

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria dell'Informazione per Videogame
e Realtà Virtuale

Progettazione e implementazione di un software per giocare a scacchi basato su Stockfish

Design and implementation of a Stockfish-Based chess-playing software

Candidato:
Piero Ciccalé

Relatore:
Prof. Aldo franco Dragoni

Anno Accademico 2025-2026



UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria dell'Informazione per Videogame
e Realtà Virtuale

Progettazione e implementazione di un software per giocare a scacchi basato su Stockfish

Design and implementation of a Stockfish-Based chess-playing software

Candidato:
Piero Ciccalé

Relatore:
Prof. Aldo franco Dragoni

Anno Accademico 2025-2026

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria dell'Informazione per Videogame e Realtà Virtuale
Via Brecce Bianche – 60131 Ancona (AN), Italy

Renewed

Every ending hides a new beginning. Life will toss and turn us, but it can only break us if we let it. It's always, always worth trying.

Every time.

Unravel

Sommario

Gli scacchi sono usati come banco di prova dell'intelligenza artificiale dagli anni Quaranta e Cinquanta, e Stockfish, un motore open source che unisce una ricerca alpha-beta a una valutazione con reti neurali, ne è un esempio attuale. Capire come funziona un motore di questo tipo è difficile, perché le informazioni sono distribuite tra blog, documentazione e codice sorgente, e usarlo in un'applicazione richiede di gestire un processo esterno e di ottenere dati che il protocollo UCI non offre direttamente. La tesi ricostruisce la storia dei motori, da Turing e Shannon fino ad AlphaZero e all'adozione della rete NNUE, e descrive Stockfish 18 a partire dal post di rilascio, dalla documentazione ufficiale e dal codice sorgente, indicando le fonti e i punti che esse non chiariscono. Presenta poi StockfishBoard, un'applicazione sviluppata in Unity che usa Stockfish come motore esterno per giocare e analizzare posizioni. Dallo studio emerge che Stockfish valuta le posizioni con reti neurali dentro una ricerca alpha-beta eseguita da più thread. L'applicazione affida al motore la verifica della legalità delle mosse e risolve i problemi di concorrenza legati all'analisi asincrona. Il contributo è di documentazione, analisi e sviluppo. Tra gli sviluppi futuri ci sono l'aggiornamento a Stockfish 19, una maggiore robustezza dell'applicazione e la sua prova con altri utenti.

Indice

1 Introduzione **1**

2 Dagli albori dell'intelligenza artificiale ai motori scacchistici moderni **3**

2.1 Premessa 3

2.2 I primi sviluppi dell'intelligenza artificiale 4

2.2.1 Le origini: Turing, Shannon e gli scacchi come banco di prova 4

2.2.2 La Conferenza di Dartmouth (1956) e la nascita del termine "intelligenza artificiale" 6

2.3 Storia dei motori scacchistici 6

2.3.1 I primi tentativi: Turochamp e i programmi pionieristici (1948-1962) 6

2.3.2 L'era della forza bruta: minimax e potatura alpha-beta . . . 8

2.3.3 Deep Blue e la sconfitta di Kasparov (1997) 9

2.3.4 L'era open source: dalla nascita di Stockfish ai suoi concorrenti 10

2.4 L'avvento del machine learning negli scacchi 12

2.4.1 AlphaZero e l'apprendimento per rinforzo (2017) 13

2.4.2 NNUE: l'ibridazione tra ricerca classica e reti neurali in Stockfish (2018-2020) 14

2.5 Sintesi del capitolo 16

3 Stockfish 18: architettura e funzionamento **18**

3.1 Introduzione a Stockfish 18 18

3.2 Architettura generale del motore 19

3.3 Rappresentazione della scacchiera 21

3.3.1 Le tabelle di trasposizione 23

3.4 La ricerca in parallelo e i thread multipli 25

3.4.1 Lazy SMP: l'idea e la sua storia 26

3.4.2 L'organizzazione dei thread nel codice 28

3.4.3 La tabella di trasposizione condivisa e gli accessi senza blocchi 30

3.4.4 La votazione tra i thread per scegliere la mossa 31

3.4.5 I nodi NUMA e la memoria condivisa 32

3.4.6 La scalabilità con il numero di thread 33

3.5 La valutazione con la rete NNUE 34

3.5.1 Dalla valutazione classica alla sola rete neurale 35

3.5.2 Le caratteristiche in ingresso nella rete 36

3.5.3 L'accumulatore e l'aggiornamento incrementale 36
3.5.4 Gli strati della rete e la quantizzazione 37
3.5.5 La rete grande e la rete piccola 38
3.5.6 Dalla uscita della rete alla valutazione finale 39
3.6 Il machine learning nell'addestramento delle reti 40
3.6.1 I dati di addestramento e il ruolo di Leela Chess Zero 40
3.6.2 Il trainer NNUE-Pytorch e la funzione di perdita 41
3.6.3 La pipeline automatizzata e riproducibile di Stockfish 18 . . . 42
3.6.4 Fishtest: il collaudo distribuito di reti e modifiche 42
3.7 Ricerca con rete neurale e approccio di AlphaZero a confronto 43
3.8 Sintesi del capitolo 44

4 Stockfish Board **46**
4.1 Presentazione del progetto 46
4.2 Architettura del sistema 47
4.2.1 Rappresentazione della scacchiera e dei pezzi 49
4.3 Integrazione con il motore Stockfish 50
4.3.1 Meccanismo a stati e validazione delle mosse 51
4.4 Logica di gioco e validazione delle mosse 53
4.5 Interfaccia utente e feedback al giocatore 55
4.5.1 Editor di posizioni personalizzate 56
4.5.2 Navigazione nella cronologia e sistema di cache 57
4.6 Principali problemi incontrati e soluzioni 59
4.7 Sintesi del capitolo 60

5 Conclusioni **62**
5.1 Sviluppi futuri 63

Elenco delle figure

3.1 Albero di ricerca semplificato con i limiti alpha e beta. Illustrazione dell'autore. 20
3.2 Diagramma di flusso di Stockfish 18. Illustrazione dell'autore. 21
3.3 Diagramma del comportamento multithread in Lazy SMP. Illustrazione dell'autore. 27
3.4 Diagramma temporale del comportamento dei thread durante una ricerca. Illustrazione dell'autore. 29
3.5 Schema semplificato della rete NNUE grande di Stockfish 18. Illustrazione dell'autore. 38

