Andrea Grendene 1184985

Sudoku Solver

Progetto per il corso di Linguaggi per il Global Computing

Introduzione

Il sudoku è un gioco di logica, inventato dal matematico svizzero Eulero di Basilea. La versione moderna venne pubblicata per la prima volta nel 1979 dall'architetto statunitense Howard Garns, nel 1984 venne diffuso in Giappone dalla casa editrice Nikoli e poi nel 2005 divenne noto a livello internazionale[1].
Per la risoluzione automatica del gioco, la tecnica più idonea è la programmazione con vincoli, in cui ogni cella vuota viene riempita con i 9 possibili valori. Essi poi verranno eliminati, ad esempio togliendo i numeri già fissati nelle righe, nelle colonne e nei box rispettivi. Nel caso dei sudoku più semplici, questa tecnica risulta essere sufficiente per trovare una soluzione. Aumentando un po' la difficoltà, però, non basta, perché rimangono delle celle con più valori possibili. Un'idea potrebbe essere implementare un algoritmo di forza bruta, in cui si prova a fissare tali numeri; il numero di combinazioni valide però è molto elevato, e di conseguenza il tempo di esecuzione sarà accettabile soltanto con poche celle vuote. Esistono però numerose altre tecniche con cui ridurre il numero di valori possibili, come spiegato nel sito "http://www.sudokuwiki.org/sudoku.htm". Da questo indirizzo sono stati inoltre ricavati i termini usati in questa relazione e nel progetto.
Per la gestione del sudoku, l'algoritmo di forza bruta e l'applicazione delle varie operazioni per diminuire il numero di vincoli possibili vengono costruiti dei thread appositi, in cui la comunicazione avviene tramite specifici canali. I linguaggi usati sono Go e Rust.

Repository, strumenti e istruzioni usati

La repository si trova a questo indirizzo: https://github.com/Cenze94/SudokuSolver. Al suo interno sono situate 5 cartelle. "Sudokus" contiene i file ".txt" con i valori di 3 sudoku, che vengono o possono essere caricati dai progetti per trovarne la soluzione e verificarne la procedura adottata. Dentro a "Relazione" ci sono tutti i file relativi alla relazione. "Go" e "Rust" contengono i file delle due versioni del progetto. "Old version" contiene i file relativi all'idea originale del progetto, in cui l'applicazione avrebbe dovuto contenere una GUI in Python comune ai due linguaggi, in modo da limitarli alla sola implementazione backend.
È stato usato Visual Studio Code come IDE, con gli appositi addon installati per i due linguaggi, e Windows come sistema operativo. Per Go è stato installato il compilatore apposito, quindi dall'interno della directory "./Go/" sono state eseguite le istruzioni "go build -o SudokuSolver.exe" e "./SudokuSolver", rispettivamente per compilare ed eseguire il progetto. Per Rust invece è stato utilizzato "Cargo" come package manager, usando il comando "env RUSTFLAGS="-A warnings" cargo run" dalla directory "./Rust/sudoku_solver/"; la variabile ambientale "RUSTFLAGS" è stata impostata per evitare i warning, dato che i nomi di variabili, funzioni e struct non seguono lo stile di formattazione standard del codice proposto da Rust. Per questo linguaggio sono stati usate "crossbeam", "num_bigint" e "num_traits" come librerie esterne, che dovrebbero essere scaricate e installate in automatico durante la prima compilazione.

Breve analisi della complessità asintotica dell'algoritmo di forza bruta

Nel caso di un sudoku con tutte le celle vuote, quindi con 9 valori possibili per cella, il numero totale di combinazioni è di circa 1,97 * 10^77. Ovviamente è una stima molto pessimistica, dato che di solito un sudoku contiene dai 20 ai 35 valori fissi già quando viene proposto, ma fornisce un'idea del perché un algoritmo di forza

bruta non è praticabile con tanti valori possibili.
Si consideri invece il seguente esempio:

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

I valori fissi sono 26, perciò il numero di possibili combinazioni diminuisce a circa 3,04 * 10^52. Come si può notare il valore ottenuto è diminuito parecchio, ma risulta ancora troppo elevato per un'applicazione pratica dell'algoritmo di forza bruta. Eliminando il maggior numero possibile di vincoli in base ai valori fissi, si ottiene il seguente risultato:

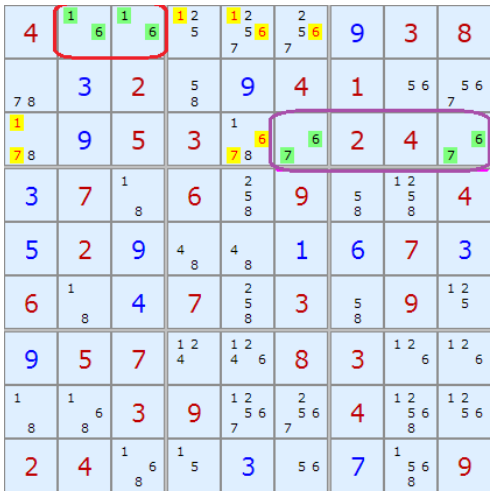
3 5 7	3 5 9	3 5 9	1 5 6 9	5 6 7 9	4	3 7 9	2	8
4	2 3 9	6	1 2 8 9	2 7 8 9	2 7 8	3 7 9	1 3 7 9	5
1	2 5 9	5 8 9	2 5 8 9	3	2 5 7 8	6	7 9	4 9
2 5	2 4 5 6 9	4 5 9	3	2 4 5 6 8	1	2 7 8 9	5 6 7 8 9	2 6 9
2 3 5	8	7	2 5 6	2 5 6	2 5 6	1	4	2 6 9
2 3 5	1 2 3 4 5 6	1 3 4 5	7	2 4 5 6 8	9	2 3 8	3 5 6 8	2 6
5 8	4 5	2	5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

Il numero di combinazioni possibili diventa perciò 3,14 * 10^28. Un valore molto minore, ma ancora intrattabile. Di conseguenza il metodo migliore è l'individuazione di altre tecniche per diminuire il numero di vincoli ancora presenti, quantomeno per minimizzare il più possibile il tempo d'esecuzione dell'algoritmo di forza bruta. Esse vengono analizzate nella sezione seguente.

Naked pairs e naked triples

Le prima due tecniche descritte nel sito[2] riguardano l'individuazione nelle righe, nelle colonne e nei box di coppie e triple di celle con un dominio comune rispettivamente di 2 e 3 valori. In questo modo è possibile stabilire che quei numeri saranno contenuti nelle celle individuate, e di conseguenza possono essere eliminati dalle altre celle della riga, della colonna o del box analizzati.

Più precisamente le naked pairs sono coppie di celle con lo stesso paio di valori possibili, come quelle evidenziate nell'immagine seguente:



I valori evidenziati in verde sono quelli delle naked pairs, mentre i numeri evidenziati in giallo sono quelli che verranno eliminati.
Nel caso delle naked triples le tre celle analizzate possono avere 2 o 3 valori possibili, purché il dominio totale risulti di 3 numeri. È possibile vederne un esempio nell'immagine seguente:

Possono esserci anche delle naked quads, ossia con gruppi di 4 celle, ma sono abbastanza rare e quindi non sono state implementate. Non ha senso analizzare gruppi di 5 celle, perché vorrebbe dire che automaticamente esiste una naked quad complementare, dato che in una riga, in una colonna o in un box ci sono al massimo 9 celle.

Struttura del codice e file principali

In entrambe le versioni del progetto è stata mantenuta la stessa struttura principale, con piccoli riadattamenti dovuti alle differenze fra i due linguaggi usati. I file principali sono:

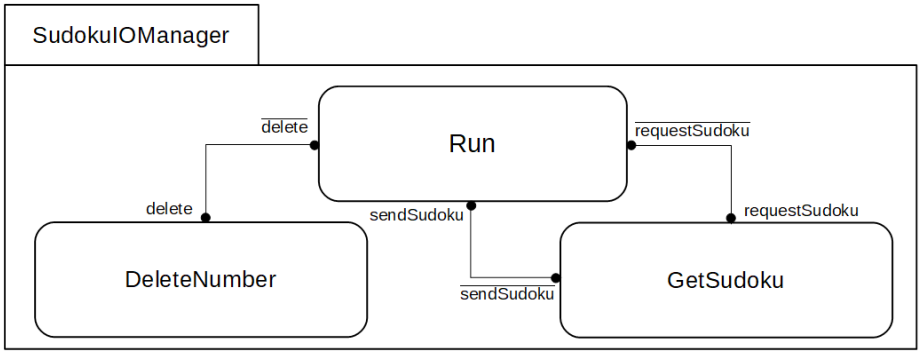
- "SudokuManager": contiene la rappresentazione del singolo sudoku e le funzioni che lo riguardano, come la cancellazione di un valore da una cella specifica, la restituzione dell'insieme di valori possibili di una data cella, la stampa del sudoku a terminale, e così via.

