Corso di laurea in Informatica
Progettazione d'algoritmi

Didattica blended

Backtracking III

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Il problema di ottimizzazione dello Zaino:
Abbiamo n oggetti, ciascuno caratterizzato da un peso p_i ed un valore v_i. Abbiamo inoltre uno zaino di capacità C.
Tra tutti i possibili sottoinsiemi degli n oggetti vogliamo trovare quello il cui peso non supera C e il cui valore sia massimo.
Il peso ed il valore di un insieme di oggetti sono dati dalla somma dei pesi e dalla somma dei valori degli oggetti dell'insieme rispettivamente.

Progettare un algoritmo che prende in input la lista dei valori degli n oggetti e la lista dei pesi degli n oggetti e la capacità dello zaino e restituisce la soluzione al problema (la lista con gli oggetti da inserire nello zaino per massimizzarne il valore).

Esempio: **Considera la seguente istanza con $C = 166$ e $n = 10$ oggetti con peso e valore riportati in tabella**

oggetto	peso	valore
0	23	92
1	31	57
2	29	49
3	89	87
4	44	68
5	53	60
6	38	43
7	63	67
8	85	84
9	82	72

La soluzione ottima pesa 165, vale 309 ed il suo vettore caratteristico è SOL=$[1,\ 1,\ 1,\ 0,\ 1,\ 0,\ 1,\ 0,\ 0,\ 0]$

Idea1:

- Possiamo risolvere il problema ricorrendo ad un algoritmo esaustivo:

  - generiamo tutti i possibili sottoinsiemi degli n oggetti. Per ognuno verifichiamo se è una soluzione ammissibile (vale a dire se il peso del sottoinsieme di oggetti non eccede la capacità dello zaino) e, in questo caso, ne calcoliamo il valore. Teniamo traccia della soluzione ammissibile di valore massimo via-via trovata. Al termine diamo in outuput il massimo trovato.

Poichè i sottoinsiemi da considerare per n oggetti sono 2^n l'algoritmo esaustivo avrà complessità esponenziale $\Omega(2^n)$.

Algoritmo esaustivo:

```
def zaino(P,V,C):
    #
    def bk(n,i,sol):
        if i==n:
            nonlocal val_ottimo, sol_ottima
            #verifico se il sottoinsieme sol e' ammissibile
            if sum([P[i] for i in range(n) if sol[i]==1])<=C:
                #verifico se la soluzione ammissibile e' ottima
                val=sum([V[i] for i in range(n) if sol[i]==1])
                if val>val_ottimo:
                    #aggiorno la soluzione ottima
                    sol_ottima=sol.copy()
                    val_ottimo=val
        else:
            sol.append(0)
            bk(n,i+1,sol)
            sol.pop()
            sol.append(1)
            bk(n,i+1,sol)
            sol.pop()
    #
    sol_ottima,val_ottimo,n=[],-1,len(P)
    bk(n,0,[])
    print(sol_ottima)
```

la complessità è $\Theta\left(n2^n\right)$

Non è difficile modificare il codice in modo che ogni nodo sappia il valore e il peso parziale della soluzione che si sta costruendo.
In questo modo i nodi foglia possono scoprire se bisogna aggiornare la soluzione ottima in tempo $O(1)$.

Nota che il numero di nodi interni da generare resta $\Theta(2^n)$ e l'asintotica dell'algoritmo risulta ora $\Theta\left(2^n\right)$

Cosa più importante: in questo modo ogni nodo interno sa quanto vale e quanto pesa la soluzione parziale costruita fino a quel momento.
Queste informazioni possono risultare utili in vista dell'introduzione di eventuali funzioni di taglio.

Algoritmo esaustivo:

```
def zaino(P,V,C):
    #
    def bk(n, val, peso, i, sol):
        if i==n:
            nonlocal val_ottimo, sol_ottima
            #verifico se il sottoinsieme sol e’ ammissibile
            if peso <=C:
                #verifico se la soluzione ammissibile e' ottima
                if val>val_ottimo:
                    #aggiorno la soluzione ottima
                    sol_ottima=sol.copy()
                    val_ottimo=val
        else:
            sol.append(0)
            bk(n, val, peso, i+1, sol)
            sol.pop()
            sol.append(1)
            bk(n, val+V[i], peso+P[i], i+1, sol)
            sol.pop()
    #
    sol_ottima, val_ottimo, n = [], -1, len(P)
    bk(n,0,0,0,[])
    print(sol_ottima)
```

la complessità è $\Theta\left(n2^n\right)$

Idea2: ricorriamo ad una algoritmo di backtraking introducendo funzioni di taglio che permetteranno di potare alcuni dei 2^n nodi presenti nell'albero di ricorsione.