4.1 Diagramma di flusso di StockfishBoard. Illustrazione dell'autore. . . 47
4.2 Ordine di stratificazione dell'input dell'utente. Illustrazione dell'autore. 50
4.3 Meccanismo a stati che gestisce la risposta di Stockfish. Illustrazione dell'autore. 52
4.4 Pulsanti di scelta del pezzo di promozione. A sinistra il prefab per il bianco, a destra quello per il nero. Illustrazione dell'autore. 54
4.5 Sistema che gestisce la cache. Illustrazione dell'autore. 58

Capitolo 1

Introduzione

Gli scacchi sono uno dei campi in cui l'intelligenza artificiale è stata messa alla prova fin dall'inizio. Nel 1948 Turing e Champernowne elaborarono Turochamp, un insieme di regole per scegliere la mossa milgiore, e nel 1950 Shannon descrisse una funzione di valutazione abbinata all'algoritmo minimax, sottolineando l'interesse teorico della questione . Per alcuni decenni i motori sono migliorati con la ricerca esaustiva e la potatura alpha-beta, fino a Deep Blue, che nel 1997 sconfisse il campione del mondo Garry Kasparov (Campbell, Hoane e Hsu, 2002). Nel 2017 AlphaZero ha mostrato un'altra strada: impara a giocare soltanto con partite contro se stesso (Silver et al., 2017b). Stockfish, motore open source nato nel 2008 da Glaurung (Chess Programming Wiki, Stockfish), unisce oggi le due tradizioni. Mantiene una ricerca di tipo alpha-beta (Maharaj et al., 2021, §2.2.1) e dal 2020 usa una rete neurale chiamata NNUE (Blog di Stockfish, Stockfish 12). Nella versione 18, uscita il 31 gennaio 2026, il codice di valutazione non contiene più termini scritti a mano (Blog di Stockfish, Stockfish 18; Stockfish, codice sorgente, evaluate.cpp, tag sf 18).

Capire come funziona un motore di questo tipo è difficile. Le informazioni sono distribuite tra il blog del progetto, la documentazione e il codice sorgente. Inoltre, Il motore cambia in fretta: la documentazione registra due nuove versioni dell'architettura di rete già dal 4 e dal 12 febbraio 2026 (Documentazione di Stockfish, storia delle architetture di rete), e il 5 settembre 2026 è uscito Stockfish 19 (Blog di Stockfish, Stockfish 19). Usare il motore in un'applicazione pone un secondo problema. Nel protocollo UCI (Universal chess Interface) lo stato della partita è gestito dall'interfaccia grafica, mentre il motore resta più semplice (Chess Programming Wiki, UCI). Chi sviluppa un'interfaccia deve quindi comunicare con un processo esterno la cui analisi può richiedere diversi secondi senza bloccare l'applicazione, e deve ottenere dal motore informazioni, come l'elenco delle mosse legali, che il protocollo non prevede tra i suoi comandi, come mostra il Capitolo 4.

Dal punto di vista scientifico, Stockfish è un caso in cui una ricerca classica e una valutazione appresa lavorano insieme. Le sue reti si addestrano con dati in parte prodotti da Leela Chess Zero, un motore ispirato ad AlphaZero (Blog di Stockfish, Stockfish 18; Sito di Leela Chess Zero, pagina iniziale), quindi i due approcci descritti

nel Capitolo 2 non sono separati. Il progetto è inoltre open source, con licenza GNU GPL versione 3 (Stockfish, About): codice e documentazione permettono di controllare le affermazioni sul suo funzionamento. Dal punto di vista tecnico, il motore raccoglie due questioni di ingegneria. La prima è valutare le posizioni in fretta con una rete neurale: la rete è pensata per essere aggiornata a ogni mossa e valutata dal processore, senza acceleratori (Documentazione di Stockfish, NNUE). La seconda è far lavorare molti thread su una tabella condivisa senza sincronizzarli, scelta che il codice motiva con il costo in punti Elo (Stockfish, codice sorgente, tt.h, tag sf 18). Dal punto di vista applicativo, UCI è un protocollo aperto che permette ai motori di dialogare con altri programmi (Chess Programming Wiki, UCI).

La tesi descrive come Stockfish 18 sceglie una mossa, a partire da fonti ufficiali e dal codice sorgente, e lo colloca nella storia degli scacchi e dell'intelligenza artificiale. Presenta poi StockfishBoard, un'applicazione sviluppata in Unity che usa Stockfish come motore esterno per giocare e analizzare posizioni.

I contributi della tesi sono tre: 1. Una sintesi storica del rapporto tra scacchi e intelligenza artificiale, dalle origini teoriche ad AlphaZero e all'adozione di NNUE in Stockfish. 2. Una descrizione di Stockfish 18 basata sul blog, sulla documentazione e sul codice del tag sf_18, in cui ogni affermazione ha la propria fonte. 3. L'applicazione StockfishBoard, progettata e realizzata dall'autore: l'architettura a livelli, l'uso del comando "perft" per ottenere le mosse legali, la macchina a stati che interpreta le risposte del motore, la cache delle analisi e le soluzioni ai problemi di concorrenza incontrati. La tesi non propone nuovi algoritmi e non misura la forza del motore. Il suo contributo è di documentazione, analisi e sviluppo.

Il Capitolo 2 ricostruisce la storia: le origini teoriche con Turing, Shannon e la conferenza di Dartmouth del 1956, l'evoluzione dei motori fino a Deep Blue e ai motori open source, e l'ingresso dell'apprendimento automatico con AlphaZero e NNUE. Il Capitolo 3 spiega come funziona Stockfish 18. Tratta la rappresentazione della posizione e la tabella di trasposizione, la ricerca in parallelo con più thread, la valutazione con la rete NNUE, l'addestramento delle reti e il collaudo su Fishtest, e il confronto con AlphaZero e Leela. Il Capitolo 4 presenta StockfishBoard: l'architettura, la rappresentazione della scacchiera, l'integrazione con Stockfish tramite UCI, la validazione delle mosse, l'interfaccia, la navigazione nella cronologia con la cache delle analisi e i problemi di concorrenza, con i limiti e gli sviluppi possibili. L'ultimo capitolo raccoglie le conclusioni.

