Implementazione algoritmo *MAC* utilizzando la tecnica *AC3*

di Umberto Farsetti

1. **Introduzione**
   Ho implementato l'algoritmo *MAC* (*maintaining arc consistency*) che riceve in input un problema di soddisfacimento dei vincoli e ne calcola la soluzione (se esiste). In particolare *MAC* utilizza al suo interno, oltre che *backtracking*, anche la tecnica *AC3* per la propagazione dei vincoli

2. **Strumenti di lavoro utilizzati**
   Per l'implementazione ho utilizzato il linguaggio *Python* senza però far uso di particolari librerie

3. **Algoritmo**
   In aggiunta a *backtraking* e *AC3* ho implementato anche l'euristica *fail first* con *minimum remaining values* che seleziona tra le variabili non ancora assegnate quella con meno valori possibili nel dominio

4. **Codice**
   Il programma è un risolutore di problemi CSP e ogni possibile problema si basa quindi su 3 elementi: variabili, domini e vincoli.
   All'interno della classe CSP vengono aggiunti e definiti i possibili problemi che risultano quindi sottoclassi di CSP.
   Una volta eseguito *problem = CSP(n, tipo)* si assegna alla variabile *problem* una lista contenente tutte le informazioni necessarie per la risoluzione del problema. La funzione *prepare_game(n, problem)* infatti si occupa di preparare il problema scelto settando variabili, domini e vincoli relativi a quel preciso caso. Se per i primi 2 non ci sono molti dettagli da specificare (in genere sono entrambe 2 liste da 0 a n) i vincoli richiedono delle dichiarazioni particolari.
   Il vincolo infatti non è altro che una funzione che, ricevute in ingresso 2 variabili e la relazione che le lega (esempio: *notEqual*), restituisce TRUE o FALSE se la relazione è rispettata o no. Per questo motivo ho pensato di strutturare i vincoli come dei dizionari in cui inserire la coppia di variabili in gioco e la loro relazione. Per esempio:
   {'index': (0, 1), 'condition': 'notEqual'}
   specifica un vincolo di non uguaglianza tra le variabili 0 e 1. Ovviamente ogni volta che si aggiunge una nuova *condition* questa va necessariamente dichiarata all'interno del metodo *check* della classe *Constraint* per essere controllata in fase di *revise*.

Creato il problema si esegue *backtrack* che riceve in ingresso il problema stesso e l'assegnamento delle variabili (inizialmente tutte FALSE).
La funzione esegue sostanzialmente le stesse azioni dell'algoritmo aggiungendo solo alcuni elementi necessari per eseguire un salto all'indietro in caso di errato assegnamento. Infatti ad ogni variabile è associato un dominio che come sappiamo viene modificato se vengono rilevate inconsistenze dopo un assegnamento. Se però uno di questi domini si riduce all'insieme vuoto si ha un fallimento e dobbiamo ripristinare i valori al punto precedente la scelta. Per fare questo si utilizza una semplice lista chiamata *storyDom* che contiene la storia dei domini di ogni singola variabile. Ogni volta che viene eseguito correttamente *AC3* il nuovo insieme dei domini delle variabili viene aggiunto a *storyDom* , mentre in caso contrario viene ripristinato l'insieme precedente.
In ogni caso chiamando *backtrack* i passaggi sono gli stessi dell'algoritmo:

- *isComplete(problem)* controlla se l'assegnamento risolve il problema restituendo TRUE o FALSE
- si associa ad *assegnedId* l'indice della variabile da assegnare. La scelta viene eseguita con la funzione *selectVar* che esegue l'euristica *fail first* con *minimum remaining values*
- si verifica se ogni possibile valore della variabile *assegnedId* è consistente con l'assegnamento. Se lo è si esegue *AC3* (e in seguito *revise*) esattamente come vogliono gli algoritmi. Se *AC3* restituisce TRUE si esegue ancora *backtrack* passando il nuovo assegnamento, altrimenti, se il valore della variabile non era corretto, si ripristinano i domini precedenti e si passa al valore successivo
- quando i valori terminano, *backtrack* restituisce FALSE. Le ultime righe di *backtrack* servono per ripristinare i domini precedenti delle variabili che erano inevitabilmente cambianti per effetto della scelta effettuata che a questo punto risulta errata

La funzione *AC3* esegue come sempre: crea inizialmente una lista *queue* contenente i vincoli tra la variabile scelta e le altre (sue vicine) non ancora assegnate (funzione *constAndNeighNotAss*) e per ognuno di questi ne estrapola indici e relazione (*condition*) passandoli in ingresso alla funzione *revise*. Se viene azzerato il dominio del vicino della variabile scelta restituisce FALSE, altrimenti, aggiunge in coda a *queue* i vincoli tra la variabile in gioco e le altre (sue vicine) non ancora assegnate (funzione *constAndNeighNotAss*).

