsApp.get(j), ++count);
      }
    }
  }
  return cellAndCount;
}

private void removeAllCandidatesFrom(Sudoku sudoku, int position,
    String removeWhere,
    ArrayList<Integer> candidates) {
  BoxManagement boxManagement = sudoku.getBoxManagement();
  int dimension = sudoku.getDimension();
  switch (removeWhere) {
  case "row":
    sudoku.getCells().get(position).stream().forEach(c ->
        c.removeCandidatesIf(candidates));
    break;
```

```
      case "column":
        ArrayList<Cell> cellsColumn = new ArrayList<>();
        for (int j = 0; j < dimension; j++) {
          cellsColumn.add(sudoku.getCell(j,position));
        }
        cellsColumn.stream().forEach(c ->
            c.removeCandidatesIf(candidates));
        break;

      case "box":
        int sqrtDimension = (int) Math.sqrt(dimension);
        int[][] boxes = boxManagement.getBoxes();
        int[][] boxIndexes =
            boxManagement.getBoxIndexes(boxes[position][0],boxes[position][1]);
        int[] irows = boxIndexes[0];
        int[] jcolumns = boxIndexes[1];
        for (int i = 0; i < sqrtDimension; i++) {
          for (int j = 0; j < sqrtDimension; j++) {
            sudoku.getCell(irows[i],
                jcolumns[j]).removeCandidatesIf(candidates);
          }
        }
        break;
      }

    }
}
package sudoku_nxn;

import java.util.ArrayList;
import java.util.HashSet;

public class PointingSudokuTechnique extends SudokuTechnique {

  @Override
  public void apply(Sudoku sudoku) {
    int dimension = sudoku.getDimension();
    for (int i = 0; i < dimension; i++) {
      for (int number = 1; number <= dimension; number++) {
        ArrayList<String> pos =
            positionsOfCandidateWhereInBox(sudoku,i, number);
```

```
        if (!pos.isEmpty() &&
            everyElementStartEndWithSameDigit(pos, true)) {
          removePairsRow(sudoku,i, number,
              getRowColumnCandidate(sudoku,i, number)[0]);
        } else if (!pos.isEmpty() &&
            everyElementStartEndWithSameDigit(pos, false)) {
          removePairsColumn(sudoku,i, number,
              getRowColumnCandidate(sudoku,i, number)[1]);
        }
      }
    }

  }

  private void removePairsColumn(Sudoku sudoku,int box, int number,
      int column) {
    int dimension = sudoku.getDimension();
    BoxManagement boxManagement = sudoku.getBoxManagement();
    int[][] boxes = boxManagement.getBoxes();
    int[][] boxIndexes = boxManagement.getBoxIndexes(boxes[box][0],
        boxes[box][1]);
    int[] irows = boxIndexes[0];
    HashSet<Integer> hashSet = createSkippingBoxSet(irows);
    for (int i = 0; i < dimension; i++) {
      if (!hashSet.contains(i)) {
        Cell cell = sudoku.getCell(i, column);
        if (cell.getCandidates().contains(number)) {
          cell.removeCandidates(number);
        }
      }
    }

  }

  private HashSet<Integer> createSkippingBoxSet(int[] array) {
    HashSet<Integer> hashSet = new HashSet<>();
    for(int i=0; i < array.length; i++) {
      hashSet.add(array[i]);
    }
    return hashSet;
  }
```

```
private void removePairsRow(Sudoku sudoku,int box, int number,
    int row) {
  int dimension = sudoku.getDimension();
  BoxManagement boxManagement = sudoku.getBoxManagement();
  int[][] boxes = boxManagement.getBoxes();
  int[][] boxIndexes =
      boxManagement.getBoxIndexes(boxes[box][0],boxes[box][1]);
  int[] jcolumns = boxIndexes[1];
  HashSet<Integer> hashSet = createSkippingBoxSet(jcolumns);
  for (int i = 0; i < dimension; i++) {
    if (!hashSet.contains(i)) {
      Cell cell = sudoku.getCell(row, i);
      if (cell.getCandidates().contains(number)) {
        cell.removeCandidates(number);
      }
    }
  }
}

private boolean
    everyElementStartEndWithSameDigit(ArrayList<String> pos,
    boolean columnTrue_rowFalse) {
  char first = pos.get(0).toCharArray()[0];
  char end = pos.get(0).toCharArray()[pos.get(0).length() - 1];
  return (columnTrue_rowFalse) ? pos.stream().allMatch((string)
      -> string.startsWith("" + first))
       : pos.stream().allMatch((string) -> string.endsWith("" +
          end));

}

private ArrayList<String> positionsOfCandidateWhereInBox(Sudoku
    sudoku,int box, int candidate) {
  BoxManagement boxManagement = sudoku.getBoxManagement();
  int sqrtDimension = (int) Math.sqrt(sudoku.getDimension());
  int[][] boxes = boxManagement.getBoxes();
  int[][] boxIndexes =
      boxManagement.getBoxIndexes(boxes[box][0],boxes[box][1]);
  int[] irows = boxIndexes[0];
  int[] jcolumns = boxIndexes[1];
  ArrayList<String> positions = new ArrayList<>();
  for (int i = 0; i < sqrtDimension; i++) {
```

```
      for (int j = 0; j < sqrtDimension; j++) {
        Cell cell = sudoku.getCell(irows[i], jcolumns[j]);
        if (cell.getCandidates().contains(candidate)) {
          positions.add("" + irows[i] + jcolumns[j]);
        }
      }
    }
    return positions;
  }

  private int[] getRowColumnCandidate(Sudoku sudoku,int box, int
      number) {
    BoxManagement boxManagement = sudoku.getBoxManagement();
    int sqrtDimension = (int) Math.sqrt(sudoku.getDimension());
    int[][] boxes = boxManagement.getBoxes();
    int[][] boxIndexes =
        boxManagement.getBoxIndexes(boxes[box][0],boxes[box][1]);
    int[] irows = boxIndexes[0];
    int[] jcolumns = boxIndexes[1];
    for (int i = 0; i < sqrtDimension; i++) {
      for (int j = 0; j < sqrtDimension; j++) {
        if (sudoku.getCell(irows[i],
            jcolumns[j]).getCandidates().contains(number)) {
          int[] rowCols = { irows[i], jcolumns[j] };
          return rowCols;
        }
      }
    }
    return null;
  }
}
package sudoku_nxn;

import java.util.ArrayList;
import java.util.Iterator;

import sudoku.exception.SudokuException;

public class Sudoku implements Iterable<Cell>, Cloneable {

  private ArrayList<ArrayList<Cell>> cells;
  private int numberFilled;
```

```
private BoxManagement boxManagement;

public ArrayList<ArrayList<Cell>> getCells() {
  return cells;
}

public Cell getCell(int i, int j) {
  return cells.get(i).get(j);
}

public Sudoku(int[][] numbers) {
  if (numbers[0].length != numbers.length) {
    throw new SudokuException("Not valid sudoku. It must be nxn
        sudoku.");
  }
  int n = numbers.length;
  boxManagement = new BoxManagement(n);
  cells = new ArrayList<ArrayList<Cell>>();
  for (int i = 0; i < n; i++) {
    cells.add(new ArrayList<Cell>());
  }
  numberFilled = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      Cell cell;
      if (numbers[i][j] != 0) {
        cell = new Cell(numbers[i][j], n);
        cells.get(i).add(cell);
        numberFilled = getNumberFilled() + 1;
      } else {
        cell = new Cell(n);
        cells.get(i).add(cell);
      }
      cells.get(i).get(j).setR(i);
      cells.get(i).get(j).setC(j);
    }
  }
}

private Sudoku(ArrayList<ArrayList<Cell>> cells, int
    numberFilled) {
  this.cells = cells;
```

