Attività Progettuale in Fondamenti di Intelligenza Artificiale M

Luca Marongiu

Ottobre 2025

Indice

1 Introduzione **2**

2 Stato dell'arte e fondamenti teorici **2**
2.1 Difficoltà e scelte non deterministiche 2
2.2 Numero minimo di indizi e istanze "minimali" 2

3 Dataset utilizzato **3**
3.1 Concetto di T&E-depth e classificazione logica 3
3.2 Sudoku utilizzati nei test 4

4 Solver utilizzati per il confronto **4**
4.1 Implementazione personale DLX (Algorithm X) 4
4.2 Modello CSP in MiniZinc + Gecode / OR-Tools 5
4.3 Solver specializzati esterni 6

5 Risultati sperimentali **6**
5.1 Setup sperimentale 6
5.2 Tempi medi di risoluzione 7
5.3 Analisi comparativa 7

6 Conclusioni **8**

1 Introduzione

Il progetto ha come obiettivo la comparazione di diversi approcci alla risoluzione automatica di Sudoku, considerando sia metodi basati su propagazione dei vincoli (CSP) sia algoritmi combinatori di tipo Exact Cover. Sono stati confrontati solver sviluppati in Python, MiniZinc e due solver in C/C++, per valutare le differenze in termini di velocità, scalabilità e comportamento rispetto alla complessità logica dei puzzle.

2 Stato dell'arte e fondamenti teorici

Il Sudoku è da tempo oggetto di studio nella comunità dell'intelligenza artificiale poiché rappresenta un eccellente caso di *Constraint Satisfaction Problem* (CSP). In letteratura viene spesso usato come problema di riferimento per analizzare le proprietà di consistenza, le euristiche di ricerca e le tecniche di propagazione dei vincoli. In questa sezione vengono sintetizzati i principali risultati teorici e sperimentali che guidano le scelte implementative di questo progetto.

2.1 Difficoltà e scelte non deterministiche

Un errore comune nella classificazione della difficoltà di un Sudoku è considerare il solo numero di celle vuote come indice della complessità. In realtà, come mostrato da Pelánek [5], la difficoltà emerge dalla **struttura logica interna del puzzle**: non da quanto manca, ma da **quanto ambigue sono le scelte** che il risolutore deve affrontare. La difficoltà massima infatti si manifesta quando il puzzle contiene il numero ottimale di **scelte apparentemente non deterministiche**: situazioni in cui più valori sembrano possibili per una variabile, ma in realtà solo uno sopravvive applicando correttamente le tecniche di consistenza.

Pelánek dimostra che i puzzle più complessi per i solver umani sono proprio quelli in cui molte scelte appaiono non deterministiche se analizzate superficialmente, ma diventano deterministiche solo applicando livelli superiori di consistenza.

Dal punto di vista computazionale, ciò significa che la difficoltà del Sudoku non cresce monotonicamente con il numero di celle vuote: esistono puzzle "mediamente pieni" che risultano i più impegnativi, perché massimizzano il numero di rami alternativi che devono essere esplorati prima che la propagazione dei vincoli li elimini.

2.2 Numero minimo di indizi e istanze "minimali"

Un risultato fondamentale, ormai classico, è quello di McGuire et al. [4], che hanno dimostrato con una ricerca esaustiva che non esistono Sudoku validi con 16 indizi aventi una sola soluzione. Il lavoro, noto come *"There is no 16-Clue Sudoku"*, ha impiegato metodi di enumerazione basati su *unavoidable sets* e *hitting sets* per mostrare che ogni Sudoku

con unicità di soluzione richiede almeno 17 indizi. Questo valore rappresenta dunque il limite inferiore teorico per l'unicità della soluzione in una griglia standard 9×9.

In sintesi, gli autori hanno considerato tutte le griglie di Sudoku completate (distinte fino alle simmetrie di riga, colonna, cifra e trasposizione) e per ciascuna hanno individuato insiemi di celle chiamati *unavoidable sets*, ovvero sottoinsiemi che devono necessariamente contenere almeno un indizio affinché la soluzione resti unica. La ricerca di un puzzle con 16 indizi si riduce quindi a un problema combinatorio: trovare un insieme di 16 celle che *interseca* tutti gli unavoidable sets, ossia un cosiddetto *hitting set*. Se un tale insieme esistesse, potrebbe fornire un Sudoku da 16 indizi con soluzione unica.