Capitolo 2

Dagli albori dell'intelligenza artificiale ai motori scacchistici moderni

2.1 Premessa

Gli scacchi occupano un posto particolare nella storia dell'intelligenza artificiale. Fin dai primi anni dell'informatica, il gioco è stato scelto come banco di prova per verificare se una macchina potesse affrontare un compito che richiede pianificazione, valutazione e decisione, proprio per la combinazione tra regole semplici e complessità combinatoria che lo caratterizza. Non sorprende quindi che la storia dei motori scacchistici si intrecci, fin dalle origini, con quella dell'intelligenza artificiale nel suo complesso: i problemi affrontati per insegnare a un computer a giocare a scacchi, rappresentare una posizione, valutarla, scegliere la mossa migliore tra un numero enorme di alternative, sono gli stessi che, in forme diverse, attraversano l'intera disciplina [1].

Questo capitolo ripercorre tale storia lungo tre fasi distinte. La prima riguarda i primi sviluppi teorici dell'intelligenza artificiale, quando gli scacchi vennero proposti, prima ancora che esistessero computer in grado di eseguirli davvero, come esempio concreto su cui costruire un ragionamento formale sul "pensiero meccanizzato". La seconda ripercorre l'evoluzione dei motori scacchistici veri e propri, dai primi programmi pionieristici fino ai sistemi capaci di sconfiggere il campione del mondo in carica. La terza affronta il cambiamento più recente, l'ingresso delle reti neurali e dell'apprendimento per rinforzo in un campo che per decenni era stato dominato da tecniche di ricerca e valutazione interamente programmate a mano.

Questo percorso non è soltanto un contesto storico fine a sé stesso. Il motore utilizzato nel lavoro descritto nel Capitolo 4, Stockfish, è esso stesso il punto d'arrivo di questa storia: un sistema di ricerca classica che negli ultimi anni ha incorporato proprio quelle tecniche di apprendimento automatico discusse nella parte finale di questo capitolo [2]. Ripercorrere come si sia arrivati a questo punto aiuta a comprendere non solo cosa fa Stockfish, ma perché è costruito nel modo in cui è costruito.

2.2 I primi sviluppi dell'intelligenza artificiale

Prima che il problema potesse essere affrontato in termini computazionali, gli scacchi avevano già una lunga storia come oggetto di fascinazione per l'idea di una macchina capace di "pensare". Nel 1770 l'inventore ungherese Wolfgang von Kempelen presentò "Il Turco", un automa che sembrava giocare a scacchi in autonomia, ma che in realtà nascondeva un giocatore umano al proprio interno.
Per quasi due secoli, l'idea di una macchina scacchistica restò più un espediente da esibizione che un problema scientifico reale. La situazione cambiò a partire dalla fine degli anni Quaranta, quando alcuni dei fondatori dell'informatica teorica iniziarono a chiedersi, questa volta sul serio, se un calcolatore potesse giocare a scacchi seguendo soltanto un insieme di regole esplicite, senza alcun inganno. Le due sezioni seguenti ricostruiscono questo passaggio: prima il lavoro teorico di Turing e Shannon, poi il momento in cui questo filone di ricerca confluì nella definizione stessa del campo dell'intelligenza artificiale.

2.2.1 Le origini: Turing, Shannon e gli scacchi come banco di prova

Il primo a tentare concretamente di far giocare una macchina a scacchi fu Alan Turing, che nel 1948, insieme al collega David Champernowne (economista e matematico inglese), elaborò un insieme di regole capaci di determinare la mossa successiva data una posizione: un programma che i due chiamarono Turochamp. Nel 1948 nessuna macchina esistente poteva eseguire quelle istruzioni. Turing finì per applicarle a mano, consultando le proprie regole scritte su carta per decidere ogni mossa durante una partita del 1952, che perse contro il collega Alick Glennie [3].
Turochamp non era un esercizio isolato, ma parte di una riflessione più ampia che Turing stava sviluppando proprio in quegli anni sulla possibilità che una macchina potesse pensare. Nello stesso 1950 in cui Shannon pubblicava il proprio lavoro sugli scacchi, Turing dava alle stampe sulla rivista Mind l'articolo "Computing Machinery and Intelligence", che si apre con una delle frasi più citate nella storia dell'informatica:

> *"I propose to consider the question, 'Can machines think?'"*
>
> Turing [4], *Computing Machinery and Intelligence*

Turing riconosceva, nello stesso articolo, che la domanda così posta fosse troppo ambigua per essere affrontata direttamente, poiché termini come "macchina" e "pensare" non ammettono definizioni condivise. Propose quindi di sostituirla con un criterio operativo più concreto, il celebre gioco dell'imitazione, in cui un interrogatore umano cerca di distinguere, attraverso uno scambio scritto di domande e risposte, una macchina da un essere umano. Questo criterio, oggi noto come test di Turing, affrontava sul piano filosofico la stessa domanda che Turochamp affrontava sul piano pratico qualche anno prima: può una macchina, seguendo soltanto regole esplicite,

riprodurre un comportamento che normalmente associamo all'intelligenza? Scacchi e conversazione erano, in questo senso, due banchi di prova diversi per la stessa ipotesi di fondo [4].
Due anni dopo il lavoro di Turing e Champernowne, Claude Shannon pubblicò l'articolo "Programming a Computer for Playing Chess". Shannon era interessato agli scacchi non in quanto gioco in sé, ma come esempio concreto di quello che chiamava "mechanized thinking", il pensiero meccanizzato: non limitarsi a scrivere un programma che giocasse, ma usare gli scacchi per esplorare cosa significhi, in termini computazionali, ragionare. Lo stesso Shannon è esplicito su questo punto:

> *"Although perhaps of no practical importance, the question is of theoretical interest"*
>
> Shannon [1], *Programming a Computer for Playing Chess*