- nostro caso procederemo come segue:
  1. potiamo i nodi che non portano a soluzioni ammissibili.
  2. potiamo nodi che se anche portassero a soluzioni ammissibili queste non sarebbero tali da migliorare la soluzione ottima fino a quel momento ottenuta.

- la funzione di taglio del punto 1 verrà applicata al figlio destro di ogni nodo (dove si decide di prendere l'oggetto).
- la funzione di taglio del punto 2 verrà applicata al figlio sinistro di ogni nodo (dove si decide di non prendere l'oggetto).

Algoritmo di backtracking in cui è stata implementata la prima delle due funzioni di taglio:

```
def zaino1(P,V,C):
    #
    def bk(n, val, peso, i, sol):
        if i==n:
            nonlocal val_ottimo, sol_ottima
            #verifico se la soluzione ammissibile e' ottima
            if val>val_ottimo:
                #aggiorno la soluzione ottima
                sol_ottima=sol.copy()
                val_ottimo=val
        else:
            sol.append(0)
            bk(n, val, peso, i+1, sol)
            sol.pop()
            if peso+P[i]<=C:
                sol.append(1)
                bk(n, val+V[i], peso+P[i], i+1, sol)
                sol.pop()
    #
    sol_ottima, val_ottimo, n = [], -1, len(P)
    bk(n,0,0,0,[])
    print(sol_ottima)
```

la complessità è $O\left(n2^n\right)$

Resta da implementare la seconda funzione di taglio:

- potare quei nodi che se anche portassero a soluzioni ammissibili queste non sarebbero tali da migliorare la soluzione ottima fino a quel momento ottenuta.

Un modo poco costoso per raggiungere parzialmente quest'obiettivo vien fuori dalla seguente semplice osservazione:

- se al livello i della ricorsione decido di non inserire l'oggetto i nello zaino, nessuna delle soluzioni ammissibili che sarà possibile ottenere potrà valere più di $val + \sum_{j=i+1}^{n-1} v_j$

- Di conseguenza se $val + \sum_{j=i+1}^{n-1} v_j \leq val\_ottimo$ possiamo potare l'albero di ricorsione evitando di porre $sol[i] = 0$

NOTE:

1. l'obiettivo è raggiunto solo parzialmente perchè $val + \sum_{j=i+1}^{n-1} v_j > val\_ottimo$ non implica che le soluzioni ammissibili che sarà possibile raggiungere saranno in grado di migliorare la soluzione ottima.

2 nel codice che segue per rendere efficiente il calcolo di questa funzione di taglio faremo si che ogni nodo dell'albero a livello i riceva come informazione $valRim$ con l'informazione sul valore totale degli oggetti restanti (vale a dire $\sum_{j=i}^{n-1} v_j$)

Algoritmo di backtracking in cui sono state implementate entrambe le funzioni di taglio:

```
def zaino2(P,V,C):
    #
    def bk(n, valRim,  val, peso, i, sol):
        if i==n:
            #aggiorno la soluzione ottima
            sol_ottima=sol.copy()
            val_ottimo=val
        else:
            if val+valRim-V[i]>val_ottimo:
                sol.append(0)
                bk(n, valRim-V[i], val, peso, i+1, sol)
                sol.pop()
            if peso+P[i]<=C:
                sol.append(1)
                bk(n,valRim-V[i], val+V[i], peso+P[i], i+1, sol)
                sol.pop()
    #
    sol_ottima, val_ottimo, n = [], -1, len(P)
    valRim=sum(V)
    bk(n,valRim,0,0,0,[])
    print(sol_ottima)
```

la complessità è $O\left(n2^n\right)$

Per il problema dello zaino quindi abbiamo visto 3 algoritmi:

- $Zaino(P, V, C)$**: algoritmo di ricerca esaustiva.**
- $Zaino1(P, V, C)$**: algoritmo di backtraking con una funzione di taglio.**
- $Zaino2(P, V, C)$**: algoritmo di backtraking con due funzioni di taglio.**

L'efficacia delle funzioni di taglio per questo tipo di algoritmi può essere validata sperimentalmente.

Ad esempio: possiamo introdurre nei codici dei 3 algoritmi un contatore dei nodi interni dell'albero di ricorsione effettivamente generati nel corso della loro esecuzione.