```
    this.numberFilled = numberFilled;
    boxManagement = new BoxManagement(this.getDimension());
  }

  public int getNumberFilled() {
    return numberFilled;
  }

  @Override
  public Iterator<Cell> iterator() {
    return new Iterator<Cell>() {

      Cell current = null;
      int numberOfCell = 0;
      int row = 0;
      int column = 0;

      @Override
      public boolean hasNext() {
        return (current == null) ? false : (numberOfCell <
            current.getN() * current.getN()) ? true : false;
      }

      @Override
      public Cell next() {
        current = cells.get(row).get(column);
        if (column == 8) {
          row++;
          column = 0;
        } else {
          column++;
        }
        numberOfCell++;
        return current;
      }

    };
  }

  public String toString() {
    StringBuilder string = new StringBuilder();
    int dimension = cells.size();
```

```
  int sqrtDimension = (int) Math.sqrt(dimension);
  for (int r = 0; r < dimension; r++) {
    if(r%sqrtDimension==0) {
      createHorizontalLine(string, dimension, sqrtDimension);
    }
    for (int c = 0; c < dimension; c++) {
      if (c % sqrtDimension == 0) {
        string.append("|");
      }
      if (this.getCell(r, c).getFinalNumber() != 0) {
        string.append(this.getCells().get(r).get(c).getFinalNumber()
            + "");
      } else {
        string.append("o");
      }
      if (c == dimension - 1) {
        string.append("|" + "\n");
      }
    }
  }
  createHorizontalLine(string, dimension, sqrtDimension);
  return string.toString();
}

private void createHorizontalLine(StringBuilder string, int
    dimension, int sqrtDimension) {
  for(int i=0; i<dimension+sqrtDimension+1;i++) {
    if(i==0 || i==dimension+sqrtDimension) {
      string.append("#");
    }else {
      string.append("-");
    }
  }
  string.append("\n");
}



@Override
public Sudoku clone() throws CloneNotSupportedException {
  ArrayList<ArrayList<Cell>> sud = new ArrayList<>();
  for (ArrayList<Cell> arrCell : cells) {
```

```
        ArrayList<Cell> arrAppo = new ArrayList<>();
        for (Cell cell : arrCell) {
          arrAppo.add(cell.clone());
        }
        sud.add(arrAppo);
      }
      return new Sudoku(sud, numberFilled);
    }

    public void addNumberToSudokuSolution(int finalNumber, Cell cell)
        {
      cell.setFinalNumber(finalNumber);
      this.numberFilled++;
    }

    public int getDimension() {
      return cells.size();
    }

    public BoxManagement getBoxManagement() {
      return boxManagement;
    }

}
package sudoku_nxn;

import java.util.Arrays;
import java.util.Optional;

import sudoku.exception.SudokuException;

public class SudokuSolver {

    private int numberIteration;
    private SudokuTechniques sudokuTechniques;

    public SudokuSolver(int numberIteration, SudokuTechniques
        sudokuTechniques) {
      this.numberIteration = numberIteration;
      this.sudokuTechniques = sudokuTechniques;
    }
```

```
public Sudoku sudokuSolution(Sudoku sudoku) throws
    CloneNotSupportedException {
  removeAllUselessCandidates(sudoku);
  if (!solvable(sudoku))
    throw new SudokuException("Sudoku not solvable");
  int count = 0;
  while (count <= numberIteration && isNotSolved(sudoku)) {
    simplify(sudoku);
    count++;
    if (!solvable(sudoku)) {
      throw new SudokuException("Sudoku not solvable");
    }
  }
  return sudoku = bruteForceSolving(sudoku);

}

private void removeAllUselessCandidates(Sudoku sudoku) {
  sudokuTechniques.getTechniques().get(0).removeAllUselessCandidates(sudoku);
}

private void simplify(Sudoku sudoku) {
  for (SudokuTechnique sudokuTechnique :
      sudokuTechniques.getTechniques()) {
    sudokuTechnique.apply(sudoku);
  }
}

private Sudoku bruteForceSolving(Sudoku sudoku) throws
    CloneNotSupportedException {
  Optional<Cell> cellSetting = Optional.empty();
  Cell[] cells = null;
  Configuration config = new Configuration();
  CellComparator compare = new CellComparator();
  boolean exit = false;
  while (!exit) {
    config.pushSudokuConfig(sudoku);
    Sudoku sudoku2 = config.popPreviousSudoku();
    sudoku = sudoku.clone();
    cells = findCellsWithNoFinalNumber(sudoku);
    Arrays.sort(cells, compare);
    if (isNotSolved(sudoku)) {
```

```
        int number = getLowerDigitToInsert(cells, cellSetting);
        if (number == -1) {
          if (config.isPreviousSudokuEmpty()) {
            throw new SudokuException("Sudoku not solvable");
          }
          sudoku = config.popPreviousSudoku();
          cellSetting = Optional.of(config.popPreviousCell());
          // re-upload
        } else {
          cellSetting = getCellSetting(sudoku, cellSetting, cells,
              number);
          config.pushCellPreviousSetting(cellSetting.get().clone());
          int row = cellSetting.get().getR();
          int column = cellSetting.get().getC();
          sudoku.addNumberToSudokuSolution(number,
              sudoku.getCell(row, column));
          config.pushSudokuConfig(sudoku2);
          int count = 0;
          removeAllUselessCandidates(sudoku);
          if (!solvable(sudoku)) {
            sudoku = config.popPreviousSudoku();
            cellSetting = Optional.of(config.popPreviousCell());
          } else {
            boolean isSolvable = true;
            while (count < numberIteration && isNotSolved(sudoku))
                {
              simplify(sudoku);
              isSolvable = solvable(sudoku);
              if (!isSolvable) {
                sudoku = config.popPreviousSudoku();
                cellSetting =
                    Optional.of(config.popPreviousCell());
                break;
              }
              count++;
            }
            if (isSolvable) {
              cellSetting = Optional.empty();
            }
          }
        }
      } else {
```

```
        exit = true;
      }
    }
    return sudoku;
}

private Optional<Cell> getCellSetting(Sudoku sudoku,
    Optional<Cell> cellSetting, Cell[] cells, int number)
     throws CloneNotSupportedException {
  Cell cell;
  if (cellSetting.isEmpty()) {
    cell = cells[0];
    cellSetting = Optional.of(cell.clone());
    cellSetting.get().setFinalNumberWithoutRemovingCandidates(number);
  } else {
    cellSetting = getPreviousCellSetting(sudoku, cellSetting);
    cellSetting.get().setFinalNumberWithoutRemovingCandidates(number);
  }
  return cellSetting;
}

private Cell[] findCellsWithNoFinalNumber(Sudoku sudoku) {
  return sudoku.getCells().stream().flatMap(el ->
      el.stream()).filter(c -> c.getFinalNumber() == 0)
       .toArray(Cell[]::new);
}

private Optional<Cell> getPreviousCellSetting(Sudoku sudoku,
    Optional<Cell> cellSetting)
     throws CloneNotSupportedException {
  Cell cell;
  int r = cellSetting.get().getR();
  int c = cellSetting.get().getC();
  cell = sudoku.getCell(r, c);
  Cell cellAppo = cell.clone();
  Integer[] candidates =
      SudokuUtilities.getAllCandidatesArray(sudoku.getDimension());
  cellAppo.removeCandidates(candidates);
  cellAppo.addCandidates(cellSetting.get()
  .getCandidates().toArray(Integer[]::new));
  cellSetting = Optional.of(cellAppo);
  return cellSetting;
```