Gli scacchi, per Shannon, non avevano alcuna importanza pratica diretta: il loro valore stava nel fornire un problema abbastanza semplice da formalizzare, ma abbastanza complesso da rendere non banale qualunque soluzione, nella speranza che la sua soluzione servisse da punto di partenza per problemi simili e più importanti [1].
Nell'articolo, Shannon introdusse due idee che avrebbero definito il campo per i decenni successivi. La prima è una funzione di valutazione: un modo per assegnare a ogni possibile posizione sulla scacchiera un punteggio numerico che ne rifletta la bontà per uno dei due giocatori, calcolato sommando fattori come il materiale posseduto, la struttura dei pedoni e la mobilità dei pezzi. La seconda è la procedura che oggi chiamiamo minimax, nome che Shannon non usa: essa impiega la funzione di valutazione per scegliere la mossa migliore non guardando soltanto al vantaggio immediato, ma anche a come l'avversario risponderebbe a ciascuna mossa possibile, assumendo che anch'esso giochi nel modo più efficace a propria disposizione. Da qui il nome, poiché un giocatore cerca di massimizzare il proprio vantaggio mentre presume che l'altro cerchi di minimizzarlo [1].
Questo principio è alla base dei motori a ricerca ad albero, incluso quello utilizzato nel Capitolo 4, per quanto arricchito nel tempo da ottimizzazioni come la potatura alpha-beta discussa nella Sezione 2.3.2. I sistemi basati su reti neurali e ricerca Monte Carlo, trattati nella Sezione 2.4, seguono invece un'altra strada. Turing e Shannon non lavorarono a questi contributi isolatamente l'uno dall'altro. Si erano conosciuti ai Bell Labs, dove Turing si trovava in visita nell'ambito di un giro dei progetti americani di crittografia militare, e i due si incontravano ogni giorno per il tè nella mensa [5].

2.2.2 La Conferenza di Dartmouth (1956) e la nascita del termine "intelligenza artificiale"

Nell'estate del 1956 si tenne al Dartmouth College, nel New Hampshire, un progetto di ricerca oggi ampiamente considerato l'evento fondativo dell'intelligenza artificiale come campo di studio autonomo. L'iniziativa nacque da una proposta scritta l'anno precedente dagli informatici statunitensi John McCarthy, Marvin Minsky e Nathaniel Rochester e da Claude Shannon, già incontrato nella sezione precedente. La proposta prevedeva uno studio di due mesi con dieci partecipanti e partiva da una congettura tanto audace quanto, a quell'epoca, del tutto indimostrata [6]:

> *"A machine can be made to simulate it"*
>
> McCarthy e altri [6], *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*

La frase completa, riferita a ogni aspetto dell'apprendimento o qualunque altra caratteristica dell'intelligenza, sosteneva che questi potessero in linea di principio essere descritti con precisione sufficiente da permettere a una macchina di riprodurli. Il termine "intelligenza artificiale" compare già nel titolo di questa proposta [6].
Il progetto non produsse i risultati ambiziosi che McCarthy si era prefissato: nessuna macchina pensante nacque nell'arco di quell'estate. Il nome coniato per l'occasione, e l'idea stessa che l'intelligenza fosse un fenomeno descrivibile e riproducibile computazionalmente, sono sopravvissuti ben oltre, definendo l'intero campo di ricerca di cui la storia dei motori scacchistici, ripresa nella Sezione 2.3, rappresenta una delle applicazioni più longeve.

2.3 Storia dei motori scacchistici

Il lavoro teorico di Turing e Shannon pose le basi concettuali per affrontare gli scacchi in termini computazionali, ma passarono ancora diversi anni prima che queste idee si traducessero in programmi capaci di giocare partite complete, e diversi decenni prima che un programma potesse competere alla pari con i migliori giocatori umani. Questa sezione ripercorre quel percorso: prima i tentativi pionieristici che seguirono immediatamente il lavoro di Shannon, poi un esame più approfondito degli algoritmi di ricerca che quei tentativi misero in pratica, infine il traguardo simbolico raggiunto nel 1997 e la nascita dei motori open source.

2.3.1 I primi tentativi: Turochamp e i programmi pionieristici (1948-1962)

Turochamp, introdotto nella Sezione 2.2.1, è il primo tentativo documentato di applicare regole esplicite al gioco degli scacchi, e il suo funzionamento contiene già

in forma riconoscibile molte delle idee che i motori successivi avrebbero sviluppato. Secondo Kasparov e Friedel [3], che ne hanno ricostruito la storia, Turing si era dato un obiettivo preciso: una macchina che, davanti a una posizione ordinaria, indicasse dopo due o tre minuti di calcolo una mossa legale passabilmente buona.
Il cuore del programma è una funzione di valutazione, cioè una regola che assegna un punteggio numerico a una posizione. Il termine principale è il materiale, con valori leggermente diversi da quelli convenzionali di oggi: pedone 1, cavallo 3, alfiere 3,5, torre 5, donna 10. A questo si aggiungono termini posizionali: il valore di un pezzo non corrisponde solo a quello assegnato a inizio partita, ma dipende anche dalla sua posizione sulla scacchiera. Per quasi tutti i pezzi vale la mobilità: quante più mosse possibili hanno, tanto più punteggio guadagnano. Per il re vale la logica opposta, perché è il pezzo da proteggere: più è esposto, cioè più linee aperte ha attorno, più il suo punteggio scende [3].
Il secondo elemento è la ricerca. Il programma esamina le mosse per alcune semimosse in avanti, tre nella partita che Turing giocò a mano, e confronta le posizioni raggiunte con la funzione di valutazione [3]. Secondo la Chess Programming Wiki, il sistema anticipava anche la strategia minimax e una profondità di ricerca variabile, in cui le mosse più promettenti vengono approfondite più delle altre [7]. Il lavoro di Turing e Champernowne risale al 1948, due anni prima dell'articolo di Shannon descritto nella Sezione 2.2.1. Il terzo elemento è la selettività. Come scrisse Champernowne, la maggior parte dell'attenzione fu dedicata a decidere quali mosse approfondire. Le catture andavano seguite almeno fino al punto in cui non fosse più possibile un'altra cattura immediata, scacchi e mosse forzanti andavano seguiti più a lungo, mentre alcune mosse venivano scartate subito come prive di senso e altre inseguite molto in profondità lungo certe linee. Il primo criterio introduce quella che oggi si chiama ricerca di quiescenza: valutare una posizione nel mezzo di uno scambio di pezzi darebbe un punteggio ingannevole, perché il materiale sta per cambiare, e quindi la ricerca prosegue finché la posizione si stabilizza. Il secondo criterio richiama la strategia di Tipo B di Shannon, descritta nella Sezione 2.3.2, e ne condivide il rischio: scartare una mossa che sembrava inutile e che invece non lo era. Il problema era che nessuna macchina esistente poteva eseguire questa mole di istruzioni [3].
Turing decise allora di fare lui stesso da processore, seguendo le regole con carta e matita. Giocò una prima partita contro la moglie di Champernowne, che era una principiante, e la vinse, poi una seconda contro il collega Alick Glennie, a Manchester nel 1952, e la perse. Dovendo eseguire tutti i calcoli a mano, ogni mossa richiedeva un tempo estremamente lungo, tanto che lo stesso Champernowne, che aveva assistito Turing nella partita del 1952, commentò così il procedimento:

> *"the arithmetic was extremely tedious with pencil and paper"*

Champernowne, citato in Kasparov e Friedel [3], *Reconstructing Turing's "paper machine"*

I primi programmi in grado di giocare partite complete su un calcolatore arrivarono più tardi. Secondo la ricostruzione storica di Chess.com, nel 1957 Alex Bernstein scrisse per un calcolatore IBM 704 quello che viene indicato come il primo programma scacchistico completo, e nel 1962 Alan Kotok, studente al MIT, sviluppò per la propria tesi di laurea il primo programma a giocare partite di scacchi regolari con un certo livello di credibilità [8]. Questi primi programmi condividevano un limite strutturale comune: la potenza di calcolo disponibile permetteva di esplorare soltanto un numero ridotto di mosse future, rendendo necessario scegliere con attenzione quali rami dell'albero delle possibilità valesse la pena esaminare. È proprio questo problema, già individuato da Shannon nel proprio articolo del 1950, a motivare lo sviluppo degli algoritmi di ricerca discussi nella sezione seguente [1].

2.3.2 L'era della forza bruta: minimax e potatura alpha-beta

Il nucleo teorico dell'articolo di Shannon merita un esame più attento di quanto fatto nella Sezione 2.2.1. Shannon lo formulò nel 1950, con diversi decenni di anticipo sugli algoritmi di ricerca oggi utilizzati: la funzione di valutazione e l'esplorazione dell'albero delle mosse che descrisse sono ancora alla base dei motori scacchistici. Da allora sono cambiati la velocità di calcolo e le tecniche di potatura, ma l'impianto resta lo stesso [1].
Shannon parte da un'osservazione pratica: anche un calcolatore estremamente veloce non può esplorare per intero l'albero delle mosse possibili in una partita a scacchi, poiché il numero di posizioni raggiungibili cresce in modo esplosivo a ogni semi-mossa aggiuntiva. Da questo vincolo deriva la necessità di una funzione di valutazione: invece di calcolare fino alla fine della partita se una posizione porti alla vittoria, il programma deve fermarsi dopo un numero limitato di mosse e stimare numericamente quanto quella posizione intermedia sia favorevole, sommando fattori come il materiale posseduto da ciascun giocatore e la mobilità dei pezzi [1].
Per decidere quale mossa giocare sulla base di questa valutazione, Shannon adatta e usa la procedura che oggi chiamiamo minimax: si esplora l'albero delle mosse fino alla profondità consentita, si valutano le posizioni finali raggiunte, e si risale l'albero assumendo che, a ogni livello, il giocatore di turno scelga la mossa più favorevole per sé, mentre l'avversario, al livello successivo, scelga quella più sfavorevole per il primo. Il nome dell'algoritmo riflette proprio questa doppia assunzione: un giocatore massimizza il proprio vantaggio, l'altro lo minimizza [1]. Shannon distingue poi, ed è forse il contributo più duraturo del suo lavoro, due strategie alternative per decidere quali rami dell'albero esplorare, che lui stesso chiama strategia di Tipo A e strategia di Tipo B. La strategia di Tipo A esplora in modo esaustivo tutte le mosse possibili fino a una profondità fissa uguale per ogni ramo, un approccio semplice da implementare ma limitato dalla potenza di calcolo disponibile: più la profondità richiesta è alta, più il numero di posizioni da valutare cresce rapidamente. La strategia di Tipo B, al contrario, usa una conoscenza specifica del gioco per

scartare in anticipo le mosse giudicate poco promettenti, concentrando la capacità di calcolo disponibile sulle linee ritenute più rilevanti, in modo più simile a come un giocatore umano restringe la propria attenzione durante una partita [1].
Questa distinzione anticipa un problema che avrebbe occupato la ricerca sui motori scacchistici per i decenni successivi. La strategia di Tipo A garantisce di non trascurare mosse potenzialmente vincenti, ma è costosa dal punto di vista computazionale. La strategia di Tipo B è più efficiente, ma rischia di scartare per errore una mossa valida se la conoscenza usata per guidare la selezione è imperfetta, come inevitabilmente accade. La soluzione che si sarebbe affermata non consistette nello scegliere definitivamente l'una o l'altra strategia, ma nel trovare un modo per rendere la ricerca esaustiva di Tipo A più efficiente senza perderne le garanzie: la potatura alpha-beta, discussa di seguito [1].
A differenza del minimax puro, che valuta ogni singola posizione foglia dell'albero esplorato, la potatura alpha-beta osserva che alcuni rami possono essere scartati senza essere mai esplorati, non perché giudicati meno promettenti secondo un criterio euristico, come nella strategia di Tipo B, ma perché è matematicamente dimostrabile che non potranno mai influenzare la scelta finale, qualunque sia il loro valore effettivo. Se durante l'esplorazione emerge che un giocatore ha già a disposizione un'alternativa migliore di quella che si sta ancora valutando in un certo ramo, approfondire ulteriormente quel ramo diventa superfluo: il risultato della ricerca completa, con o senza quella porzione dell'albero, sarebbe comunque identico [9]. A differenza di molte altre tecniche nella storia dei motori scacchistici, la potatura alpha-beta non ha un singolo autore riconosciuto: venne scoperta indipendentemente da più ricercatori tra la fine degli anni Cinquanta e l'inizio degli anni Sessanta, tra cui il matematico sovietico Alexander Brudno e un gruppo di ricercatori statunitensi [7].
Nel 1975 Knuth e Moore dimostrarono formalmente la correttezza della tecnica, cioè che la potatura non altera mai il risultato che si otterrebbe con una ricerca minimax completa, e ne analizzarono l'efficienza nei casi tipici [9]. Da allora, la potatura alpha-beta è rimasta per decenni la base della ricerca nei motori scacchistici, Stockfish compreso. I motori moderni la affiancano però a potature e riduzioni selettive, elencate nella Sezione 2.3.4, che a differenza dell'alpha-beta non garantiscono più lo stesso risultato della ricerca completa [7].

