Intelligenza Artificiale: Propagazione di vincoli tramite backtracking

Andrea Rossini

2 settembre 2022

1 Introduzione

In questo progetto relativo all'esame di Intelligenza Artificiale si vuole implementare ed applicare l'algoritmo di backtracking, usando sia forward checking che MAC, a 3 problemi di soddisfacimento di vincoli scelti a piacere da CSPLib (https://www.csplib.org/).
Per la stesura del codice, ho scelto di usare il seguente approccio: per prima cosa ho cercato di capire come funzionassero le 2 strategie di risoluzione per ciascun problema, per poi successivamente replicare queste nel codice.
La scelta dei 3 CSP è ricaduta su:

1. **Problema delle N regine**: come già spiegato a lezione, si tratta di uno scenario nel quale dobbiamo piazzare, su una scacchiera di dimensioni N x N, N regine in modo tale che nessuna sia in grado di attaccare l'altra (https://www.csplib.org/Problems/prob054).

2. **Killer Sudoku**: trattasi di una variante del classico sudoku, nella quale troviamo dei cage, ossia gruppi di celle adiacenti che si distinguono grazie al colore assegnatogli. All'interno di una delle celle che compone un cage troviamo un piccolo numero in alto a sinistra: esso indica a quanto deve ammontare la somma dei valori che compongono il cage stesso (https://www.csplib.org/Problems/prob057).

3. **Problema delle N regine (versione BLOCKED)**: variante del problema delle n regine nella quale, oltre ai vincoli precedentemente citati, il problema prende in input anche una lista di celle "vietate", dove chiaramente le regine non possono essere posizionate (https://www.csplib.org/Problems/prob080).

1.1 Documentazione del codice

Il programma sviluppato per eseguire i test richiesti contiene al suo interno i seguenti file:

1. Il file "NQueensProblem.py", che contiene il codice sviluppato per risolvere il problema delle n regine. Il metodo *checkForwardAttempt()* si occupa di propagare i vincoli, ogni qualvolta una regina viene posizionata sulla scacchiera, usando il meccanismo del forward checking, andando a marcare le caselle interessate dai vincoli con un "placeholder" che indica riga e colonna della cella dalla quale il vincolo è stato propagato. Il metodo *checkAttemptWithMAC()* tenta la mossa corrente usando lo stesso meccanismo di AC-3 usato da MAC: ogni volta che piazza una regina, controlla le celle "illegali" nella successiva colonna di destra e, se nessuna regina può esservi piazzata in modo sicuro, torna indietro alla precedente regina e prova a spostarla nella successiva riga disponibile (dall'alto verso il basso).

2. Il file "KillerSudokuProblemFC.py", contenente il codice che utilizza Forward Checking per il problema del Killer Sudoku. Il metodo *defineCagesFromJson()* si occupa di leggere dai file JSON quali celle compongono i cages e a quanto ammonta la somma dei valori al loro interno, per poi crearne delle tuple. Troviamo poi una serie di metodi per controllare che in ogni quadrato, riga o colonna non ci siano valori duplicati; *inferenceOnPossibleAssignmentsWithFC()* propaga i vincoli usando il forward checking, e allo stesso tempo controlla che la somma dei valori all'interno di un cage non superi quella prestabilita dai vincoli del problema.

3. Il file "KillerSudokuProblemWithPulp.py", che contiene la classe che si occuperà di risolvere il problema del Killer Sudoku usando il MAC come strategia di propagazione dei vincoli; viene usata la libreria **PuLP** che permette un'ottimizzazione lineare del codice; in questo modo, il MAC è visto come un problema di programmazione lineare (Ho scelto di usare questa versione dal momento che, non avendo trovato l'esatta strategia di risoluzione per il MAC, ho deciso di affidarmi a tale libreria, la quale consente di definire il problema, la funzione obiettivo e i vincoli per poi risolverlo in un tempo lineare) .

4. Il file "BlockedNQueensProblem.py", che contiene il codice per la versione blocked del problema delle n regine. Sostanzialmente troviamo le stesse linee di codice del modulo *NqueensProblem.py*, con alcune aggiunte: infatti la scacchiera verrà inizializzata con alcune celle marcate con 'F' (Forbidden) nelle quali non potrà essere posizionata alcuna regina, e nei metodi per la propagazione dei vincoli già citati verranno aggiunti dei controlli per evitare di piazzare una regina in una cella vietata; troviamo in fondo il metodo *initializeChessboardsForTests()*, che si occupa di inizializzare le scacchiere 4x4, 5x5 e 6x6 con la prima 'Q' in [0,0] e alcune celle marcate con 'F'.