```
    }

    private boolean isNotSolved(Sudoku sudoku) {
      return sudoku.getNumberFilled() != sudoku.getDimension()*
          sudoku.getDimension();
    }

    private int getLowerDigitToInsert(Cell[] cells, Optional<Cell>
        cellSetting) {
      if (cellSetting.isEmpty()) {
        return cells[0].getCandidates().get(0);
      } else {
        Optional<Integer> number =
            cellSetting.get().getCandidates().stream()
             .filter(c -> c >
                 cellSetting.get().getFinalNumber()).findFirst();
        return (number.isPresent()) ? number.get() : -1;
      }
    }

    private boolean solvable(Sudoku sudoku) {
      boolean somethingToInsert =
          !isAnyEmptyCellWithNoCandidates(sudoku);
      BoxManagement boxManagement = sudoku.getBoxManagement();
      for (int i = 0; i < sudoku.getDimension(); i++) {
        if (!checkRowColumnConstraint(sudoku, i) ||
            !checkBoxConstraint(sudoku, boxManagement, i)) {
          return false;
        }
      }
      return somethingToInsert;

    }

    private boolean checkRowColumnConstraint(Sudoku sudoku, int
        rowColumn) {
      int dimension = sudoku.getDimension();
      for (int numb = 1; numb <= dimension; numb++) {
        int countRow = 0;
        int countCol = 0;
        for (int j = 0; j < dimension; j++) {
```

```
        if (sudoku.getCell(rowColumn, j).getFinalNumber() == numb)
            {
          countRow++;
        }
        if (sudoku.getCell(j, rowColumn).getFinalNumber() == numb)
            {
          countCol++;
        }
        if (countRow > 1 || countCol > 1) {
          return false;
        }
      }
    }
    return true;
}

private boolean checkBoxConstraint(Sudoku sudoku, BoxManagement
    boxManagement, int box) {
    int[][] boxes = boxManagement.getBoxes();
    int[][] boxIndexes = boxManagement.getBoxIndexes(boxes[box][0],
        boxes[box][1]);
    int[] jrows = boxIndexes[0];
    int[] zcolumns = boxIndexes[1];
    int dimension = sudoku.getDimension();
    int sqrtDimension = (int) Math.sqrt(dimension);
    for (int number = 1; number <= dimension; number++) {
      int countBox = 0;
      for (int j = 0; j < sqrtDimension; j++) {
        for (int z = 0; z < sqrtDimension; z++) {
          if (sudoku.getCell(jrows[j],
              zcolumns[z]).getFinalNumber() == number) {
            countBox++;
          }
          if (countBox > 1) {
            return false;
          }
        }
      }
    }
    return true;
}
```

```
  private boolean isAnyEmptyCellWithNoCandidates(Sudoku sudoku) {
    return sudoku.getCells().stream().flatMap(c -> c.stream())
        .anyMatch(c -> c.getCandidates().isEmpty() &&
            c.isNotAlreadyOccupied());
  }

}
package sudoku_nxn;

import java.util.ArrayList;

public abstract class SudokuTechnique {

  public abstract void apply(Sudoku sudoku);

  public void removeAllUselessCandidates(Sudoku sudoku) {
    int dimension = sudoku.getDimension();
    BoxManagement boxManagement = sudoku.getBoxManagement();
    for (int i = 0; i < dimension; i++) {
      ArrayList<Cell> cellsColumn = new ArrayList<Cell>();
      for (int j = 0; j < dimension; j++) {
        int finalNumber = sudoku.getCell(i, j).getFinalNumber();
        if (finalNumber != 0) {
          int[][] boxIndexes = boxManagement.getBoxIndexes(i, j);
          boxManagement.removeBox(boxIndexes[0], boxIndexes[1],
              finalNumber, sudoku);
          sudoku.getCells().get(i).forEach(c ->
              c.removeCandidates(finalNumber)); // remove from row
        }
        cellsColumn.add(sudoku.getCell(j, i));
      }
      for (int z = 0; z < dimension; z++) {
        int finalNumber = cellsColumn.get(z).getFinalNumber();
        if (finalNumber != 0) {
          cellsColumn.forEach((c) ->
              c.removeCandidates(finalNumber));
        }
      }
    }

  }
```

```
}
package sudoku_nxn;

import java.util.ArrayList;

import sudoku.exception.NullTechniqueException;

public class SudokuTechniques {

  public final static class Builder {

    private ArrayList<SudokuTechnique> techniques;

    private Builder(SudokuTechnique tech) {
      techniques = new ArrayList<>();
      addTechnique(tech);
    }

    public static Builder newInstance(SudokuTechnique tech) {
      return new Builder(tech);
    }

    public Builder addTechnique(SudokuTechnique tech) {
      if (tech == null) {
        throw new NullTechniqueException("Technique must not be
            null");
      }
      techniques.add(tech);
      return this;
    }

    public SudokuTechniques build() {
      return new SudokuTechniques(techniques);
    }
  }

  private Sudoku sudoku;
  private ArrayList<SudokuTechnique> techniques;

  private SudokuTechniques(ArrayList<SudokuTechnique>
      sudokuTechniques) {
    this.techniques = sudokuTechniques;
```

```
  }

  public ArrayList<SudokuTechnique> getTechniques() {
    return techniques;
  }

  public boolean isTechniquesEmpty() {
    return techniques.isEmpty();
  }

  public Sudoku getSudoku() {
    return sudoku;
  }

}
package sudoku_nxn;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class SudokuUtilities {

  public static ArrayList<Cell> getCellsColumn(Sudoku sudoku, int
      column) {
    int dimension = sudoku.getDimension();
    ArrayList<Cell> cellsColumn = new ArrayList<>(dimension);
    for (int j = 0; j < dimension; j++) {
      cellsColumn.add(sudoku.getCell(j, column));
    }
    return cellsColumn;
  }

  public static List<Integer> getAllCandidates(int n) {
    return Stream.iterate(1, i -> i + 1).limit(n +
        1).collect(Collectors.toList());
  }

  public static Integer[] getAllCandidatesArray(int n) {
    return getAllCandidates(n).toArray(Integer[]::new);
  }
```

```
}
package sudoku9x9.exception;

public class NullTechniqueException extends RuntimeException {

  private static final long serialVersionUID =
      -6404649193882340937L;

  public NullTechniqueException(String message) {
    super(message);
  }

}
package sudoku.exception;

public class SudokuException extends RuntimeException {

  private static final long serialVersionUID =
      -5524999724270067642L;

  public SudokuException(String message) {
    super(message);
  }

}
```

A.2 ALGORITMO SAT

Abbiamo implementato sia uno strumento che ci consentisse di creare la formula booleana che ci descrive il sudoku, sia un algoritmo che trova un assegnamento ad una formula espressa in CNF: il meccanismo che è stato utilizzato è il DPLL[9].

A.2.1 *DPLL e suo funzionamento*

Il DPLL è un algoritmo che sfrutta il backtracking, al quale però aggiunge una notevole miglioria: infatti, quando si assegna un valore ad una variabile, si rimuove tale variabile da ogni clausola della formula e si rimuovono le clausole già soddisfatte da questo assegnamento. Otterremo

così un assegnamento parziale e, qualora una clausola diventasse vuota, allora significherebbe che sicuramente abbiamo effettuato un errore e dovremo ripristinare la configurazione precedente, provando un assegnamento alternativo. Ad ogni assegnamento si controlla se sono presenti clausole con un solo letterale: l'assegnamento che riguarda tale variabili è obbligato (a causa del fatto che la formula è espressa in forma CNF e tutte le clausole devono essere soddisfatte). Se presenti, si assegna il valore della variabile in modo che soddisfi la clausola e si rimuove tale informazione dalla formula CNF, oltre a rimuovere la clausola contenente solo quell'elemento.
E' un algoritmo di stampo puramente ricorsivo e, data la sua natura ricorsiva, è necessario ogni volta che riduciamo la formula, creare una copia di quest'ultima, affinché si possa ripristinare una configurazione precedente. Per rappresentare l'assegnamento, viene utilizzata una stringa lunga quante sono le variabili presenti nella formula: se il carattere in posizione i-esima è un 1 oppure uno 0, allora si assegna a tale variabile il valore corrispondente, se invece il carattere in posizione i-esima è ".", allora vuol dire che non c'è stato ancora un assegnamento riguardante l'elemento in tale posizione. Esistono delle ottimizzazioni di questo algoritmo che dipendono fortemente dalla variabile scelta per l'assegnamento. Noi adotteremo l'approccio più semplice, che consiste nello scegliere la prima variabile a cui non è ancora stato assegnato un valore. Il metodo che si occupa di questo è chooseCNFVariable.