- “SudokuIOManager”: gestisce l’interazione fra il sudoku e il resto dell’applicazione; avvia un thread per la cancellazione asincrona dei valori possibili delle celle e la restituzione di un’eventuale copia del sudoku, oltre a contenere altri metodi sincroni.
- “ContraintsElimination”: contiene un metodo per l’eliminazione di tutti i vincoli del sudoku in base ai valori già fissati; avvia un thread per l’analisi e l’eventuale cancellazione dei numeri per le righe, uno per le colonne e uno per i box.
- “NakedPairs” e “NakedTriples”: hanno la stessa struttura del file precedente, ma per individuare rispettivamente le naked pairs e le naked triples.
- “CheckSudokuMethods”: contiene due metodi, il primo verifica se il sudoku è completo, ossia se ha soltanto valori fissati, mentre il secondo usa la struttura dei tre file precedenti e controlla se i valori fissati del sudoku sono corretti.
- “BruteForceMethods”: avvia l’algoritmo di forza bruta per la risoluzione del sudoku; prova a fissare ogni valore possibile di una data cella, elimina i vincoli non più validi, verifica se il risultato è corretto e avvia un nuovo thread per ripetere la procedura con la nuova matrice.
- “Main”: carica il sudoku, avvia il SudokuIOManager, esegue i vari step per la ricerca della soluzione e stampa il risultato ottenuto ad ogni passo; per verificare il funzionamento dell’algoritmo di forza bruta è prevista un’altra esecuzione, determinata da un’apposita variabile booleana, in cui viene caricato un sudoku quasi completo.

Struttura dei canali

Nel progetto sono stati usati i canali per evitare l’utilizzo di memoria condivisa, e quindi di lock. L’obiettivo è stato raggiunto nel caso di Go, mentre in Rust è stato usato un RwLock con il SudokuIOManager per poter compilare l’applicazione. Di seguito vengono rappresentati e descritti i processi e i canali usati, secondo quanto visto nella prima parte del corso. È stata aggiunta anche una notazione in più per i diagrammi CCS, ripresa dai diagrammi UML, per indicare in quali file sono contenuti i processi e i canali considerati. Nelle espressioni CCS, invece, è stato usato lo 0 per indicare la fine del metodo o del processo.

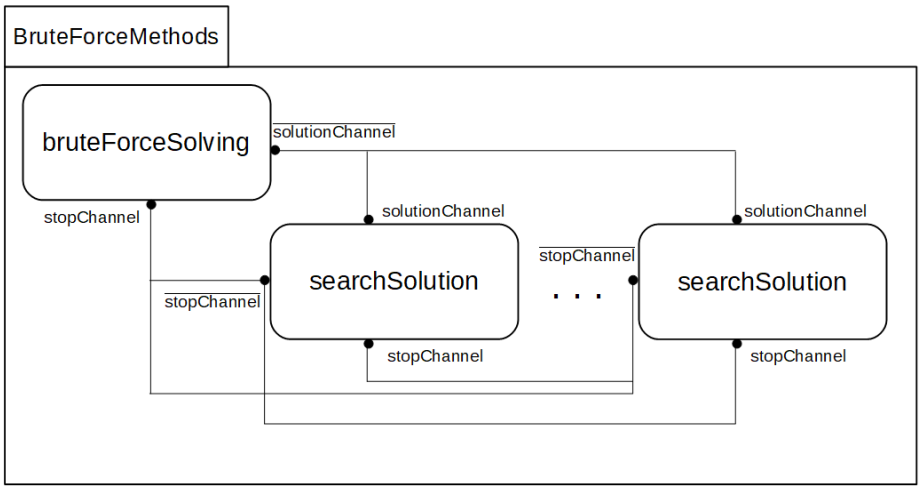
SudokuIOManager



$SudokuIOManager \stackrel{def}{=} (Run \mid DeleteNumber \mid GetSudoku) \setminus delete \setminus sendSudoku \setminus requestSudoku$
$Run \stackrel{def}{=} \overline{delete}.Run + \overline{requestSudoku}.Run1$
$Run1 \stackrel{def}{=} sendSudoku.Run$
$DeleteNumber \stackrel{def}{=} delete.DeleteNumber$
$GetSudoku \stackrel{def}{=} requestSudoku.GetSudoku1$
$GetSudoku1 \stackrel{def}{=} \overline{sendSudoku}.GetSudoku$

I metodi “DeleteNumber” e “GetSudoku” verrebbero invocati dagli altri file, quindi dall’ambiente esterno a Run. Per semplicità sono stati considerati due processi a parte, anche se tecnicamente non avviano thread, perché sono loro a gestire i canali indicati.