2.3.3 Deep Blue e la sconfitta di Kasparov (1997)

Deep Blue è il sistema con cui un computer ha battuto in un match un campione del mondo in carica, Garry Kasparov, nel 1997. Il progetto nacque negli anni Ottanta alla Carnegie Mellon University con il nome di ChipTest e proseguì dal 1989 presso l'IBM, dove una parte del gruppo si trasferì [10]. La sezione descrive i principi teorici su cui il sistema poggiava: l'organizzazione della ricerca, le estensioni selettive e la funzione di valutazione. Le informazioni tecniche provengono soprattutto dall'articolo che i progettisti hanno pubblicato nel 2002 [10]. Secondo Campbell, Hoane e Hsu, la

versione del 1997 era un calcolatore IBM RS/6000 SP a 30 nodi, ciascuno dotato di 16 chip dedicati agli scacchi, per un totale di 480 chip. Ogni chip esaminava da 2 a 2,5 milioni di posizioni al secondo. Gli autori indicano per l'intero sistema una velocità tipica compresa tra i 100 e i 200 milioni di posizioni al secondo, a seconda del tipo di posizione, mentre la versione del 1996 raggiungeva tra i 50 e i 100 milioni [10]. La ricerca era organizzata su tre livelli. Un processore principale esplorava i livelli superiori dell'albero, i processori di lavoro esaminavano le posizioni foglia e i chip dedicati completavano gli ultimi livelli. La parte software, scritta in C e affidata ai processori generici, esplorava la zona vicina alla radice con estensioni ricorsive complesse e tabelle di trasposizione. La parte hardware eseguiva una ricerca alpha-beta a profondità fissa, con finestra nulla e ricerca di quiescenza, di norma su 4 o 5 semi-mosse. In ogni chip erano integrati il generatore di mosse, una pila delle mosse, la funzione di valutazione e il controllo della ricerca alpha-beta [10].
La funzione di valutazione non era un programma eseguito da un processore generico, ma era realizzata nei circuiti dei chip dedicati. Secondo gli autori il chip riconosce circa 8.000 pattern, a ciascuno dei quali è assegnato un valore, per un totale di 8.150 parametri tra 54 registri e 8.096 voci di tabella. Esistono due livelli. La valutazione veloce calcola il punteggio di una posizione in un solo ciclo di clock e contiene i termini principali facili da calcolare, in particolare il valore dei pezzi corretto in base alla casella occupata. La valutazione lenta scandisce la scacchiera una colonna alla volta e calcola concetti come il controllo delle case, le inchiodature, la sicurezza del re, la struttura pedonale, i pedoni passati, la torre in settima traversa e i pezzi intrappolati. Le caratteristiche di entrambi i livelli hanno pesi programmabili, che si possono modificare con facilità. Gli autori precisano che la gran parte delle caratteristiche e dei pesi fu creata o regolata a mano [10].
Alla ricerca si affiancavano conoscenze preparate in anticipo. Il libro delle aperture comprendeva 4.000 posizioni, e un secondo archivio ricavato da 700.000 partite assegnava bonus alle mosse. Il libro fu creato a mano, in prevalenza dal grande maestro Joel Benjamin con l'aiuto di altri grandi maestri. Per i finali erano disponibili basi di dati con tutte le posizioni di cinque pezzi o meno [10]. Il sistema affrontò Garry Kasparov in due incontri. Nel febbraio del 1996 Deep Blue perse il match per 4 a 2. Nel maggio del 1997, a New York, la versione potenziata vinse il rematch per 3,5 a 2,5 [10]. Secondo Chess.com, nel match del 1996, a Filadelfia, Deep Blue aveva vinto la prima partita, diventando il primo computer a battere un campione del mondo in una partita di livello da torneo [8].

2.3.4 L'era open source: dalla nascita di Stockfish ai suoi concorrenti

Prima di Stockfish esistevano diversi motori di cui il codice sorgente era pubblico. Crafty fu scritto da Robert Hyatt in ANSI C a partire dai primi anni Novanta e deriva in parte da Cray Blitz, vincitore dei campionati mondiali di scacchi per computer del 1983 e del 1986. Fu tra i primi a usare le rotated bitboards (una tecnica per

generare le mosse dei pezzi che si muovono per più caselle, come la donna, la torre e l'alfiere) e la consultazione delle tablebase di Nalimov (un database di posizioni di finale di cui è noto in anticipo il risultato), e fu oggetto di diversi studi sulla programmazione scacchistica [7]. Fruit, di Fabien Letouzey, comparve nel 2004 e al campionato mondiale del 2005 si classificò secondo, dietro a Zappa. La Chess Programming Wiki riferisce che Vasik Rajlich, autore di Rybka, ha dichiarato di aver preso molti elementi dal codice di Fruit 2.1 e che diversi motori hanno compiuto un forte progresso dopo la sua pubblicazione. La stessa fonte attribuisce a Fruit la diffusione delle late move reductions (una tecnica di programmazione per rendere la ricerca nell'albero delle mosse molto più efficiente) e presenta come una sua innovazione la valutazione tapered (una tecnica che gestisce il punteggio di una posizione in base allo stato della partita, quindi una certa mossa fatta nel mediogioco ha un peso diverso se è fatta in un finale), precisando però che non era del tutto nuova [7].
Glaurung, di Tord Romstad, fu rilasciato per la prima volta nell'autunno del 2004. La versione 2 fu riscritta usando le bitboards come rappresentazione principale della scacchiera. Usava la ricerca Principal Variation Search con selettività avanzata, e contribuì insieme a Fruit alla diffusione delle late move reductions. Secondo la Chess Programming Wiki, il fatto che fosse open source rendeva le sue tecniche accessibili alla comunità degli sviluppatori, e la stessa fonte lo indica come il precursore di Stockfish [7]. Nel novembre del 2008 Marco Costalba derivò Stockfish dalla versione 2.1 di Glaurung e pubblicò la versione 1.0 [2]. La Chess Programming Wiki elenca come sviluppatori Romstad, Costalba, Joona Kiiski e Gary Linscott, e spiega che il nome richiama il commercio storico dello stoccafisso dalla Norvegia all'Italia, paesi d'origine rispettivamente di Romstad e di Costalba [7].
Il sito ufficiale dichiara che la licenza di Stockfish è gratuita e distribuita sotto GNU General Public License, versione 3. Chi distribuisce Stockfish deve includere il codice sorgente completo oppure un riferimento a esso, ma essendo completamente open source, il progetto è accessibile a chiunque [2]. Personalmente credo fermamente che sia stata proprio questa libertà a rendere Stockfish uno dei migliori motori scacchistici al mondo.
Nel corso degli anni lo sviluppo di Stockfish si è spostato dai primi autori a una comunità di volontari. Secondo la Chess Programming Wiki, Costalba e Kiiski lasciarono la guida del progetto nel 2011 [7], e oggi, come indica il sito ufficiale, il motore è sviluppato e mantenuto dalla comunità di Stockfish [2]. Uno strumento centrale di questo modello è Fishtest, un sistema di test distribuito creato da Linscott all'inizio del 2013. I volontari mettono a disposizione il tempo di calcolo dei propri computer, e ogni modifica al codice viene valutata con molte partite, usando il sequential probability ratio test, un test che esamina i risultati man mano che arrivano le partite e si ferma quando il risultato è statisticamente significativo [11]. La Chess Programming Wiki riporta, a luglio del 2024, 2.226 collaboratori che mettono a disposizione i propri computer e 363 sviluppatori [7].
Per quanto riguarda la ricerca, la Chess Programming Wiki elenca tra le tecniche im-

