30/12/2020

Generazione e

Progetto Parallel Algorithm A.A 20/21

Loris Cino

Indice

Introduzione al problema

L'Hitori è un rompicapo giapponese per certi versi simile al Sudoku. Alla sua base ci sono sostanzialmente tre regole:

- Non possono esserci numeri ripetuti in ciascuna riga o colonna.
- Non possono esserci due caselle annerite adiacenti.
- I bianchi devono essere connessi tra di loro. Non possono esserci bianchi isolati.

Lo scopo del gioco è annerire delle caselle per fare in modo che queste tre regole siano rispettate.

1	5	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

Figura 1: esempio di puzzle Hitori

I numeri ammessi sono da 1 sino alla dimensione del puzzle, nell'esempio 5.

Tecnica usata

Per risolvere il problema è stata utilizzata una tecnica di programmazione chiamata **Backtracking**. Questa tecnica consiste nel rappresentare tutte le possibili soluzioni in un albero e attraversarle fino ad ottenere una soluzione eliminando i rami che non rispettano i vincoli del problema.

```
procedure EXPLORE(node n)
    if REJECT(n) then return
    if COMPLETE(n) then
        OUTPUT(n)
    for n_i : CHILDREN(n) do EXPLORE(n_i)
```

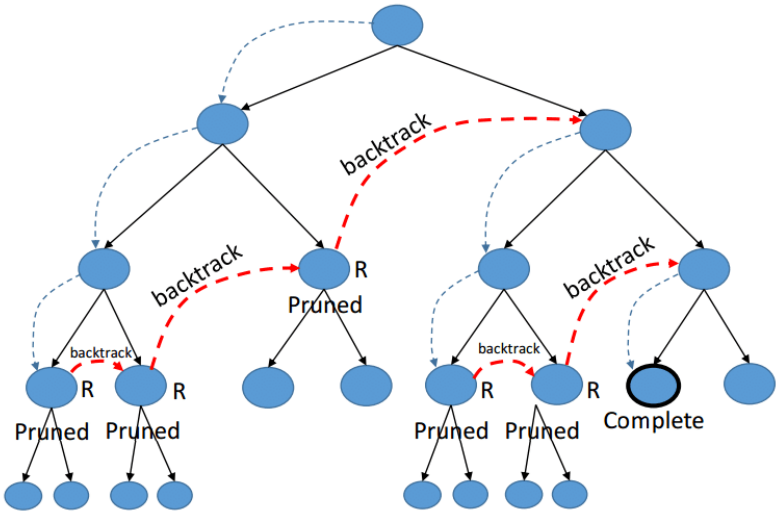


Figura 2: Pseudocodice e grafo backtracking preso da https://www.researchgate.net/figure/Backtracking-algorithm-taken-from-1_fig2_316610194

Questa tecnica è stata utilizzata in entrambi i casi, sia per la verifica che per la generazione di un puzzle risolvibile. Nel primo caso mi limito ad applicare le regole per trovare la soluzione. Nel secondo caso, invece, modifico anche i numeri presenti all'interno del puzzle per trovare un puzzle risolvibile secondo i vincoli del gioco.

Per generare un puzzle Hitori ogni nodo padre avrà 2xN nodi figli, dove N è la dimensione del puzzle in quanto in ogni casella è possibile avere N numeri ed ognuno di questi numeri può essere, nella soluzione, annerito o meno. Mentre nella verifica ogni nodo padre avrà solo due nodi figli in quanto il numero all'interno delle caselle è fissato.