```
package satsudoku;

import java.util.Map.Entry;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import satsudoku.exception.NotClonableExpressionException;

public class SatSolver {

  private String result;
  private Map<Integer, String> variablesMap;
  private CNFExpression cnfAppo;
```

```
public SatSolver(ArrayList<? extends Variable> vars) {
  variablesMap = IntStream.range(0, vars.size()).boxed()
       .collect(Collectors.toMap(i -> i, i ->
           vars.get(i).getVarRapresentation()));
}

public boolean findAssignment(CNFExpression cnf) {
  StringBuffer str = new StringBuffer();
  for (int i = 0; i < variablesMap.size(); i++) {
    str.append('.');
  }
  return DPLL(cnf, str.toString());
}

private boolean DPLL(CNFExpression cnf2, String assignment) {
  if (cnf2.getClauses().size() == 0) {
    result = new String(assignment);
    return true;
  }
  if (isCNFContainingEmptyClause(cnf2)) {
    return false;
  }
  ArrayList<Litteral> litterals =
      getLitteralsFromUnitClause(cnf2);
  ArrayList<Character> elementsToBeAssigned = new ArrayList<>();
  ArrayList<Variable> variabs = new ArrayList<>();
  StringBuilder stringAssign = new StringBuilder(assignment);
  if (!litterals.isEmpty()) {
    for (Litteral litt : litterals) {
      char elementToBeAssigned = '1';
      if (litt.isNegated()) {
        elementToBeAssigned = '0';
      }
      int pos = getPos(litt);
      if (stringAssign.toString().charAt(pos) != '.') {
        if (stringAssign.toString().charAt(pos) !=
            elementToBeAssigned) {
          return false;
        }
      }
      stringAssign.setCharAt(pos, elementToBeAssigned);
```

```
        variabs.add(litt.getVar());
        elementsToBeAssigned.add(elementToBeAssigned);
      }
      return DPLL(reduction(cnf2, variabs, elementsToBeAssigned),
          stringAssign.toString());
    }
    stringAssign = new StringBuilder(assignment);
    int pos = 0;
    Variable var = chooseCNFVariable(assignment);
    pos = getPos(var);
    stringAssign.setCharAt(pos, '1');
    ArrayList<Variable> wrapVar = wrap(var);
    ArrayList<Character> wrapChar1 = wrap('1');
    ArrayList<Character> wrapChar0 = wrap('0');

    StringBuilder stringAssign2 = new StringBuilder(assignment);
    stringAssign2.setCharAt(pos, '0');
    return DPLL(reduction(cnf2, wrapVar, wrapChar1),
        stringAssign.toString())
        || DPLL(reduction(cnf2, wrapVar, wrapChar0),
            stringAssign2.toString());
  }

  private <T extends Object> ArrayList<T> wrap(T element) {
    ArrayList<T> arr = new ArrayList<>();
    arr.add(element);
    return arr;
  }

  private int getPos(Variable var) {
    int pos = -1;
    Set<Entry<Integer, String>> entrySet = variablesMap.entrySet();
    for (Entry<Integer, String> entry : entrySet) {
      if (entry.getValue().equals(var.getVarRapresentation())) {
        pos = entry.getKey();
        break;
      }
    }
    return pos;
  }

  private int getPos(Litteral litt) {
```

```
    return getPos(litt.getVar());
  }

  protected CNFExpression reduction(CNFExpression cnf2,
      ArrayList<Variable> vars, ArrayList<Character> values) {
    try {
      cnfAppo = cnf2.clone();
    } catch (CloneNotSupportedException e) {
      throw new
          NotClonableExpressionException(cnfAppo.getClass().getSimpleName()
          + " must implement Cloneable");
    }
    for (int i = 0; i < vars.size(); i++) {
      Variable var = vars.get(i);
      Character value = values.get(i);
      Iterator<Clause> iter = cnfAppo.getClauses().iterator();
      while (iter.hasNext()) {
        Clause clause = iter.next();
        Iterator<Litteral> iterLitterals =
            clause.getLitterals().iterator();
        while (iterLitterals.hasNext()) {
          Litteral litt = iterLitterals.next();
          if (variableEqualsRapresentation(var, litt)) {
            iterLitterals.remove();
            if (litt.isNegated() && value == '0') {
              iter.remove();
              break;
            }
            if (!litt.isNegated() && value == '1') {
              iter.remove();
              break;
            }
          }
        }
      }
    }
    return cnfAppo;
  }

  private boolean variableEqualsRapresentation(Variable var,
      Litteral litt) {
    return
```

```
        litt.getVar().getVarRapresentation().equals(var.getVarRapresentation());
  }

  protected Variable chooseCNFVariable(String assignment) {
    for (int i = 0; i < assignment.length(); i++) {
      if (assignment.charAt(i) == '.') {
        String str = variablesMap.get(i);
        return new Variable(str);
      }
    }
    return null;
  }

  private ArrayList<Litteral>
      getLitteralsFromUnitClause(CNFExpression cnf2) {
    ArrayList<Litteral> litterals = new ArrayList<>();
    for (Clause clause : cnf2.getClauses()) {
      if (clause.getLitterals().size() == 1) {
        litterals.add(clause.getLitterals().get(0));
      }
    }
    return litterals;
  }

  private boolean isCNFContainingEmptyClause(CNFExpression cnf2) {
    for (Clause clause : cnf2.getClauses()) {
      if (clause.getLitterals().isEmpty()) {
        return true;
      }
    }
    return false;
  }

  public String getResult() {
    return result;
  }

  public Map<Integer, String> getVariablesMap() {
    return variablesMap;
  }

}
```

A.2.2 *Formula CNF che rappresenta il sudoku*

Sono state creati i seguenti oggetti per creare la formula in CNF:

- Variable, che rappresenta una variabile booleana
- SudokuVariable, variabile del formato (r,c,v), utilizzata per il sudoku
- Litteral, rappresenta una variabile affermata o negata
- Clause: questo oggetto contiene un ArrayList di Litteral
- CNFExpression: l'espressione CNF è caratterizzata da un insieme di oggetti Clause (si utilizza un ArrayList).

L'oggetto Clause e l'oggetto CNFExpression hanno dei metodi che consentono di concatenare alle clausole e alle espressioni altri elementi.

```
package satsudoku;

public class Variable implements Cloneable {

  private String varRapresentation;

  public Variable(String var) {
    this.varRapresentation = var;
  }

  public String getVarRapresentation() {
    return varRapresentation;
  }

  public void setVarRapresentation(String var) {
    this.varRapresentation = var;
  }

  @Override
  public String toString() {
    return "Variable [var=" + varRapresentation + "]";
  }

  @Override
  public Variable clone() throws CloneNotSupportedException {
```

```
    return new Variable(varRapresentation);
  }

}
package satsudoku;

public class SudokuVariable extends Variable{
  private int r;
  private int c;
  private int v;

  public SudokuVariable(String var) {
    super(var);
    getPosition(var);
  }

  private void getPosition(String var) {
    String appo = var.substring(1, var.length() - 1);
    try {
      String[] splitted = appo.split(",");
      r = Integer.parseInt(splitted[0]);
      c = Integer.parseInt(splitted[1]);
      v = Integer.parseInt(splitted[2]);
    }catch(NumberFormatException exp){
      throw new NumberFormatException("Variable of Sudoku is expressed
          by the string(row,column,value)");
    }
  }

  public int getR() {
    return r;
  }

  public int getC() {
    return c;
  }

  public int getV() {
    return v;
  }

  @Override
```

```
  public SudokuVariable clone() throws CloneNotSupportedException {
    return new SudokuVariable(this.getVarRapresentation());
  }