Attraverso un algoritmo ottimizzato di enumerazione e verifica, eseguito su diverse piattaforme di calcolo parallelo, gli autori hanno esaminato tutte le 5,472,730,538 classi di griglie distinte. Nessun hitting set valido di 16 celle è stato trovato, dimostrando in modo esaustivo che non esiste alcun Sudoku con 16 indizi e soluzione unica. Pertanto, 17 rappresenta il numero minimo possibile di indizi per garantire l'unicità della soluzione.

3 Dataset utilizzato

Per valutare le prestazioni dei diversi solver in relazione alla complessità intrinseca del problema, è stato utilizzato il dataset *Sudoku-classif*, sviluppato da Denis Berthier e reso disponibile su GitHub [1]. A differenza di altri dataset basati su misure empiriche (numero di indizi, tempi di soluzione umani, ecc.), *Sudoku-classif* adotta una classificazione fondata su criteri teorici di **complessità logico-inferenziale**, derivati dalla teoria dei *Constraint Satisfaction Problems (CSP)*.

3.1 Concetto di T&E-depth e classificazione logica

La classificazione proposta da Berthier si basa sul concetto di **Try & Error depth (T&E-depth)**, formalizzato nei suoi lavori sulla *Pattern-Based Constraint Satisfaction* e su *CSP-Rules*. Il parametro T&E-depth rappresenta il numero minimo di livelli di ipotesi necessari affinché la propagazione logica dei vincoli renda il puzzle completamente deterministico. In termini CSP, misura la **profondità inferenziale** richiesta per raggiungere la consistenza globale.

In particolare:

- un puzzle in **T&E(0)** è risolvibile tramite sola *arc-consistency*, ossia mediante eliminazioni dirette e determinazione dei *single*;
- un puzzle in **T&E(1)** richiede un unico livello di scelta seguita da propagazione, equivalente all'applicazione di un semplice schema di *look-ahead*;
- un puzzle in **T&E(2)** implica due livelli di inferenza annidata, con analisi di dipendenze tra variabili o catene logiche complesse;

- un puzzle in **T&E(3)** o superiore necessita di inferenze più profonde, come le *OR-k relations* (relazioni disgiuntive tra k candidati) o schemi di *Full Look-Ahead (FLA)* introdotti da Berthier per gestire pattern impossibili.

La profondità T&E(n) non misura quindi la difficoltà computazionale per un algoritmo, ma il numero di passaggi inferenziali che un risolutore puramente logico deve compiere prima che la propagazione renda il sistema deterministico. Per questo motivo, essa costituisce una metrica oggettiva di **difficoltà logico-inferenziale**, strettamente connessa ai concetti di consistenza e propagazione nei CSP.

3.2 Sudoku utilizzati nei test

Nel progetto sono stati selezionati puzzle appartenenti a tre collezioni rappresentative del dataset *Sudoku-classif*, corrispondenti a diversi livelli di profondità T&E:

- **Royle-49158-17c** — puzzle con 17 indizi, tutti in T&E(0)–T&E(1), utilizzati come casi **facili**;
- **eleven-26370-TE2** — puzzle in T&E(2),utilizzati come casi **intermedi**;
- **mith-158276-TE3** — puzzle in T&E(3), utilizzati come casi **difficili**.

Questa scelta consente di confrontare le prestazioni dei diversi solver su insiemi di Sudoku classificati secondo una metrica teoricamente fondata e indipendente da considerazioni empiriche. Inoltre, permette di analizzare la relazione tra difficoltà logico-inferenziale (T&E-depth) e difficoltà computazionale effettiva, mostrando come i puzzle logicamente più profondi possano risultare più vincolati e quindi più semplici da risolvere per i solver automatici.

4 Solver utilizzati per il confronto

4.1 Implementazione personale DLX (Algorithm X)

Il primo solver implementato si basa sull'algoritmo **Dancing Links** (**DLX**) proposto da Donald Knuth [3]. DLX è un'implementazione particolarmente efficiente dell'algoritmo `Algorithm X`, un metodo di backtracking non deterministico per la risoluzione di problemi di *Exact Cover*.