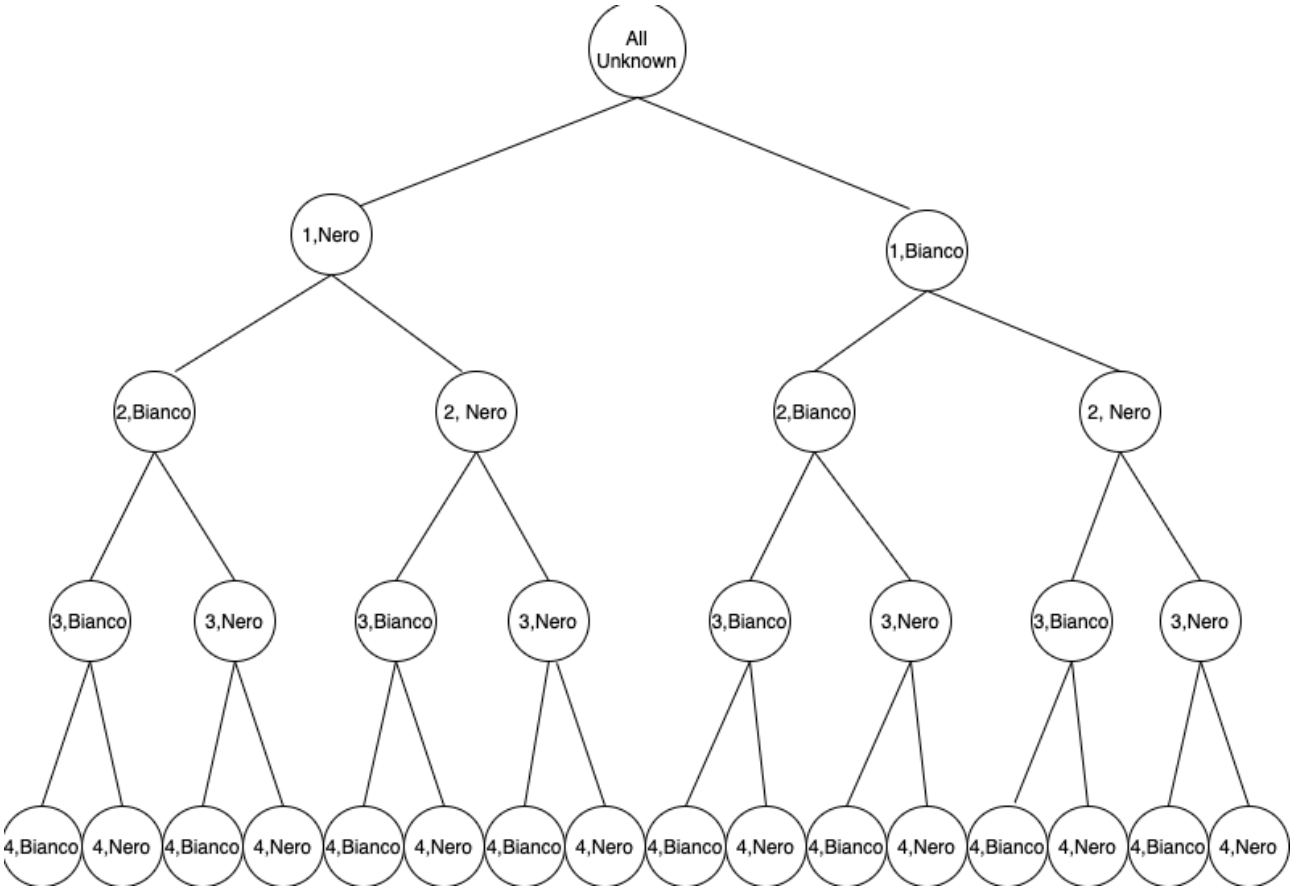


Figura 3: Esempio di un albero per la verifica di un puzzle 2x2

Come si può già dedurre il numero di nodi è enorme in entrambi i casi, è un numero esponenziale rispetto alla grandezza del puzzle. Nel primo caso ci saranno 2^{N^2} nodi generati ma applicando le regole del gioco, la maggior parte dei rami verrà potato e di conseguenza, per quanto scritto nell'articolo *V. N. Rao and V. Kumar, "On the efficiency of parallel backtracking," in IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 4, pp. 427-437, April 1993, doi: 10.1109/71.219757*, la complessità dell'algoritmo risulterà essere polinomiale.

Per quanto riguarda l'occupazione di memoria questo algoritmo risulta essere molto efficiente, con una implementazione ricorsiva viene impiegata solo una quantità di memoria lineare rispetto alla profondità del grafo. Anche dal punto di vista del calcolo, matrici di dimensione 100x100 ed anche oltre, possono essere generate e anche risolte in pochi secondi, come si vedrà dopo.

Implementazione

Il puzzle è stato modellato come una matrice di *block*, cioè un tipo di dato composto da due valori, uno *short int* e *char*. Il primo contiene il numero della casella corrispondente mentre il secondo conterrà lo stato della cella, cioè *b* se la casella sarà annerita, *w* se la casella è bianca, *u* se non si sa ancora lo stato della casella.

Funzione che verifica l'unicità del numero nelle righe/colonne

Questa funzione si occupa di verificare che non vi siano due bianchi nella stessa riga -o colonna- con lo stesso valore. Inoltre, se vi è una casella bianca con un determinato valore, annerisce tutte le caselle con lo stesso numero che hanno stato sconosciuto.