}

package satsudoku;

public class Litteral implements Cloneable{

  private Variable var;
  private boolean isNegated;

  public Litteral(Variable var, boolean isNegated) {
    this.var = var;
    this.isNegated = isNegated;
  }

  public Litteral(Variable var) {
    this.var = var;
    this.isNegated = false;
  }

  public Variable getVar() {
    return var;
  }

  public void setVar(Variable var) {
    this.var = var;
  }

  public boolean isNegated() {
    return isNegated;
  }

  public void setNegated(boolean isNegated) {
    this.isNegated = isNegated;
  }

  @Override
  public Litteral clone() throws CloneNotSupportedException {
    return new Litteral(var.clone(),isNegated);
```

```
  }

  @Override
  public String toString() {
    return "Litteral [var=" + var + ", isNegated=" + isNegated + "]";
  }

}
package satsudoku;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Clause implements Cloneable{

  private ArrayList<Litteral> litterals;

  public Clause() {
    litterals = new ArrayList<>();
  }

  public Clause(ArrayList<Litteral> litterals) {
    this.litterals = litterals;
  }

  public ArrayList<Litteral> getLitterals() {
    return litterals;
  }

  public Clause concatLitterals(Litteral... litterals) {
    this.litterals.addAll(List.of(litterals));
    return this;
  }

  @Override
  public Clause clone() throws CloneNotSupportedException {
    ArrayList<Litteral> newLitterals = new ArrayList<>();
    for(Litteral litt : litterals) {
      newLitterals.add(litt.clone());
    }
    return new Clause(newLitterals);
```

```
  }

  @Override
  public String toString() {
    return "Clause [litterals=" +
        Arrays.toString(litterals.toArray(Litteral[]::new)) + "]";
  }


}
package satsudoku;

import java.util.ArrayList;
import java.util.List;

public class CNFExpression implements Cloneable {

  private ArrayList<Clause> clauses;

  public CNFExpression() {
    clauses = new ArrayList<>();
  }

  public CNFExpression(ArrayList<Clause> clauses) {
    this.clauses = clauses;
  }

  public CNFExpression concatClauses(Clause... clauses) {
    this.clauses.addAll(List.of(clauses));
    return this;
  }

  public CNFExpression concatClauses(ArrayList<Clause> clauses) {
    this.clauses.addAll(clauses);
    return this;
  }

  public ArrayList<Clause> getClauses() {
    return clauses;
  }

  @Override
```

```
  public CNFExpression clone() throws CloneNotSupportedException {
    ArrayList<Clause> newClauses = new ArrayList<>();
    for(Clause clause : clauses) {
      newClauses.add(clause.clone());
    }
    return new CNFExpression(newClauses);
  }

}
package sudoku9x9;

import java.util.ArrayList;
import java.util.Iterator;

public class Sudoku implements Iterable<Cell>, Cloneable {

  private ArrayList<ArrayList<Cell>> cells;
  private int numberFilled;

  public ArrayList<ArrayList<Cell>> getCells() {
    return cells;
  }

  public Cell getCell(int i, int j) {
    return cells.get(i).get(j);
  }

  public Sudoku(int[][] numbers) {
    cells = new ArrayList<ArrayList<Cell>>();
    for (int i = 0; i < 9; i++) {
      cells.add(new ArrayList<Cell>());
    }
    numberFilled = 0;
    for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 9; j++) {
        Cell cell;
        if (numbers[i][j] != 0) {
          cell = new Cell(numbers[i][j]);
          cells.get(i).add(cell);
          numberFilled = getNumberFilled() + 1;
        } else {
          cell = new Cell();
```

```
          cells.get(i).add(cell);
        }
        cells.get(i).get(j).setR(i);
        cells.get(i).get(j).setC(j);
      }
    }
  }

  private Sudoku(ArrayList<ArrayList<Cell>> cells, int
      numberFilled) {
    this.cells = cells;
    this.numberFilled = numberFilled;
  }

  public int raiseNumberFilled() {
    return ++numberFilled;
  }

  public int getNumberFilled() {
    return numberFilled;
  }

  @Override
  public Iterator<Cell> iterator() {
    return new Iterator<Cell>() {

      int numberOfCell = 0;
      int row = 0;
      int column = 0;

      @Override
      public boolean hasNext() {
        return numberOfCell < 81;
      }

      @Override
      public Cell next() {
        Cell current = cells.get(row).get(column);
        if (column == 8) {
          row++;
          column = 0;
        } else {
```

```
            column++;
          }
          numberOfCell++;
          return current;

        }

      };
    }

    public String toString() {
      StringBuilder string = new StringBuilder();
      for (int r = 0; r < 9; r++) {
        if (r % 3 == 0)
          string.append("#-------#-------#-------#" + "\n");
        for (int c = 0; c < 9; c++) {
          if (c % 3 == 0) {
            string.append("| ");
          }
          if (this.getCells().get(r).get(c).getFinalNumber() != 0) {
            string.append(this.getCells()
            .get(r).get(c).getFinalNumber() + " ");
          } else {
            string.append("o ");
          }
          if (c == 8) {
            string.append("|" + "\n");
          }
        }

      }
      string.append("#-------#-------#-------#");
      return string.toString();
    }

    @Override
    public Sudoku clone() throws CloneNotSupportedException {
      ArrayList<ArrayList<Cell>> sud = new ArrayList<>();
      for (ArrayList<Cell> arrCell : cells) {
        ArrayList<Cell> arrAppo = new ArrayList<>();
        for (Cell cell : arrCell) {
          arrAppo.add(cell.clone());
```

```
      }
      sud.add(arrAppo);
    }
    return new Sudoku(sud, numberFilled);
  }

  public void addNumberToSudokuSolution(int finalNumber, Cell cell)
      {
    cell.setFinalNumber(finalNumber);
    this.numberFilled++;
  }

}
```

Esiste un metodo per ogni serie di condizioni logiche che sono state descritte nel capitolo 2 e queste serie di espressioni vengono concatenate fra di loro in una unica grande espressione. E' stato utilizzato nel nostro caso l'Extended Encoding.

```
package satsudoku;

import java.util.ArrayList;

import java.util.List;

public class SudokuExpressionBuilder {

  private static List<SudokuVariable> variables = getVariables(9);

  public static List<SudokuVariable> getVariables() {
    return variables;
  }

  public static CNFExpression getGeneralSudokuCNFExpression() {
    return getSudokuExpression();
  }

  public static SudokuVariable getVariable(String str) {
    for (SudokuVariable var : variables) {
      if (str.equals(var.getVarRapresentation())) {
        return var;
      }
    }
```

```
    return null;
  }

  private static ArrayList<SudokuVariable> getVariables(int n) {
    ArrayList<SudokuVariable> vars = new ArrayList<>();
    for (int i = 0; i <= n - 1; i++) {
      for (int j = 0; j <= n - 1; j++) {
        for (int v = 1; v <= n; v++) {
          SudokuVariable var = new SudokuVariable("(" + i + "," + j
              + "," + v + ")");
          vars.add(var);
        }
      }
    }
    return vars;
  }

  private static CNFExpression getDefinessCellExpression() {
    CNFExpression exp = new CNFExpression();
    for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 9; j++) {
        ArrayList<Litteral> clauseArr = new ArrayList<>();
        for (int n = 1; n <= 9; n++) {
          String var = "(" + i + "," + j + "," + n + ")";
          clauseArr.add(new Litteral(getVariable(var)));
        }
        Clause clause = new Clause(clauseArr);
        exp.concatClauses(clause);
      }