Rappresentazione del problema. Il Sudoku può essere trasformato in un problema di Exact Cover: ogni possibile assegnamento (r, c, n) (numero n nella cella (r, c)) deve soddisfare esattamente quattro vincoli:

1. la cella (r, c) deve contenere un solo numero;
2. il numero n deve comparire una volta sola nella riga r;

3. il numero n deve comparire una volta sola nella colonna c;
4. il numero n deve comparire una volta sola nel sotto-blocco di appartenenza.

Questi vincoli vengono enumerati e raccolti in una struttura X, mentre per ogni possibile assegnamento (r, c, n) si costruisce un insieme $Y[(r, c, n)]$ che indica quali vincoli quell'assegnamento soddisfa.

Struttura dati adottata. L'implementazione sviluppata simula l'Algorithm X utilizzando le strutture dati native di Python, senza implementare la rete di puntatori doppiamente collegati della versione originale di Knuth:

- X è un dizionario che associa a ciascun vincolo (colonna) l'insieme delle righe che lo soddisfano;
- Y è un dizionario che associa a ogni assegnamento (r, c, n) i vincoli che esso copre.

Algoritmo di ricerca. Il cuore dell'algoritmo è la procedura ricorsiva `solve_dlx`, che segue esattamente i passi descritti da Knuth:

1. se non rimangono vincoli in X, è stata trovata una soluzione;
2. altrimenti si seleziona il vincolo con il minor numero di possibilità (euristica di riduzione della ramificazione);
3. per ciascun assegnamento che soddisfa quel vincolo, si aggiunge l'assegnamento alla soluzione parziale e si **coprono** i vincoli coinvolti;
4. si prosegue ricorsivamente; al termine, si esegue la procedura inversa (**scopri**) per ripristinare lo stato e continuare l'esplorazione.

Le funzioni `copri` e `scopri` simulano le operazioni di cover/uncover dell'Algorithm X: rimuovono e ripristinano elementi da dizionari e insiemi, permettendo di eseguire efficacemente le operazioni di backtracking richieste dall'algoritmo. L'implementazione completa si trova nel file `versione_dancing_links.py`.

4.2 Modello CSP in MiniZinc + Gecode / OR-Tools

In parallelo, il Sudoku è stato modellato in MiniZinc (variabili $X_{i,j} \in \{1, \ldots, 9\}$ con vincoli `all_different` su righe, colonne e box 3×3). Lo *stesso* modello è stato risolto con due backend distinti:

- **Gecode**: solver CP nativo con propagatori efficienti per `all_different`;
- **OR-Tools**: solver CP-SAT (Google) fornisce propagazione aggressiva e euristiche consolidate a livello industriale.

Questo setup consente un confronto pari: stesso modello ad alto livello, differiscono solo i meccanismi di propagazione/ricerca del backend.

4.3 Solver specializzati esterni

Per avere un riferimento con implementazioni altamente ottimizzate, ho incluso due solver stand-alone molto noti:

- **Fast Solver 9r2** (*DLX* in C): risolutore basato su *Exact Cover*/DLX [6];
- **FSSS2** (bitboard + SIMD): redesign ad alte prestazioni (C++) che sfrutta rappresentazioni a bitboard e istruzioni SIMD per ridurre drasticamente il costo del backtracking [2].

5 Risultati sperimentali

In questa sezione vengono riportati e analizzati i risultati del benchmark condotto sui sei risolutori considerati nel progetto:

- **DLX** – implementazione Python dell'Algorithm X basato su *Exact Cover*;
- **Fast9r2** – solver C ottimizzato basato su Dancing Links ([6]);
- **FSSS2** – solver ad alte prestazioni basato su bitboard e SIMD ([2]);
- **MiniZinc + Gecode** – modello CSP risolto tramite backend Gecode;
- **MiniZinc + OR-Tools** – stesso modello CSP eseguito tramite backend OR-Tools (CP-SAT);
- **OR-Tools nativo** – implementazione diretta in Python del motore CP-SAT.

5.1 Setup sperimentale

Ogni solver ha risolto i Sudoku contenuti in tre file di test, corrispondenti a livelli di difficoltà logica crescenti secondo la classificazione T&E-depth [1]:

- **Facile:** puzzle in T&E(0)–T&E(1) (collezione Royle-17c);
- **Medio:** puzzle in T&E(2) (collezione eleven-26370-TE2);
- **Difficile:** puzzle in T&E(3) (collezione mith-TE3).