5. Il file "UtilityForAlgorithms.py", contenente l'omonima classe, avente al suo interno un elenco di metodi statici che verranno utilizzati dai vari algoritmi relativi ai 3 problemi scelti.

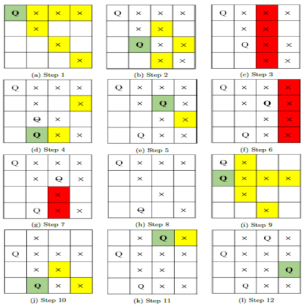

Figura 1: Passaggi per l'esecuzione del Forward Checking nella risoluzione del problema delle n regine: si nota come ogni qualvolta una regina viene posizionata, vengono propagati i suoi vincoli in **TUTTE** le celle interessate.

6. Altri file di formato .json e .txt, i quali verranno utilizzati dalle 2 versioni del killer sudoku per inizializzare la griglia con i cages e la relativa somma totale.

2 Svolgimento

2.1 Architettura calcolatore e ambiente di sviluppo

Gli esperimenti vengono condotti su un Acer Swift SF314-52 con processore Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.71 GHz, 8GB di RAM, con sistema operativo Windows 10 Home a 64 bit, processore basato su x64. Tutto il codice è stato scritto utilizzando l'IDE Pycharm Professional 2022.3, con versione di Python 3.10.

2.2 Considerazioni preliminari sul funzionamento dei metodi

2.2.1 N Queens (classico e blocked)

Nel problema delle n regine (così come nella versione blocked), il forward checking è implementato nel metodo *checkForwardAttempt()*: inserisce la prima regina (**Q**) nella cella [0,0] e marca con con 2 interi (corrispondenti agli indici di riga e colonna della cella nella quale abbiamo posizionato la regina) le celle nelle colonne successive nelle quali, per quel particolare assegnamento, sicuramente non potrà essere inserita alcuna regina (avendo scelto di scandire le colonne, è inutile controllare le celle appartenenti alla stessa colonna). Poi contiamo il numero di celle "legali" per ogni colonna e scegliamo, per proseguire l'esecuzione, quella che ne contiene meno; a questo punto si posiziona la regina nella prima cella disponibile nella colonna scelta e si prosegue con la propagazione dei vincoli, avendo cura di "tornare indietro" (cioè tornare alla colonna dove abbiamo messo l'ultima **Q** e tentare con la riga successiva, se nessuna riga è disponibile torniamo alla regina ancora precedente e così via, finchè non troviamo una cella disponibile) nel caso un assegnamento portasse ad avere una colonna fatta solo di celle illegali (Figura 1).
Il MAC è implementato in *checkAttemptWithMAC()*: anche qui si assegna la prima **Q** in [0,0] e si crea la consistenza d'arco andando a segnare come illegali le celle nella stessa riga o stessa diagonale della colonna successiva. A questo punto si passa alla colonna successiva e piazziamo la **Q** nella prima riga disponibile, sempre se un tale assegnamento è consistente coi vincoli preesistenti; se non lo è, si torna indietro e si prova, come prima, la riga successiva e se nessuna riga fosse disponibile, torniamo indietro di una colonna e facciamo scorrere la rispettiva **Q** giù di una riga (Figura 2).