Eseguendo i tre algoritmi sull'istanza d'esempio con gli $n = 10$ oggetti, vale a dire:
P=$[23, 31, 29, 89, 44, 53, 38, 63, 85, 82]$
V=$[92, 57, 49, 87, 68, 60, 43, 67, 84, 72]$
C=166
per i $2^n - 1 = 1023$ nodi interni dell'albero di ricorsione si ha:

- Per $Zaino(P, V, C)$ vengono visitati 1023 su 1023 nodi interni
- Per $Zaino1(P, V, C)$ vengono visitati 402 su 1023 nodi interni
- Per $Zaino2(P, V, C)$ vengono visitati 371 su 1023 nodi interni

Note:

- la scelta delle funzioni di taglio. In genere migliore è la funzione di taglio più questa risulta computazionalmente onerosa. Considerando che ogni nodo dell'albero di ricorsione dovrà applicarla, non sempre funzioni in grado di tagliare di più producono algoritmi più veloci.

- le soluzioni da cui partire. L'effettività di alcune funzioni di taglio può dipende dalla qualità della soluzione ottima prodotta fino a quel momento. Un esempio di ciò si ha con la seconda funzione di taglio che abbiamo utilizzato per lo zaino.
  In questi casi può essere fruttuoso far partire il backtraking da una soluzione di buona qualità che poi il backtraking tenterà di migliorare.
  Per ottenere *"buone"* soluzioni da cui partire è possibile ricorrere, in una fase di preprocessing, ad algoritmi approssimanti che garantiscono di produrre soluzioni di buona qualità (anche se non ottime) in tempi contenuti.

ESEMPIO:

Il problema delle n regine. Data una scacchiera $n \times n$ vogliamo calcolare quanti diversi modi ci sono di disporre sulla scacchiera n regine in modo che nessuna di esse sia in grado di catturarne un'altra in base alle regole degli scacchi. Perció una disposizione dovrà prevedere che nessuna delle n regine abbia colonna, riga o diagonale in comune con un altra.

Ad esempio, ecco due delle possibile disposizioni per $n = 8$:

							x
	x						
			x				
x							
						x	
				x			
		x					
					x		

			x				
	x						
						x	
		x					
					x		
							x
x							
				x			

Bisogna quindi progettare una funzione di backtracking che prende come parametro l'intero n e restituisce il numero di disposizioni lecite possibili.

- Non è difficile convincersi che per $n = 2$ ed $n = 3$ non esiste alcuna disposizione lecita.
- Per i primi valori di n la funzione deve dare le seguenti risposte:

n	disposizioni
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712
14	365596
15	2279184

Le disposizioni che vogliamo contare soddisfano 3 vincoli:

1. **le n regine sono su righe distinte**
2. **le n regine sono su colonne distinte**
3. **le n regine sono su diagonali distinte**

- I primi due vincoli fanno si che in ciascuna delle disposizioni da contare in ogni riga della scacchiera ci sarà esattamente una regina e lo stesso varrà per ogni colonna.
- Questo significa che posso vedere ciascuna delle disposizioni da contare come una permutazione dei numeri da 0 a $n-1$ dove in posizione i compare la colonna in cui è stata sistemata la regina della i-esima riga.

							x	7
	x							1
			x					3
x								0
						x		6
				x				4
		x						2
					x			5

$= [7, 1, 3, 0, 6, 4, 2, 5]$

Una disposizione che soddisfa i primi due vincoli del problema (vale a dire regine in righe e colonne distinte) può dunque essere vista come una permutazione dei primi n interi.

Non tutte le permutazioni corrispondono alle disposizioni lecite che vogliamo contare.

Esempio di permutazione che non corrisponde ad una disposizione lecita per il problema delle n regine (la regina sulla riga 2 e quella sulla riga 5 condividono la stessa diagonale)

$[7, 1, 3, 0, 4, 6, 2, 5] =$

7								x
1		x						
3				x				
0	x							
4					x			
6							x	
2			x					
5						x		

Possiamo modificare il programma per generare e stampare tutte le permutazioni dei primi n interi in modo che, anzicché stampare, conti le permutazioni. Aggiungeremo poi a questa versione del programma delle funzioni di taglio in modo che vengano generate (e quindi contate) solo permutazioni corrispondenti a disposizioni lecite.