piegate l'approfondimento iterativo con finestre di aspirazione, la Principal Variation Search, la ricerca parallela Lazy SMP (dal gennaio 2016), la potatura con mossa nulla con riduzione dinamica della profondità, le late move reductions, le internal iterative reductions, il razoring, la ricerca di quiescenza, ProbCut e la potatura basata su SEE (Static Exchange Evaluation: serve per scartare mosse che quasi certamente perdono materiale) [7]. Per la valutazione, le versioni precedenti usavano una valutazione tapered basata su materiale, tabelle pezzo-casella, struttura pedonale e sicurezza del re [7]. La versione 12, uscita il 2 settembre 2020, ha introdotto la rete neurale NNUE [2]. La valutazione classica è stata poi rimossa: la Chess Programming Wiki lo collega alla versione 16 del 30 giugno 2023 [7], mentre la documentazione ufficiale indica l'agosto del 2023 [12]. Per questo la Sezione 3.5.1 adotta Stockfish 16.1 come prima versione senza valutazione classica. L'argomento è trattato nella sezione dedicata alle reti neurali.
Stockfish comunica con le interfacce grafiche attraverso il protocollo UCI (Universal Chess Interface), ideato da Rudolf Huber e Stefan Meyer-Kahlen e rilasciato nel novembre del 2000. Secondo la Chess Programming Wiki, è un protocollo aperto che permette ai motori di giocare partite in modo automatico e di dialogare con altri programmi. Grazie al protocollo UCI il modello della partita e il suo stato sono gestiti dall'interfaccia grafica, mentre il motore resta più semplice [7]. Il progetto descritto nel Capitolo 4 usa questo protocollo per comunicare con Stockfish. Quanto ai risultati, secondo la Chess Programming Wiki Stockfish ha raggiunto la superfinale del TCEC dalla quarta stagione e ha superato i concorrenti commerciali Komodo e Houdini [7]. Il sito ufficiale afferma che il motore è usato con fiducia da grandi maestri e dalle principali piattaforme di scacchi, e si tratta di una dichiarazione del progetto stesso [2].

2.4 L'avvento del machine learning negli scacchi

Le sezioni precedenti hanno mostrato che i motori scacchistici si sono basati per decenni su una struttura comune, formata da una funzione di valutazione e da una ricerca nell'albero delle mosse. Nel 2017 un articolo di David Silver e collaboratori descrive questa tradizione in questi termini:

> *"handcrafted evaluation functions that have been refined by human experts over several decades"*
>
> Silver e altri [13]

L'abstract osserva che i programmi più forti dell'epoca combinavano tecniche di ricerca sofisticate, adattamenti specifici al dominio e funzioni di valutazione scritte a mano. Gli autori usano questa descrizione come punto di partenza per proporre un metodo diverso, nel quale il programma impara a giocare senza conoscenze degli scacchi oltre alle regole [13]. Il metodo nasce nel gioco del Go. Silver e collaboratori

presentano AlphaGo, un programma che usa value networks per valutare le posizioni e policy networks per scegliere le mosse. Le reti sono addestrate con l'apprendimento supervisionato su partite di esperti umani e con l'apprendimento per rinforzo da partite giocate contro se stesse. Gli autori riportano una percentuale di vittorie del 99,8 % contro altri programmi di Go e una vittoria per 5 a 0 contro il campione europeo [14].
Nel 2017 AlphaGo Zero viene descritto come un algoritmo basato soltanto sull'apprendimento per rinforzo, senza dati umani e senza conoscenze oltre alle regole del gioco. Gli autori indicano una vittoria per 100 a 0 contro la versione di AlphaGo che aveva battuto un campione del mondo, ottenuta con circa tre giorni di addestramento e cinque milioni di partite contro se stesso [15]. L'articolo di Silver e collaboratori sugli scacchi generalizza questo approccio. Nell'abstract gli autori scrivono di averlo esteso in un unico algoritmo, AlphaZero, che parte da gioco casuale e, senza conoscenze oltre alle regole, raggiunge entro 24 ore un livello superumano negli scacchi, nello shogi e nel Go [13]. Le sezioni che seguono descrivono AlphaZero e l'apprendimento per rinforzo, Leela Chess Zero e la rete neurale NNUE adottata da Stockfish.