Per ciascun puzzle, ogni solver è stato eseguito per 100 run consecutivi; è stata calcolata la media dei tempi di esecuzione (in millisecondi). I test sono stati condotti nello stesso ambiente hardware e software.

5.2 Tempi medi di risoluzione

Solver	Facile (ms)	Medio (ms)	Difficile (ms)
DLX	7.04	92.87	58.99
Fast9r2	15.44	13.27	11.34
FSSS2	13.52	11.54	10.08
Gecode (MiniZinc)	1053.77	977.48	1085.02
OR-Tools (MiniZinc)	1092.21	1214.49	3210.00
OR-Tools (nativo)	9.50	24.85	19.88

Tabella 1: Tempi medi di risoluzione per ciascun solver e livello di difficoltà logica (T&E-depth).

5.3 Analisi comparativa

Comportamento rispetto alla difficoltà T&E-depth. Un risultato particolarmente interessante emerso dai test è che i Sudoku classificati come "difficili" in termini di **T&E-depth** non sempre corrispondono a tempi di risoluzione più lunghi per i solver computazionali. Nei solver basati su **enumerazione combinatoria ottimizzata** (Fast9r2 e FSSS2), i puzzle in T&E(3) risultano addirittura **più rapidi da risolvere** rispetto a quelli in T&E(1)–T&E(2), poiché la maggiore rigidità dei vincoli riduce lo spazio effettivo delle configurazioni compatibili. Al contrario, l'implementazione **DLX in Python**, pur adottando lo stesso schema di *Exact Cover* (Algorithm X), mostra tempi più elevati sui puzzle più complessi a causa dell'overhead interpretativo e della gestione non ottimizzata delle strutture dati.

Questo comportamento è coerente con quanto osservato in letteratura: la classificazione T&E-depth misura la *profondità logica* necessaria per dedurre la soluzione tramite tecniche di consistenza, e non la difficoltà computazionale effettiva. Un puzzle in T&E(3) può richiedere inferenze più profonde per un risolutore umano, ma risultare più deterministico per un solver automatico, che esplora sistematicamente uno spazio di ricerca già fortemente vincolato.

Solver DLX, Fast9r2, FSSS2. Il solver **DLX** in Python mostra tempi contenuti sui puzzle più semplici, ma cresce sensibilmente su T&E(2) e T&E(3), a causa della natura interpretata del linguaggio e della gestione non ottimizzata delle strutture dati. Il solver **Fast9r2**, scritto in C e basato su una versione altamente ottimizzata dell'algoritmo di Exact Cover (DLX), mantiene prestazioni estremamente elevate e stabili in tutti i livelli di difficoltà logica. Infine, il solver **FSSS2**, pur non adottando DLX, utilizza una strategia di **backtracking bit-parallel** su rappresentazioni a bitboard con istruzioni SIMD, che consente di esplorare lo spazio delle configurazioni con efficienza eccezionale. In entrambi i casi, le istanze più complesse (T&E(3)) risultano spesso più rapide da risolvere, poiché

la maggiore interconnessione dei vincoli riduce lo spazio effettivo di ricerca combinatoria pur aumentando la profondità logica richiesta per dedurne la soluzione.

Solver CSP (Gecode e OR-Tools via MiniZinc). I solver **MiniZinc** presentano tempi nettamente superiori, nell'ordine dei secondi, a causa dell'overhead introdotto dal framework *FlatZinc* e della comunicazione con i backend sottostanti. Tuttavia, questa architettura garantisce **portabilità e indipendenza dal solver**, rendendo MiniZinc uno strumento ideale per il confronto teorico di modelli CSP e per la prototipazione rapida di vincoli complessi.