Programma di backtraking che conta le disposizioni in cui le n regine non sono mai nella stessa riga e mai nella stessa colonna (Per quanto detto la risposta sarà $n!$):

```
def regine1(n):
    return regine1R(n,0,[],set())

def regine1R(n,r, sol, presi):
    tot=0
    if r==n: return 1
    for c in range(n):
        if c not in presi:
            sol.append(c)
            presi.add(c)
            tot+=regine1R(n,r+1,sol,presi)
            sol.pop()
            presi.remove(c)
    return tot


>>> regine1(3)
6
>>> regine1(5)
120
```

Bisogna ora aggiungere al codice i vincoli attinenti alle diagonali:
la posizione che si vuole assegnare ad una nuova regina non deve appartenere ad una diagonale già occupata

Ogni cella della scacchiera appartiene a due diagonali: una diagonale che va da sinistra a destra ed un altra diagonale che va da destra a sinistra.

Serve un metodo efficiente per ricavare dalla posizione in cui vorrei sistemare la regina le due diagonali cui la cella appartiene:

- **due celle (i, j) e (i', j') sono nella stessa diagonale da sinistra a destra se e solo se $i - j = i' - j'$**
- **due celle (i, j) e (i', j') sono nella stessa diagonale da destra a sinistra se e solo se $i + j = i' + j'$**

Ad esempio di seguito per $n = 8$ vengono evidenziate:

- in blu la diagonale destra-sinista di valore $i + j = 4$
- in rosso la diagonale sinistra-destra di valore $i - j = -2$

		$(0,2)$		$(0,4)$			
			$(1,3)$				
		$(2,2)$		$(2,4)$			
	$(3,1)$				$(3,5)$		
$(4,0)$						$(4,6)$	
							$(5,7)$

- **due celle (i, j) e (i', j') sono nella stessa diagonale da sinistra a destra se e solo se $i - j = i' - j'$**
- **due celle (i, j) e (i', j') sono nella stessa diagonale da destra a sinistra se e solo se $i + j = i' + j'$**

In un insieme S_D posso tener traccia delle diagonali che vanno da sinistra a destra già occupate mentre in un insieme D_S posso tener traccia delle diagonali già occupate che vanno da destra a sinistra.

Per sapere dunque se la diagonale sinistra-destra della cella (i, j) è già occupata basta vedere se (i, j) è in S_D.

Analogamente, per sapere se la diagonale destra-sinistra della cella (i, j) è già occupata basta vedere se $i + j$ è in D_S.

```
def regine2(n):
    return regine2R(n, 0, [ ], set( ), set( ), set( ))

def regine2R(n, r, sol, presi, S_D, D_S ) :
    tot = 0
    if r == n : return 1
    for c in range(n) :
        if c not in presi and \
          r − c not in S_D and r + c not in D_S:
            sol.append(c); presi.add(c)
            S_D.add(r − c)
            D_S.add(r + c)
            tot+ = regine2R(n, r + 1, sol, presi, S_D, D_S )
            sol.pop(); presi.remove(c)
            S_D.remove(r − c)
            D_S.remove(r + c)
    return tot
```

ESEMPIO:

Il Sudoku è una griglia di 9×9 celle. Il giocatore aggiunge i numeri da 1 a 9 ai campi vuoti in modo che ogni numero deve comparire una volta sola in:

- ogni riga
- ogni colonna
- ogni settore 3 x 3.

Esempio di istanza:

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

ESEMPI:

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

$\Rightarrow$

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

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

$\Rightarrow$

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

Progettare un algoritmo che prende come parametro una matrice 9×9 che codifica uno schema di sudoku da risolvere (lo 0 indica il campo vuoto) e restituisce lo schema risolto.

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	2	8	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

$\Rightarrow$

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

Versione backtraking in cui viene implementato solo il primo vincolo che impone che ciascuna riga sia una permutazione:

```
def sudoku(M):
    #
    def bk(i,j,sol):
        if i==9: return sol
        else:
            if M[i][j]!=0:
                if j<8: a=bk(i,j+1,sol)
                else: a= bk(i+1,0,sol)
                if a: return a
            else:
                for k in range(1,10):
                    if k not in riga[i]:
                        sol[i][j]=k
                        riga[i].add(k)
                        if j<8: a=bk(i,j+1,sol)
                        else: a=bk(i+1,0,sol)
                        if a: return a
                        riga[i].remove(k)
        return None
    #
    sol=[ [0 for _ in range(9)] for _ in range(9)]
    riga={ i:set() for i in range(0,9)}
    for i in range(9):
        for j in range(9):
            if M[i][j]!=0 :
                sol[i][j]= M[i][j]
                riga[i].add(M[i][j])
    print(bk(0,0,sol))
```

La soluzione che si ottiene quando in ogni riga ciascuno dei 9 numeri compare un'unica volta.