Infine la funzione *revise* non fa altro che chiamare un controllo sulle variabili in gioco andando così a cercare se esiste un valore che crea inconsistenza. La funzione *check* contiene al suo interno tutti i possibili metodi per verificare se la *condition* (la relazione tra le 2 variabili) è rispettata.

Infine è importante notare come *AC3* lavori sempre con vincoli binari dato che il tutto si basa sulla definizione di consistenza tra 2 variabili. Quando ci troviamo di fronte un vincolo non binario il programma esegue una conversione introducendo per ogni vincolo n-ario una variabile aggiuntiva di supporto che definisce la relazione che lega le variabili in gioco (*binarization*). Per capire possiamo vedere un rapido esempio: si vogliono assegnare 3 valori diversi a 3 variabili tale che:

$X_1 + X_2 + X_3 = 6$ con $X_1, X_2, X_3 = [1, 2, 3]$

In questo caso si aggiunge una variabile Y che avrà per dominio tutte le possibili combinazioni dei singoli valori che rendono vera la relazione:

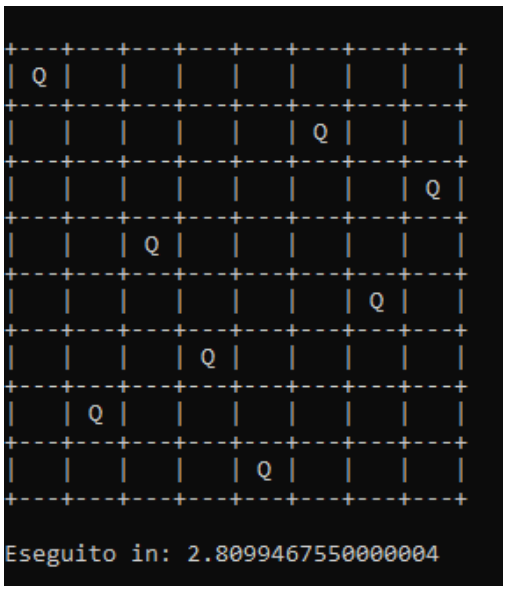
[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

Così facendo X_1 sarà legato a Y da un vincolo binario che impone ai possibili valori di X_1 di trovarsi nella prima posizione dei singoli vettori di Y. Lo stesso chiaramente vale anche per le altre variabili che dovranno però trovarsi in posizione 2 e 3. Quando verrà selezionato il primo valore, per esempio $X_1 = 1$, il dominio di Y si riduce, in fase di *revise*, mantenendo solo quelle possibili combinazioni in cui l'elemento in prima posizione (dato che è stato scelto X_1) è uguale a 1. Si ottiene quindi: [1,2,3], [1,3,2]

5. Problemi svolti

Per verificare l'effettivo funzionamento del programma ho svolto 4 problemi di soddisfacimento dei vincoli, 3 dei quali presenti su http://www.csplib.org/.

- *N-Queens (prob054)*: in una scacchiera NxN posizionare N regine in modo che non possano mangiarsi a vicenda



- *The n-Fractions Puzzle (prob041)*: assegnare a 9 variabili un valore da 1 a 9 in modo da risolvere la seguente equazione:

$$\frac{A}{10B+C}+\frac{D}{10E+F}+\frac{G}{10H+I}=1$$

```
trovata la soluzione


[2, 3, 5, 8] [1, 4, 6, 7]

Somma =  18
Somma quadrati =  102
Eseguito in: 6.158924325
```

- *Number Partitionating (prob049)*: partizionare N numeri in 2 insiemi A e B in modo che:
  i. A e B abbiano lo stesso numero di elementi
  ii. La somma dei numeri in A deve essere uguale alla somma dei numeri in B
  iii. La somma dei quadrati dei numeri in A deve essere uguale alla somma dei quadrati dei numeri in B

  La scelta di N non può essere casuale ma deve valere N≥8 e N multiplo di 4

```
trovata la soluzione


[2, 4, 5, 6, 10, 12] [1, 3, 7, 8, 9, 11]

Somma =  39
Somma quadrati =  325
Eseguito in: 3.428781702
```

- *Map Coloring*: colorare le singole regioni di una mappa in modo che non risultino stati adiacenti col medesimo colore

```
trovata la soluzione


WA red
NT green
SA blue
Q red
NSW green
V red
T red
Eseguito in: 0.10011696000000003
```

6. Link

- *Binarization*: http://ktiml.mff.cuni.cz/~bartak/constraints/binary.html
- *The n-Fractions Puzzle (prob041):* http://www.csplib.org/Problems/prob041/
- *Number Partitionating (prob049)*: http://www.csplib.org/Problems/prob049/
- *N-Queens (prob054)*: http://www.csplib.org/Problems/prob054/