Solver CP-SAT nativo. L'implementazione diretta di **OR-Tools nativo** mostra prestazioni eccellenti, nettamente superiori a quelle dell'implementazione Python di DLX, pur operando su un modello di natura completamente diversa. Grazie alla propagazione efficiente dei vincoli e all'integrazione con il motore SAT sottostante, OR-Tools riduce drasticamente lo spazio di ricerca senza dover enumerare esplicitamente tutte le combinazioni possibili, come avviene invece nell'Algorithm X. Eliminando il passaggio di traduzione in FlatZinc, il tempo medio di risoluzione scende di due ordini di grandezza rispetto alla versione MiniZinc, e risulta circa tre volte inferiore a quello del solver DLX in Python nei casi più complessi. Questo conferma l'efficacia del paradigma CP-SAT, che combina la propagazione booleana con la ricerca a vincoli per ottenere prestazioni elevate anche in problemi fortemente strutturati.

6 Conclusioni

L'analisi sperimentale ha evidenziato differenze significative tra i principali paradigmi di risoluzione del Sudoku, mostrando come la **difficoltà logico-cognitiva** e la **difficoltà computazionale** rappresentino due dimensioni distinte dello stesso problema.

I solver basati su **Exact Cover** (DLX, Fast9r2) si sono rivelati estremamente efficienti, grazie alla rappresentazione compatta dei vincoli e all'esplorazione sistematica dello spazio delle soluzioni. **FSSS2** — grazie all'uso di strutture bitboard e istruzioni SIMD — ha ottenuto le prestazioni migliori in assoluto, con tempi quasi costanti anche per puzzle a elevata profondità logica (T&E(3)).

I solver **CSP** basati su MiniZinc (Gecode e OR-Tools via FlatZinc) hanno invece mostrato tempi sensibilmente superiori, a causa dell'overhead introdotto dal framework e della traduzione del modello. Tale approccio, tuttavia, garantisce **portabilità e astrazione**, rendendo MiniZinc ideale per la modellazione solver-indipendente e la validazione teorica dei modelli di vincolo.

L'implementazione **OR-Tools nativa**, costruita direttamente sull'API CP-SAT, ha raggiunto prestazioni comparabili ai solver compilati, eliminando il livello intermedio di MiniZinc e sfruttando appieno le ottimizzazioni del motore CP-SAT

Dal confronto con la classificazione T&E-depth emerge un risultato di rilievo: la difficoltà logico-inferenziale, che misura la profondità delle deduzioni necessarie per una risoluzione puramente logica, non coincide con la difficoltà computazionale misurata in termini di tempo di esecuzione. Le istanze più complesse dal punto di vista logico (T&E(3)) non sono necessariamente le più onerose per un solver automatico; al contrario, possono risultare più rapide grazie alla maggiore rigidità dei vincoli, che riduce lo spazio di ricerca effettivo.

In sintesi:

- **FSSS2** rappresenta l'efficienza pura, grazie alle ottimizzazioni a basso livello;
- **OR-Tools nativo** offre il miglior equilibrio tra generalità e prestazioni;
- **MiniZinc** rimane lo strumento di riferimento per la modellazione solver-indipendente e l'analisi comparativa;
- i risultati mostrano una **correlazione inversa** tra difficoltà logico-inferenziale (T&E-depth) e difficoltà computazionale effettiva.

In conclusione, il progetto mostra come l'integrazione di diversi paradigmi — Exact Cover, CSP e CP-SAT — permetta di esplorare in modo complementare le due nature del Sudoku: problema logico e cognitivo da un lato, combinatorio e computazionale dall'altro. La distinzione tra questi due livelli di difficoltà chiarisce l'apparente paradosso dei risultati e offre una prospettiva più ampia sulla nozione di **complessità** nei problemi di soddisfacimento di vincoli.

Riferimenti bibliografici

[1] Denis Berthier. Sudoku-classif: Classification of sudoku puzzles by logical complexity. https://github.com/denis-berthier/sudoku-classif.

[2] Mladen Dobrichev. Fsss2: Fast simple sudoku solver 2. https://github.com/dobrichev/fsss2/tree/master.

[3] Donald E Knuth. Dancing links. URL: https://arxiv.org/abs/cs/0011047.

[4] Gary McGuire, Bastian Tugemann, and Gilles Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem. URL: https://arxiv.org/abs/1201.0749.

[5] Radek Pelánek. Difficulty rating of sudoku puzzles: An overview and evaluation. URL: https://arxiv.org/abs/1403.7373.

[6] Ignacio Zelaya. Sudoku fast solver v.9r2. https://github.com/attractivechaos/plb/blob/master/sudoku/incoming/fast_solv_9r2.c.