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	2	8	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

$\Rightarrow$

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

Versione backtraking in cui vengono implementati solo i primi due vincoli che impongono che ciascuna riga e ciascuna colonna sia una permutazione (in rosso sono evidenziate le istruzioni aggiunte rispetto alla prima versione del programma)

```
def sudoku(M) :

    def bk(i, j) :
        if i == 9 : return sol
        else:
            if M[i][j]! = 0 :
                if j < 8 : a = bk(i, j + 1)
                else: a = bk(i + 1, 0)
                if a : return a
            else:
                for k in range(1, 10) :
                    if k not in riga[i] and k not in col[j]
                        sol[i][j] = k
                        riga[i].add(k)
                        col[j].add(k)
                        if j < 8: a = bk(i, j + 1)
                        else:a = bk(i + 1, 0)
                        if a : return a
                        riga[i].remove(k)
                        col[j].remove(k)
        return None

    sol = [ [0 for _ in range(9)] for _ in range(9) ]
    riga = { i : set( ) for i in range(0, 9) }
    col = { i : set( ) for i in range(0, 9) }
    for i in range(9) :
        for j in range(9) :
            if M[i][j]! = 0 :
                sol[i][j] = M[i][j]
                riga[i].add(M[i][j])
                colonna[j].add(M[i][j])
    print(bk(0, 0))
```

La soluzione che si ottiene quando in ogni riga e in ogni colonna ciascuno dei 9 numeri compare un'unica volta.

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	2	8	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

$\Rightarrow$

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

Bisogna ora aggiungere al codice il vincolo attinente ai 9 settori dello schema:
il numero che si vuole assegnare ad una cella non deve essere stato già assegnato ad un'altra cella dello stesso settore.

Serve un metodo efficiente per ricavare dalla cella il suo settore:

Nota che la cella (i, j) appartiene al settore $3 \cdot \left\lfloor \frac{i}{3} \right\rfloor + \left\lfloor \frac{j}{3} \right\rfloor$.

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8

Versione backtracking in cui vengono implementati i tre vincoli che assicurano che ciascuna riga, ciascuna colonna e ciascun settore è una permutazione (in rosso sono evidenziate le istruzioni aggiunte rispetto alla versione precedente del programma).

```
def sudoku(M) :

    def bk(i, j) :
        if i == 9 : return sol
        if M[i][j]! = 0 :
            if j < 8 : a = bk(i, j + 1)
            else a = bk(i + 1, 0)
            if a : return a
        else:
            for k in range(1, 10) :
                if k not in riga[i] and k not in col[j]   \ and
                   k not in settore[ 3 * (i//3) + j//3 ] :
                    sol[i][j] = k
                    riga[i].add(k)
                    col[j].add(k)
                    settore[ 3 * (i//3) + j//3 ].add(k)
                    if j < 8 : a = bk(i, j + 1)
                    else: a = bk(i + 1, 0)
                    if a : return a
                    riga[i].remove(k)
                    col[j].remove(k)
                    settore[ 3 * (i//3) + j//3 ].remove(k)
        return None

    sol = [ [0 for _ in range(9)] for _ in range(9) ]
    riga = { i : set( ) for i in range(0, 9) }
    col = { i : set( ) for i in range(0, 9) }
    settore = { i : set( ) for i in range(0, 9) }
    for i in range(9) :
        for j in range(9) :
                if M[i][j]! = 0 :
                    sol[i][j] = M[i][j]
                    riga[i].add(M[i][j])
                    colonna[j].add(M[i][j])
                    settore[ 3 * (i//3) + j//3 ].add(M[i][j])
    print(bk(0, 0))
```

La soluzione che si ottiene nella versione in cui tutti e tre i vincoli sono verificati:

5	3	0	0	7	0	0	0	0
6	0	0	1	9	5	0	0	0
0	9	8	0	0	0	0	6	0
8	0	0	0	6	0	0	0	3
4	0	0	8	0	3	0	0	1
7	0	0	0	2	0	0	0	6
0	6	0	0	0	0	2	8	0
0	0	0	4	1	9	0	0	5
0	0	0	0	8	0	0	7	9

$\Rightarrow$

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