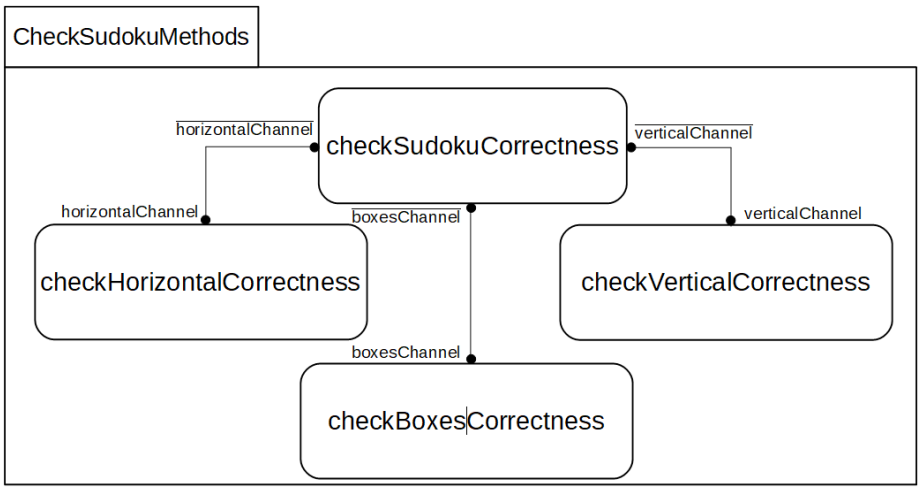
BruteForceMethods

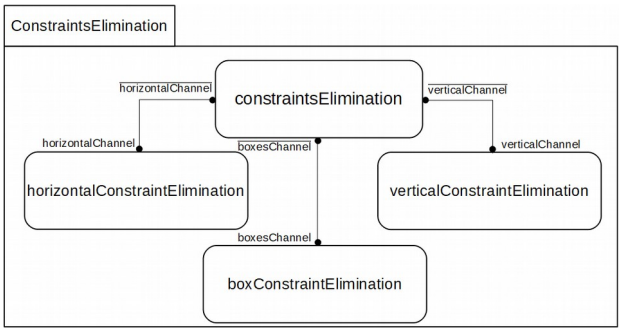


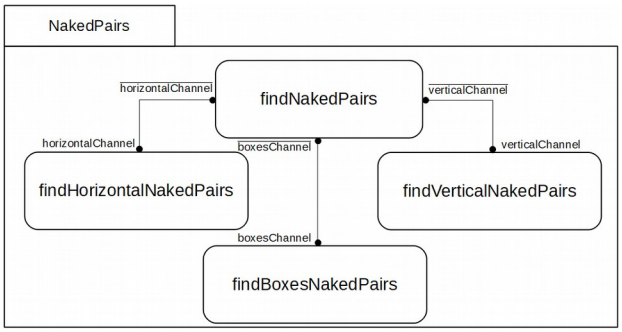
BruteForceMethods $\stackrel{def}{=}$ *bruteForceSolving \ solutionChannel \ stopChannel*
bruteForceSolving $\stackrel{def}{=}$ *(searchSolution \ solutionChannel \ stopChannel).bruteForceSolving1*
bruteForceSolving1 $\stackrel{def}{=}$ $\overline{solutionChannel}$*.bruteForceSolving2*
bruteForceSolving2 $\stackrel{def}{=}$ *stopChannel.0*
searchSolution $\stackrel{def}{=}$ *solutionChannel.0 + stopChannel.searchSolution1 + (searchSolution \ solutionChannel \ stopChannel).searchSolution3 + 0*
searchSolution1 $\stackrel{def}{=}$ $\overline{stopChannel}$*.searchSolution2*
searchSolution2 $\stackrel{def}{=}$ *stopChannel.0*
searchSolution3 $\stackrel{def}{=}$ *(searchSolution \ solutionChannel \ stopChannel).searchSolution3 + 0*

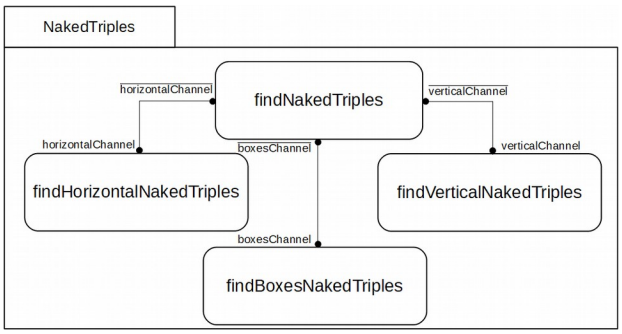
Il metodo “searchSolution” viene invocato un numero indefinito di volte, in base a quante combinazioni valide del sudoku analizzato esistono. Perciò nel diagramma sono stati aggiunti i 3 puntini per indicare che ce ne sono altri con lo stesso comportamento. Per quanto riguarda le espressioni CCS, è stato scelto di inserire l’avvio dei nuovi processi “searchSolution” come una scelta, senza però scrivere le condizioni in cui essa sia possibile. Questo è stato fatto per semplicità, dato che il processo verrebbe avviato soltanto se esiste un valore possibile da fissare che produce un sudoku ancora valido, e quindi si sarebbe dovuto aggiungere un pezzo di codice come condizione.