Funzione che verifica le regole di adiacenza

Semplicemente verifica che non ci siano due neri adiacenti e richiama un'altra funzione che si occupa di verificare che tutti i bianchi siano collegati.

Funzione di verifica delle regole

Questa funzione si occupa di verificare che tutte le regole siano rispettate e itera l'applicazione delle regole fino a quando non vi sono più variazioni nella tabella.

Tutte le funzioni viste fino ad ora ritornano -1 nel caso in cui una o più regole non siano rispettate.

Funzione di backtracking

Semplicemente implementa la tecnica di backtracking e si occupa di tenere il conto di quante caselle con stato sconosciuto sono rimaste, senza doverle contare ogni volta.

Parallelizzazione

Sono stati implementati due approcci, uno che rendeva parallela la ricerca nel ramo tramite backtracking (suggerito dal libro) ed un altro basato sulla suddivisione in blocchi e righe della matrice.

Il suggerimento del libro di testo era quello di effettuare il calcolo in maniera uguale fino ad un certo punto e successivamente dividere i rami tra i vari processi.

L'approccio seguito per la generazione è leggermente diverso in quanto, senza effettuare nessun calcolo, sono state fissati i valori di un certo numero di caselle e i diversi rami assegnati ai processi differiscono nei valori presenti in queste caselle.

1	4	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

1	5	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

Figura 4Rami diverso nella generazione

Per il caso della verifica l'approccio seguito è molto simile solo che non si bloccano i valori delle celle ma si blocca il colore della cella: se bianco o nero.

1	5	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

1	5	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

Figura 5Esempi di rami differenti per la verifica

Per evitare che il programma trovi sempre lo stesso puzzle i valori delle caselle bloccati (o lo stato delle caselle bloccate) sono state codificate in un intero. Ognuno

di questi numeri rappresenta un sottoramo da analizzare. Tutti questi interi sono stati inseriti in un vettore e vengono estratti in ordine casuale.

Per evitare una distribuzione non uniforme del carico di lavoro dovuto ad una diversa complessità dei rami, si è assegnato un numero molto grande di rami per processo.

Questo metodo purtroppo ha fallito in quanto i rami non conformi alle regole venivano subito potati mentre il ramo con la soluzione richiedeva lo stesso numero di calcoli.

Successivamente si è applicata una parallelizzazione differente basata sulla suddivisione della matrice. Per applicare la regola delle righe si è effettuata una suddivisione a righe della matrice, assegnando circa n/p righe ad ogni processo.

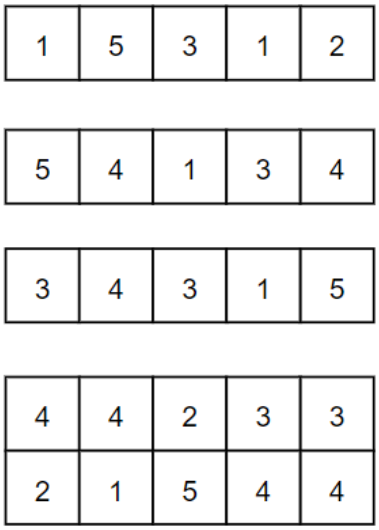


Figura 6Suddivisione esempio per righe

Per quanto riguarda le colonne si è fatta la stessa cosa ma i vari processi, quando vanno a ricevere le proprie colonne, le salvano come se fossero righe, ciò ha permesso di aumentare il **cache hit rate** e di avere performance, come si vedrà, **superlineari**.

1	5	3	4	2

5	4	4	4	1

3	1	3	2	5

1	3	1	3	4
2	4	5	3	4

Figura 7Suddivisione esempio per colonne

Per quanto riguarda la applicazione delle regole di adiacenza si è preferito optare per una suddivisione a blocchi. I blocchi sono stati creati con due righe e due colonne in più, per permettere lo scambio dei bordi.