    }
    return exp;
  }

  private static CNFExpression getDefinessRowExpression() {
    CNFExpression exp = new CNFExpression();
    for (int i = 0; i < 9; i++) {
      for (int n = 1; n <= 9; n++) {
        ArrayList<Litteral> clauseArr = new ArrayList<>();
        for (int j = 0; j < 9; j++) {
          String var = "(" + i + "," + j + "," + n + ")";
          clauseArr.add(new Litteral(getVariable(var)));
```

```
        }
        Clause clause = new Clause(clauseArr);
        exp.concatClauses(clause);
      }

    }
    return exp;
  }

  private static CNFExpression getDefinessColumnExpression() {
    CNFExpression exp = new CNFExpression();
    for (int j = 0; j < 9; j++) {
      for (int n = 1; n <= 9; n++) {
        ArrayList<Litteral> clauseArr = new ArrayList<>();
        for (int i = 0; i < 9; i++) {
          String var = "(" + i + "," + j + "," + n + ")";
          clauseArr.add(new Litteral(getVariable(var)));
        }
        Clause clause = new Clause(clauseArr);
        exp.concatClauses(clause);
      }

    }
    return exp;
  }

  private static CNFExpression getDefinessBoxExpression() {
    CNFExpression exp = new CNFExpression();
    for (int z = 1; z <= 9; z++) {
      Clause clause = new Clause();
      for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
          for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
              int r1 = 3*i+r;
              int c1 = 3*j+c;
              String var = "(" + r1 + "," + c1 + "," + z + ")";
              clause.concatLitterals(new
                  Litteral(getVariable(var)));
            }
          }
        }
```

```
      }
      exp.concatClauses(clause);
    }
    return exp;
  }

  private static CNFExpression getCellUniqueness() {
    CNFExpression exp = new CNFExpression();
    for(int r=0; r<9; r++) {
      for(int c=0; c < 9; c++) {
        for(int numb = 1; numb <= 9-1; numb++) {
          for(int numb2 = numb+1; numb2 <= 9; numb2++) {
            Clause clause = new Clause();
            String var1 = "(" + r + "," + c + "," + numb + ")";
            String var2 = "(" + r + "," + c + "," + numb2 + ")";
            Litteral litt1 = new Litteral(getVariable(var1),true);
            Litteral litt2 = new Litteral(getVariable(var2),true);
            clause.concatLitterals(litt1,litt2);
            exp.concatClauses(clause);
          }
        }
      }
    }
    return exp;
  }

  private static CNFExpression getUniquenessBoxExpression() {
    CNFExpression exp = new CNFExpression();
    for (int z = 1; z <= 9; z++) {
      for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
          for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
              for (int k = c + 1; k < 3; k++) {
                int r1 = 3 * i + r;
                int c1 = 3 * j + c;
                int c2 = 3 * j + k;
                String var1str = "(" + r1 + "," + c1 + "," + z + ")";
                String var2str = "(" + r1 + "," + c2 + "," + z + ")";
                Variable var1 = getVariable(var1str);
                Variable var2 = getVariable(var2str);
                ArrayList<Litteral> arr = new ArrayList<>();
```

```
                arr.add(new Litteral(var1, true));
                arr.add(new Litteral(var2, true));
                Clause clause = new Clause(arr);
                exp.concatClauses(clause);
              }
            }
          }
        }
      }
    }
    for (int z = 1; z <= 9; z++) {
      for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
          for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
              for (int k = r + 1; k < 3; k++) {
                for (int l = 0; l < 3; l++) {
                  int r1 = 3 * i + r;
                  int c1 = 3 * j + c;
                  int r2 = 3 * i + k;
                  int c2 = 3 * j + l;
                  String var1str = "(" + r1 + "," + c1 + "," + z +
                      ")";
                  String var2str = "(" + r2 + "," + c2 + "," + z +
                      ")";
                  Variable var1 = getVariable(var1str);
                  Variable var2 = getVariable(var2str);
                  ArrayList<Litteral> arr = new ArrayList<>();
                  arr.add(new Litteral(var1, true));
                  arr.add(new Litteral(var2, true));
                  Clause clause = new Clause(arr);
                  exp.concatClauses(clause);
                }

              }
            }
          }
        }
      }
    }
    return exp;
  }
```

```
private static CNFExpression getUniquenessRowExpression() {
  CNFExpression exp = new CNFExpression();
  for (int r = 0; r < 9; r++) {
    for (int n = 1; n <= 9; n++) {
      for (int ci = 0; ci < 9 - 1; ci++) {
        for (int cj = ci + 1; cj < 9; cj++) {
          String var1str = "(" + r + "," + ci + "," + n + ")";
          String var2str = "(" + r + "," + cj + "," + n + ")";
          Variable var1 = getVariable(var1str);
          Variable var2 = getVariable(var2str);
          ArrayList<Litteral> arr = new ArrayList<>();
          arr.add(new Litteral(var1, true));
          arr.add(new Litteral(var2, true));
          Clause clause = new Clause(arr);
          exp.concatClauses(clause);

        }
      }
    }
  }
  return exp;
}

private static CNFExpression getUniquenessColumnExpression() {
  CNFExpression exp = new CNFExpression();
  for (int c = 0; c < 9; c++) {
    for (int n = 1; n <= 9; n++) {
      for (int ri = 0; ri < 9 - 1; ri++) {
        for (int rj = ri + 1; rj < 9; rj++) {
          String var1str = "(" + ri + "," + c + "," + n + ")";
          String var2str = "(" + rj + "," + c + "," + n + ")";
          Variable var1 = getVariable(var1str);
          Variable var2 = getVariable(var2str);
          ArrayList<Litteral> arr = new ArrayList<>();
          arr.add(new Litteral(var1, true));
          arr.add(new Litteral(var2, true));
          Clause clause = new Clause(arr);
          exp.concatClauses(clause);
        }
      }
    }
```

```
    }
    return exp;
  }

  private static CNFExpression getSudokuExpression() {
    return new
        CNFExpression().concatClauses(columnUniquenessExpression())
         .concatClauses(rowUniquenessExpression())
         .concatClauses(getUniquenessBoxExpression().getClauses())
         .concatClauses(defineCellExpression())
         .concatClauses(getDefinessRowExpression().getClauses())
         .concatClauses(getDefinessColumnExpression().getClauses())
         .concatClauses(getDefinessBoxExpression().getClauses())
         .concatClauses(getCellUniqueness().getClauses());

  }

  private static ArrayList<Clause> rowUniquenessExpression() {
    return getUniquenessRowExpression().getClauses();
  }

  private static ArrayList<Clause> columnUniquenessExpression() {
    return getUniquenessColumnExpression().getClauses();
  }

  private static ArrayList<Clause> defineCellExpression() {
    return getDefinessCellExpression().getClauses();
  }
}
```

A.2.3 *Riduzione formula CNF*

La formula CNF viene ridotta, in quanto alcuni numeri nello schema del sudoku sono stati già inseriti. In particolar modo vengono rimosse le variabili che siamo sicuri abbiano già un certo valore di verità. Inoltre vengono rimosse le clausole che sono già soddisfatte da questo assegnamento iniziale parziale.