2.2.2 Killer sudoku

Nel problema del Killer sudoku, le griglie iniziali vuote con la sola indicazione dei valori dei cages sono state prese da CSPLib e da https://www.dailykillersudoku.com/search?dt=2020-02-15. I risultati riportati si riferiscono alla griglia di CSPLib, per cambiarla è sufficiente cambiare il file di input da *CSPLibTestCagesDataSource* (sia .json che .txt) a *OtherTestCagesDataSource* all'interno rispettivamente di KillerSudokuProblemFC.py (riga 153) e KillerSudokuProblemWithPulp.py (riga 99). Per quanto riguarda il Forward Checking, iniziamo dalla cella [0,0], proviamo a posizionare un valore e passiamo alla cella della colonna successiva (ci muoviamo in orizzontale, e una volta arrivati all'ultima colonna passiamo alla riga successiva): per le celle che fanno parte di uno stesso cage, controlliamo che la somma dei valori non sia maggiore del valore prefissato (contenuto nel file JSON corrispondente) e, se così fosse, torniamo indietro grazie al backtracking finchè non troviamo i valori giusti. Per l'esecuzione della versione con MAC non sono riuscito a trovare gli esatti passi eseguiti dall'algoritmo, dunque ho utilizzato la libreria pulp, la quale consente di risolvere il problema fornitogli tramutandolo in uno di programmazione lineare, permettendo un'ottimizzazione lineare del codice: è sufficiente definire la funzione obiettivo (minimizzare nel nostro caso, è il parametro di default di *pulp.LpProblem()*) e i vincoli del problema, e la libreria (tramite il metodo *solve()*) si occuperà di trovare la soluzione ottima.

Figura 2: Passaggi per l'esecuzione del MAC nella risoluzione del problema delle n regine: si nota come, in questo caso, quando si posiziona una regina, vengono propagati i vincoli solo nella colonna adiacente a destra.

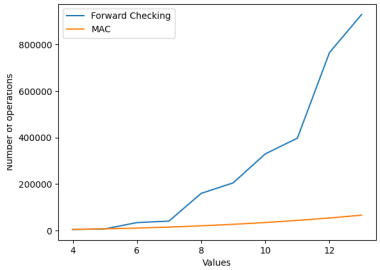

Figura 3: Grafico che mostra il numero totale di operazioni per il problema delle n regine (incluso il posizionare una 'Q' e i relativi vincoli, il rimuoverli, ma anche lo scandire una particolare riga o colonna) eseguendo il codice 50 volte per ogni dimensione: si nota che il FC esegue un numero di operazioni che cresce esponenzialmente con l'aumentare delle dimensioni della scacchiera (in totale ben 2874900), mentre il MAC è molto meno dispendioso, con una crescita molto meno accentuata (solo 284250 operazioni, un numero enormemente più piccolo).

2.3 Esperimenti

Per quanto riguarda il problema delle n regine (in entrambe le versioni), i test consistono nell'eseguire sia il Forward Checking che il MAC 50 volte per ogni dimensione della scacchiera (da 4x4 a 13x13), per poi contare il numero totale di operazioni effettuate (quante volte inserisco una 'Q' e propago i suoi vincoli, ma anche quante volte li rimuovo) e il tempo di esecuzione (per ogni ciclo di 50 iterazioni calcolo il tempo medio).
Per il killer sudoku, poichè per definizione del problema la griglia è fissata a dimensione 9x9, non avremo dimensioni crescenti di quest'ultima: per i test, verranno eseguite entrambe le modalità 50 volte.

- **N regine**: Si nota, dal grafico in figura 3, che utilizzando come strategia di inferenza il MAC si ha un grande risparmio in quanto a numero di operazioni effettuate (circa 10 volte in meno rispetto al FC). Per quanto riguarda i tempi di esecuzione, vediamo dal grafico in figura 4 come anche qui il MAC si appresti a fornire la soluzione in tempi minori rispetto al FC, soprattutto per dimensioni della scacchiera da 7x7 in su
- **N regine Blocked**: Per questa versione sono state usate solo scacchiere di dimensione 4x4, 5x5 e 6x6 (solamente perchè avrei dovuto definire esplicitamente a mano le scacchiere, e con dimensioni sempre più grandi diventa piuttosto lungo, senza contare che il codice è praticamente uguale a quello delle n regine classico). Anche qui, dai grafici in figura 6 e 5, si evince che il MAC funziona decisamente meglio sia in termini di tempo

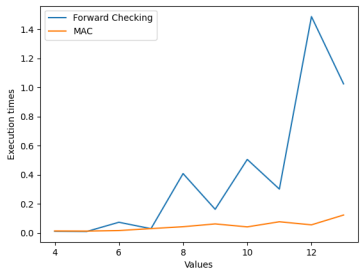

Figura 4: Grafico che mostra i tempi di esecuzione per il problema delle n regine (espressi in secondi) in funzione delle dimensioni della scacchiera: si nota come per dimensioni "piccole" (da 4x4 a 7x7) i tempi siano simili, poi il MAC si dimostra più rapido nell'esecuzione (in totale il Forward Checking impiega 5.733900199968048 secondi contro gli 0.7573920000104408 secondi del MAC: un bel risparmio!).