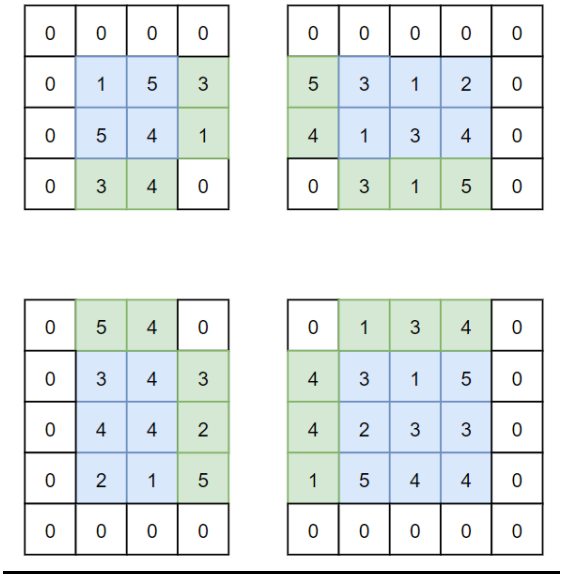


Figura 8 Suddivisione a blocchi, bordi in blu e bordi in verde

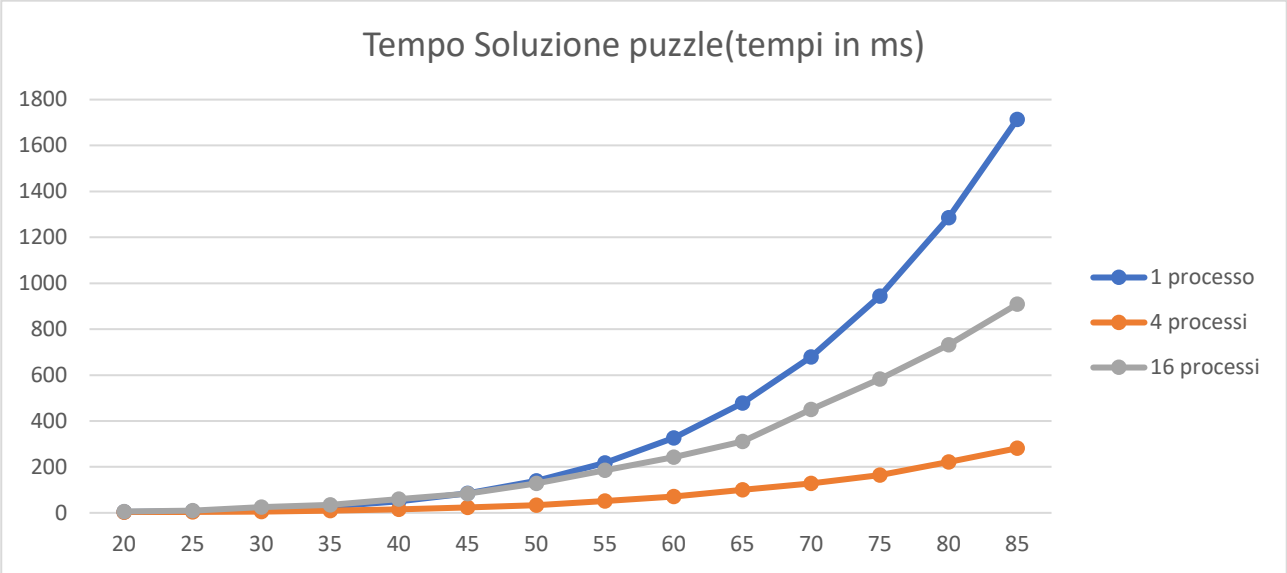
Terminazione (parallel backtracking)

Quando un processo trova una soluzione deve avvertire gli altri processi che possono terminare i loro calcoli in quanto il problema è stato risolto.

Nel caso in cui due processi trovino in tempi molto ravvicinati la soluzione, può succedere un processo, che ha in coda dei messaggi non letti, invochi la funzione *MPI_finalize* e ciò porta ad una terminazione del programma con errori. Per evitare ció è stato utilizzato *l'anello Dijkstra* in una versione leggermente semplificata.

Risultati

Come si è già detto si sono avuti risultati superlineari grazie ad una migliore ottimizzazione della cache.



I risultati riportano la media di 10 esecuzioni con 1, 4, 16 processori nella soluzione di puzzle di varie dimensioni da 20x20 fino a 85x85. Fino a matrici 40x40 non vi sono grosse differenze, si ha comunque uno speedup lineare ma, poiché tempi di calcolo sono di pochi millisecondi, non si apprezza nessuna differenza significativa, mentre per matrici di dimensioni 85x85 si ha addirittura uno speedup doppio rispetto a quello lineare.

La macchina su cui sono stati effettuati i test ha meno di 16 processori ed è per questo che si ha un peggioramento delle performance dell'algoritmo in quelle esecuzioni.