CheckSudokuMethods, ConstraintsElimination, NakedPairs e NakedTriples









CheckSudokuMethods $\stackrel{def}{=}$ *checkSudokuCorrectness \ horizontalChannel \ verticalChannel \ boxesChannel*
checkSudokuCorrectness $\stackrel{def}{=}$ *(checkHorizontalCorrectness \ horizontalChannel).checkSudokuCorrectness1*
checkSudokuCorrectness1 $\stackrel{def}{=}$ *(checkVerticalCorrectness \ verticalChannel).checkSudokuCorrectness2*
checkSudokuCorrectness2 $\stackrel{def}{=}$ *(checkBoxesCorrectness \ boxesChannel).checkSudokuCorrectness3*
checkSudokuCorrectness3 $\stackrel{def}{=}$ $\overline{horizontalChannel}$*.checkSudokuCorrectness4*
checkSudokuCorrectness4 $\stackrel{def}{=}$ $\overline{verticalChannel}$*.checkSudokuCorrectness5*
checkSudokuCorrectness5 $\stackrel{def}{=}$ $\overline{boxesChannel}$*.0*
checkHorizontalCorrectness $\stackrel{def}{=}$ *horizontalChannel.0*
checkVerticalCorrectness $\stackrel{def}{=}$ *verticalChannel.0*
checkBoxesCorrectness $\stackrel{def}{=}$ *boxesChannel.0*

ConstraintsElimination $\stackrel{def}{=}$ *constraintsElimination \ horizontalChannel \ verticalChannel \ boxesChannel*
constraintsElimination $\stackrel{def}{=}$ *(horizontalConstraintElimination \ horizontalChannel).constraintsElimination1*
constraintsElimination1 $\stackrel{def}{=}$ *(verticalConstraintElimination \ verticalChannel).constraintsElimination2*
constraintsElimination2 $\stackrel{def}{=}$ *(boxesConstraintElimination \ boxesChannel).constraintsElimination3*
constraintsElimination3 $\stackrel{def}{=}$ $\overline{horizontalChannel}$*.constraintsElimination4*
constraintsElimination4 $\stackrel{def}{=}$ $\overline{verticalChannel}$*.constraintsElimination5*
constraintsElimination5 $\stackrel{def}{=}$ $\overline{boxesChannel}$*.constraintsElimination6*
constraintsElimination6 $\stackrel{def}{=}$ *constraintsElimination + 0*
horizontalConstraintElimination $\stackrel{def}{=}$ *horizontalChannel.0*
verticalConstraintElimination $\stackrel{def}{=}$ *verticalChannel.0*
boxesConstraintElimination $\stackrel{def}{=}$ *boxesChannel.0*

Questi 4 file usano la stessa struttura e gli stessi nomi dei canali. Di fatto cambiano soltanto i nomi dei metodi, quindi nelle espressioni CCS sono stati descritti soltanto “CheckSudokuMethods” e “ConstraintsElimination”, perché nel primo file non c’è un’iterazione all’interno della funzione principale ma negli altri 3 sì.

Conclusioni

L’utilizzo dei canali ha permesso di semplificare la comunicazione fra i thread, oltre ad evitare di preoccuparsi di come gestire le eventuali variabili condivise. Anche se alcune situazioni non sono risultate immediate da gestire senza queste ultime, come ad esempio la comunicazione di modifiche avvenute dai thread del file “ConstraintsElimination” alla funzione principale, a causa dell’abitudine, è stato possibile apprezzare sempre di più l’utilità e la semplicità d’utilizzo dei canali. Come visto durante il corso, essi permettono di facilitare la gestione dei thread, ma questo non significa che non possano causare errori se usati male. In altre parole, risulterà comunque fondamentale basarsi su una buona progettazione, sebbene dovrebbe essere più semplice da realizzare con i canali rispetto ai tradizionali sistemi di condivisione e gestione della memoria.

Sitografia

1. https://it.wikipedia.org/wiki/Sudoku
2. http://www.sudokuwiki.org/sudoku.htm