```
package satsudoku;

import java.util.ArrayList;
```

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;

import satsudoku.exception.NotClonableExpressionException;

public class ReduceSudokuCNFFormula {

  private SudokuSat sudoku;

  private CNFExpression exp;
  private ArrayList<SudokuVariable> vars;
  private HashSet<String> impossibleCell;

  public ReduceSudokuCNFFormula(SudokuSat sudoku) {
    this.sudoku = sudoku;
    this.vars =
        deepCopyVars(SudokuExpressionBuilder.getVariables());
    try {
      this.exp =
          SudokuExpressionBuilder.getGeneralSudokuCNFExpression().clone();
    } catch (CloneNotSupportedException e) {
      throw new
          NotClonableExpressionException(exp.getClass().getSimpleName()
          + " must implements Cloneable");
    }
    impossibleCell = new HashSet<>();
    simplifyExp();
  }

  private void simplifyExp() {
    int[][] numbers = sudoku.getNumbers();
    for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 9; j++) {
        if (numbers[i][j] != 0) {
          int number = numbers[i][j];
          String var = "(" + i + "," + j + "," + number + ")";
          vars.removeIf(variable ->
              variable.getVarRapresentation().equals(var));
          removeAffirmedVariableFromClauses(SudokuExpressionBuilder.getVariable(var));
          removeAllPossibleVariableWatching(new
              SudokuVariable(var));
```

```
      }

    }
  }
}

private void removeNegatedVariablesFromClauses(SudokuVariable
    var) {
  Iterator<Clause> itClauses = exp.getClauses().iterator();
  while (itClauses.hasNext()) {
    Clause clause = itClauses.next();
    Iterator<Litteral> itLitterals =
        clause.getLitterals().iterator();
    while (itLitterals.hasNext()) {
      Litteral litt = itLitterals.next();
      if (litt.getVar().getVarRapresentation()
      .equals(var.getVarRapresentation())) {
        itLitterals.remove();
        if (litt.isNegated()) {
          itClauses.remove();
          break;
        }
      }
    }
  }
}

private void removeAffirmedVariableFromClauses(SudokuVariable
    var) {
  Iterator<Clause> it = exp.getClauses().iterator();
  while (it.hasNext()) {
    Clause clause = it.next();
    Iterator<Litteral> itLitterals =
        clause.getLitterals().iterator();
    while (itLitterals.hasNext()) {
      Litteral litt = itLitterals.next();
      if (litt.getVar().getVarRapresentation()
      .equals(var.getVarRapresentation())) {
        itLitterals.remove();
        if (!litt.isNegated()) {
          it.remove();
          break;
```

```
            }
          }
        }
      }
    }

    private ArrayList<SudokuVariable>
        deepCopyVars(List<SudokuVariable> variables) {
      ArrayList<SudokuVariable> vars = new ArrayList<>();
      for (SudokuVariable var : variables) {
        try {
          vars.add(var.clone());
        } catch (CloneNotSupportedException e) {
          throw new
              NotClonableExpressionException(var.getClass().getSimpleName()
              + " must implement clone method.");
        }
      }
      return vars;
    }

    public SudokuSat getSudoku() {
      return sudoku;
    }

    public ArrayList<SudokuVariable> getVars() {
      return vars;
    }

    public CNFExpression getExp() {
      return exp;
    }

    public void removeAllPossibleVariableWatching(SudokuVariable var)
        {
      int r = var.getR();
      int c = var.getC();
      int v = var.getV();
      removeAllVariablesInCell(r, c, v);
      removeAllVariablesInRow(r, c, v);
      removeAllVariablesInColumn(r, c, v);
      removeAllVariablesInBox(r, c, v);
```

```
}

private void removeAllVariablesInCell(int r, int c, int v) {
  Iterator<SudokuVariable> it = vars.iterator();
  while (it.hasNext()) {
    SudokuVariable var = it.next();
    if (var.getR() == r && var.getC() == c && var.getV() != v) {
      if (impossibleCell.add(var.getVarRapresentation())) {
        removeNegatedVariablesFromClauses(var);
        it.remove();
      }
    }
  }

}

private void removeAllVariablesInBox(int r, int c, int v) {
  // GET Indexes
  int[] indexes = getBoxIndexes(r, c);

  for (int row = 0; row < 3; row++) {
    for (int col = 0; col < 3; col++) {
      int row1 = 3 * indexes[0] + row;
      int col1 = 3 * indexes[1] + col;
      Iterator<SudokuVariable> it = vars.iterator();
      while (it.hasNext()) {
        SudokuVariable var = it.next();
        if (var.getR() == row1 && var.getC() == col1 &&
            var.getV() == v) {
          if (impossibleCell.add(var.getVarRapresentation())) {
            removeNegatedVariablesFromClauses(var);
            it.remove();
          }
        }
      }
    }
  }

}

private int[] getBoxIndexes(int r, int c) {
```

```
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++) {
        for (int row = 0; row < 3; row++) {
          for (int col = 0; col < 3; col++) {
            int row1 = 3 * i + row;
            int col1 = 3 * j + col;
            if (row1 == r && col1 == c) {
              int[] arr = new int[2];
              arr[0] = i;
              arr[1] = j;
              return arr;
            }
          }
        }
      }
    }
    return null;
  }

  private void removeAllVariablesInColumn(int r, int c, int v) {
    Iterator<SudokuVariable> it = vars.iterator();
    while (it.hasNext()) {
      SudokuVariable var = it.next();
      if (var.getR() != r && var.getC() == c && var.getV() == v) {
        if (impossibleCell.add(var.getVarRapresentation())) {
          removeNegatedVariablesFromClauses(var);
          it.remove();
        }
      }
    }

  }

  private void removeAllVariablesInRow(int r, int c, int v) {
    Iterator<SudokuVariable> it = vars.iterator();
    while (it.hasNext()) {
      SudokuVariable var = it.next();
      if (var.getC() != c && var.getR() == r && var.getV() == v) {
        if (impossibleCell.add(var.getVarRapresentation())) {
          removeNegatedVariablesFromClauses(var);
          it.remove();
        }
```

```
      }
    }
  }

}
```

A.2.4 *Variabili di supporto*

```
package satsudoku;

import satsudoku.exception.SudokuException;

public class SudokuSat {

  private int[][] numbers;

  public SudokuSat(int[][] numbers) {
    if (numbers.length != numbers[0].length && numbers.length != 9)
        {
      throw new SudokuException("Sudoku is not 9x9 grid");
    }
    this.numbers = numbers;
  }

  public int[][] getNumbers() {
    return numbers;
  }

  public String toString() {
    StringBuilder string = new StringBuilder();
    for (int r = 0; r < 9; r++) {
      if (r % 3 == 0)
        string.append("#-------#-------#-------#" + "\n");
      for (int c = 0; c < 9; c++) {
        if (c % 3 == 0) {
          string.append("| ");
        }
        string.append(numbers[r][c] + " ");
        if (c == 8) {
          string.append("|" + "\n");
        }
```

```
        }

      }
      string.append("#-------#-------#-------#");
      return string.toString();
    }

    public void setRCV(int r, int c, int v) {
      this.numbers[r][c] = v;
    }

}
package satsudoku;

import satsudoku.exception.SudokuException;

public class SudokuSatSolver {

    private ReduceSudokuCNFFormula reductor;
    private SatSolver sat;

    public SudokuSatSolver(SudokuSat sudoku) {
      reductor = new ReduceSudokuCNFFormula(sudoku);
      sat = new SatSolver(reductor.getVars());
    }

    public ReduceSudokuCNFFormula getReductor() {
      return reductor;
    }

    public SudokuSat sudokuSolution() {
      sat.findAssignment(reductor.getExp());
      String result = sat.getResult();
      if(result==null) {
        throw new SudokuException("Sudoku not solvable...");
      }
      for (int i = 0; i < result.length(); i++) {
        if (result.charAt(i) == '1') {
          String str = sat.getVariablesMap().get(i);
          SudokuVariable var = new SudokuVariable(str);
          int r = var.getR();
          int c = var.getC();
```

```
          int v = var.getV();
          reductor.getSudoku().setRCV(r, c, v);
        }
      }
      return reductor.getSudoku();
    }

}
package satsudoku.exception;

public class NotClonableExpressionException extends
    RuntimeException {

  private static final long serialVersionUID =
      -6705887492282926112L;

  public NotClonableExpressionException(String message) {
    super(message);
  }

}
package satsudoku.exception;

public class SudokuException extends RuntimeException {

  private static final long serialVersionUID =
      -5524999724270067642L;

  public SudokuException(String message) {
    super(message);
  }
}
```