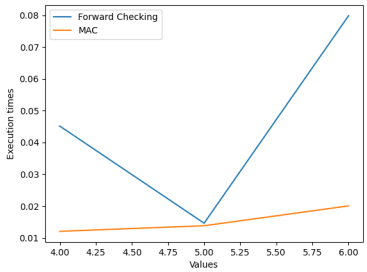

Figura 5: Grafico che mostra i tempi di esecuzione per il problema delle n regine BLOCKED (espressi in secondi) in funzione delle dimensioni della scacchiera: si nota come, eccetto per la scacchiera di dimensione 5x5 per la quale i tempi sono simili, il MAC permetta un risparmio di tempo effettivo rispetto al Forward Checking.

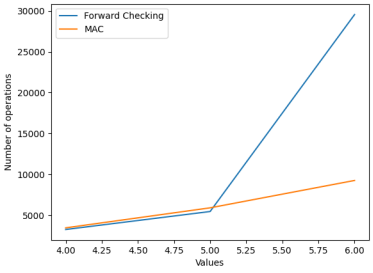

Figura 6: Grafico che mostra il numero di operazioni effettuate da MAC e Forward Checking per risolvere il problema delle n regine blocked: vediamo come risulti più efficace il MAC, specialmente da scacchiere di dimensioni 5x5 in su.

(0.04348010001012881 secondi contro 0.15022860000863147 secondi del FC) che di operazioni (18600 contro le 38250 del FC).

- **Killer Sudoku**: Si eseguono entrambi i moduli "KillerSudokuProblemFC.py" e "KillerSudokuProblemWithPulp.py; per quanto riguarda la versione con Forward Checking, vengono eseguite 14550 operazioni in 387.936742400012 secondi, mentre tramite l'utilizzo di PuLP si ha un netto miglioramento (3325 operazioni in 15.120743299990863 secondi)

3 Conclusioni

In conclusione, possiamo affermare che la strategia di propagazione di vincoli tramite MAC risulta più soddisfacente rispetto al Forward Checking, essendo più efficace ed efficiente (sia per quanto riguarda il numero di operazioni effettuate per trovare la soluzione, che per il tempo totale che impiega nel trovarne una). Infatti, in ognuno dei 6 casi di test, la strategia che utilizza MAC permette un ingente miglioramento delle prestazioni, come ben visibile dalla tabella 1.
Questo è un aspetto molto importante per problemi come quello delle N regine, che contengono N x N variabili, e più alto è il valore assunto da N, più complesso diventa il problema.

4 Fonti

1) https://www.researchgate.net/publication/323067430_Exhaustive_study_of_essential_constraint_satisfactior
problem_techniques_based_on_N-Queens_problem#pf3 per il funzionamento degli algoritmi sul problema delle N regine

CONFRONTO PRESTAZIONI FORWARD CHECKING vs MAC											
N REGINE				**N REGINE BLOCKED**				**KILLER SUDOKU**			
Operazioni		**Tempo**		**Operazioni**		**Tempo**		**Operazioni**		**Tempo**	
FC	**MAC**	**FC**	**MAC**	**FC**	**MAC**	**FC**	**MAC**	**FC**	**MAC**	**FC**	**MAC**
2874900	284250	5.7339s	0.757s	38250	18600	0.1502s	0.0435s	14550	3325	387.9367s	15.1207s
1/10		**1/8**		**1/2**		**1/4**		**1/4**		**1/25**	

Tabella 1: Tabella che mostra il confronto finale delle prestazioni tra FC e MAC. L'ultima riga rappresenta il "fattore di miglioramento" del MAC (ossia di quanto riduce il tempo o il numero di operazioni rispetto al FC)

2) Stuart Russell e Peter Norvig, *Artificial Intelligence A Modern Approach, Fourth Edition*, Pearson 2021

3) https://coin-or.github.io/pulp/ Per la documentazione sulla libreria pulp

4) https://www.freecodecamp.org/news/python-parse-json-how-to-read-a-json-file/ per la documentazione sull'utilizzo dei file JSON in Python