A.3 MAIN PER TESTARE

A.3.1 *Brute Force*

Qui sotto viene riportato il main contenente alcuni esempi che vedono sfruttare le tecniche che abbiamo definito.

```
package sudoku_nxn;
```

```
public class Main {

  public static void main(String[] args) throws
      CloneNotSupportedException {
    int[][] sudoku1 = { { 2, 0, 0, 0, 7, 0, 0, 3, 8 }, { 0, 0, 0,
        0, 0, 6, 0, 7, 0 }, { 3, 0, 0, 0, 4, 0, 6, 0, 0 },
         { 0, 0, 8, 0, 2, 0, 7, 0, 0 }, { 1, 0, 0, 0, 0, 0, 0, 0, 6
             }, { 0, 0, 7, 0, 3, 0, 4, 0, 0 },
         { 0, 0, 4, 0, 8, 0, 0, 0, 9 }, { 0, 6, 0, 4, 0, 0, 0, 0, 0
             }, { 9, 1, 0, 0, 6, 0, 0, 0, 2 } };
    int[][] sudoku2 = { { 2, 0, 7, 0, 0, 9, 1, 3, 4 }, { 0, 4, 0,
        2, 1, 0, 0, 6, 7 }, { 3, 0, 0, 7, 6, 4, 8, 0, 9 },
         { 0, 8, 2, 4, 3, 0, 7, 0, 6 }, { 0, 7, 9, 8, 2, 1, 4, 5, 0
             }, { 4, 0, 3, 0, 7, 6, 2, 1, 0 },
         { 5, 0, 6, 3, 4, 7, 0, 0, 1 }, { 7, 9, 0, 0, 5, 8, 0, 4, 0
             }, { 8, 3, 4, 1, 0, 0, 6, 0, 5 } };
    int[][] sudoku3 = { { 0, 1, 7, 9, 0, 3, 6, 0, 0 }, { 0, 0, 0,
        0, 8, 0, 0, 0, 0 }, { 9, 0, 0, 0, 0, 0, 5, 0, 7 },
         { 0, 7, 2, 0, 1, 0, 4, 3, 0 }, { 0, 0, 0, 4, 0, 2, 0, 7, 0
             }, { 0, 6, 4, 3, 7, 0, 2, 5, 0 },
         { 7, 0, 1, 0, 0, 0, 0, 6, 5 }, { 0, 0, 0, 0, 3, 0, 0, 0, 0
             }, { 0, 0, 5, 6, 0, 1, 7, 2, 0 } };
    int[][] sudoku4 = { { 4, 0, 0, 0, 0, 0, 9, 3, 8 }, { 0, 3, 2,
        0, 9, 4, 1, 0, 0 }, { 0, 9, 5, 3, 0, 0, 2, 4, 0 },
         { 3, 7, 0, 6, 0, 9, 0, 0, 4 }, { 5, 2, 9, 0, 0, 1, 6, 7, 3
             }, { 6, 0, 4, 7, 0, 3, 0, 9, 0 },
         { 9, 5, 7, 0, 0, 8, 3, 0, 0 }, { 0, 0, 3, 9, 0, 0, 4, 0, 0
             }, { 2, 4, 0, 0, 3, 0, 7, 0, 9 } };

    int[][] sudoku5 = { { 0, 0, 0, 0, 0, 0, 0, 0, 0 }, { 0, 0, 0,
        0, 0, 3, 0, 8, 5 }, { 0, 0, 1, 0, 2, 0, 0, 0, 0 },
         { 0, 0, 0, 5, 0, 7, 0, 0, 0 }, { 0, 0, 4, 0, 0, 0, 1, 0, 0
             }, { 0, 9, 0, 0, 0, 0, 0, 0, 0 },
         { 5, 0, 0, 0, 0, 0, 0, 7, 3 }, { 0, 0, 2, 0, 1, 0, 0, 0, 0
             }, { 0, 0, 0, 0, 4, 0, 0, 0, 9 } };

    int[][] sudoku6 = { { 0, 0, 0, 0, 0, 0, 0, 0, 0 }, { 0, 4, 0,
        0, 6, 0, 0, 2, 0 }, { 7, 6, 0, 0, 4, 0, 0, 9, 5 },
         { 0, 0, 0, 5, 0, 3, 0, 0, 0 }, { 2, 1, 0, 0, 0, 0, 0, 4, 8
             }, { 0, 0, 0, 4, 0, 8, 0, 0, 0 },
         { 4, 2, 0, 0, 7, 0, 0, 1, 9 }, { 0, 9, 0, 0, 3, 0, 0, 7, 0
```

```
            }, { 0, 0, 0, 0, 0, 0, 0, 0, 0 } };

    int[][] sudoku7 = { { 8, 0, 0, 0, 0, 0, 0, 0, 0 }, { 0, 0, 3,
        6, 0, 0, 0, 0, 0 }, { 0, 7, 0, 0, 9, 0, 2, 0, 0 },
         { 0, 5, 0, 0, 0, 7, 0, 0, 0 }, { 0, 0, 0, 0, 4, 5, 7, 0, 0
            }, { 0, 0, 0, 1, 0, 0, 0, 3, 0 },
         { 0, 0, 1, 0, 0, 0, 0, 6, 8 }, { 0, 0, 8, 5, 0, 0, 0, 1, 0
            }, { 0, 9, 0, 0, 0, 0, 4, 0, 0 } };
    int[][] sudoku8 = { { 0, 0, 6, 0, 0, 0, 4, 0, 0 }, { 0, 0, 0,
        0, 5, 0, 0, 7, 0 }, { 0, 7, 0, 1, 0, 0, 0, 3, 0 },
         { 8, 0, 0, 0, 7, 9, 0, 0, 6 }, { 0, 6, 0, 3, 0, 1, 0, 5, 0
            }, { 7, 0, 0, 6, 2, 0, 0, 0, 4 },
         { 0, 9, 0, 0, 0, 7, 0, 2, 0 }, { 0, 3, 0, 0, 6, 0, 0, 0, 0
            }, { 0, 0, 8, 0, 0, 0, 9, 0, 0 } };
    int[][] sudoku11 = {{1,0,0,0},{2,3,0,4},{0,0,3,2},{0,0,0,0}};
    System.out.println((solveSudoku(sudoku1)));
}

private static String solveSudoku(int[][] sudoku1) throws
    CloneNotSupportedException {
    Sudoku sudoku = new Sudoku(sudoku1);
    SudokuTechniques techs = createSudokuTechniques(sudoku);
    SudokuSolver sudokuSol = new SudokuSolver(3, techs);
    sudoku = sudokuSol.sudokuSolution(sudoku);
    return sudoku.toString();
}

private static SudokuTechniques createSudokuTechniques(Sudoku
    sudoku) {
    EasyCellSudokuTecnique tech1 = new EasyCellSudokuTecnique();
    HiddenSingleSudokuTechnique tech2 = new
        HiddenSingleSudokuTechnique();
    NTupleSudokuTechnique tech3 = new NTupleSudokuTechnique();
    PointingSudokuTechnique tech4 = new PointingSudokuTechnique();
    SudokuTechniques techs =
        SudokuTechniques.Builder.newInstance(tech1)
            .addTechnique(tech2)
            .addTechnique(tech3)
            .addTechnique(tech4)
            .build();
    return techs;
}
```

```
}
```

A.3.2 *Sat Solver*

Per evitare ripetizioni, gli schemi utilizzati sono i soliti dell'algoritmo di brute force.

```
\\use previous sudoku schemes
SudokuSat sudokusol = new SudokuSat(sudoku1);
     SudokuSatSolver sud = new SudokuSatSolver(sudokusol);
     long start = System.nanoTime();
     System.out.println(sud.sudokuSolution());
     long end = System.nanoTime();
     System.out.println("Time in ms: " + (end - start) / 1000000);
```