2.4.1 AlphaZero e l'apprendimento per rinforzo (2017)

L'articolo Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm fu reso disponibile nel dicembre del 2017 su arXiv, un archivio online ad accesso aperto, da David Silver e altri ricercatori, tra cui Demis Hassabis [13]. Una versione successiva fu pubblicata nel dicembre del 2018 sulla rivista Science [16]. Secondo l'abstract, l'obiettivo è estendere a più giochi l'approccio di AlphaGo Zero (una intelligenza artificiale specializzata nel gioco del Go), fondato sull'apprendimento per rinforzo tabula rasa da partite giocate contro se stessi. Il programma parte da gioco casuale e non riceve conoscenze del dominio oltre alle regole del gioco [13].
Il sistema si basa su una rete neurale che riceve in ingresso la posizione e produce due uscite: un vettore di probabilità per le mosse e un valore scalare. Per gli scacchi la posizione è rappresentata con 119 piani spaziali, e la distribuzione sulle mosse ha la forma di un insieme 8x8x73, che corrisponde a 4.672 mosse possibili [13]. Il valore stima l'esito atteso della partita e tiene conto dei pareggi, a differenza di AlphaGo Zero, che distingueva solo vittoria e sconfitta [13].
La ricerca è di tipo Monte Carlo. Gli autori la descrivono come una serie di partite simulate che scendono nell'albero a partire dalla radice, scegliendo in ogni posizione una mossa con poche visite, alta probabilità e alto valore. Negli scacchi AlphaZero esamina circa 80.000 posizioni al secondo, mentre Stockfish ne esamina circa 70 milioni. Secondo gli autori, AlphaZero compensa il numero minore di valutazioni usando la rete neurale per concentrare la ricerca in modo molto più selettivo [13].
AlphaZero non usa conoscenze del dominio oltre alle regole. L'articolo elenca cinque elementi: la codifica delle regole, la conoscenza perfetta delle regole, la codifica dei piani di ingresso e di uscita, la scala del rumore di esplorazione in base al numero di

mosse e le regole di terminazione della partita [13].
A proposito della ricerca alpha-beta, descritta nella Sezione 2.3.2, gli autori scrivono che i risultati sul tempo di riflessione mettono in discussione un'idea diffusa:

> *"calling into question the widely held belief that alpha-beta search is inherently superior"*
>
> Silver e altri [13]

Nel passo gli autori riferiscono che la ricerca di AlphaZero migliora con il tempo di riflessione in modo più efficace di quella di Stockfish e di Elmo, i programmi usati come termine di confronto negli scacchi e nello shogi [13].
Per gli scacchi l'addestramento durò 9 ore e comprese 700.000 passi con mini-batch da 4.096 esempi, circa 44 milioni di partite giocate contro se stesso e 800 simulazioni per ogni mossa. Le partite furono generate con 5.000 TPU di prima generazione, mentre l'aggiornamento della rete usò 64 TPU di seconda generazione [13][7]. Gli autori indicano che il sistema superò Stockfish già dopo 4 ore, pari a 300.000 passi [13].
Nel confronto del dicembre 2017 furono giocate 100 partite con il tempo fisso di un minuto per mossa, e Stockfish disponeva di 64 thread e 1 GB di tabella hash. Il risultato fu di 28 vittorie, 72 patte e nessuna sconfitta per AlphaZero. Gli autori riportano inoltre che, durante l'addestramento, AlphaZero scoprì e usò spesso le 12 aperture umane più comuni, e che partendo da ciascuna di esse riuscì a battere Stockfish [13].
Nella versione pubblicata su Science nel 2018 la Chess Programming Wiki riferisce un confronto di 1.000 partite, di cui circa 200 pubblicate, in cui AlphaZero vinse 155 partite e ne perse 6. Gli avversari indicati sono Stockfish 8, una versione in sviluppo del 13 gennaio 2018 vicina a Stockfish 9, Brainfish con l'archivio delle aperture Cerebellum e Stockfish 9. Stockfish giocava con 44 thread su 44 core, una tabella di trasposizione da 32 GiB, le basi di dati Syzygy a 6 pezzi e un tempo di 3 ore per giocatore e partita più 15 secondi di incremento per mossa. Ogni ricerca di AlphaZero era eseguita su una macchina con 4 TPU di prima generazione [7].
Le condizioni del confronto del 2017 (tempo fisso di un minuto per mossa e 1 GB di tabella hash per Stockfish, secondo [13]) furono oggetto di critiche. La Chess Programming Wiki le riassume parlando di un possibile vantaggio hardware di AlphaZero e di condizioni di gioco criticate, senza dettagliare le singole obiezioni [7].

2.4.2 NNUE: l'ibridazione tra ricerca classica e reti neurali in Stockfish (2018-2020)

NNUE è l'acronimo di Efficiently Updatable Neural Network, cioè rete neurale aggiornabile in modo efficiente. La tecnica non nasce negli scacchi ma nello shogi, il gioco da tavolo giapponese, e fu introdotta nel 2018 da Yu Nasu, programmatore di shogi, in un documento intitolato NNUE Efficiently Updatable Neural-Network

based Evaluation Functions for Computer Shogi. Il testo fu diffuso dallo Ziosoft Computer Shogi Club ed è scritto in giapponese, con un abstract in inglese [7].
L'idea di partenza si ispira alle tabelle piece square usate da Kunihito Hoki nel motore di shogi Bonanza, in cui il valore di un pezzo dipende dalla casella che occupa e dalla posizione del re, oltre che da coppie di pezzi e dal lato che muove [7]. La documentazione ufficiale di Stockfish racconta che NNUE fu inventata per lo shogi da Yu Nasu, che nel maggio del 2018 Motohiro Isozaki la integrò nel motore di shogi YaneuraOu e che l'adattamento agli scacchi per Stockfish si deve a Hisayori Noda, noto con il nome di nodchip, e risale al giugno del 2019 [12].
La caratteristica che dà il nome a NNUE è l'aggiornamento efficiente della rete. La documentazione ufficiale di Stockfish indica come requisito che gli ingressi cambino il meno possibile da una valutazione alla successiva. Il requisito è formulato in questo modo:

> *"The inputs should change as little as possible between subsequent evaluations."*
>
> Autori di Stockfish [12]

Una mossa modifica soltanto poche features della posizione. Per questo la rete non ricalcola da capo il primo strato a ogni valutazione: ne conserva il risultato, chiamato accumulatore, e lo aggiorna quando la posizione cambia, sottraendo i contributi delle features che scompaiono e sommando quelli delle features che compaiono [12][7]. Le reti sono volutamente poco profonde e lavorano con numeri interi invece che in virgola mobile. In questo modo la valutazione può essere eseguita direttamente dal processore, senza acceleratori [12].
Il post con cui il sito ufficiale di Stockfish presentò la novità spiega che NNUE fu sviluppata per lo shogi da nodchip, ynasu87 e yaneurao e poi portata in Stockfish. Il codice entrò nel repository il 6 agosto 2020, ma restava disattivato in modo predefinito: per usarlo bisognava attivarlo con un'opzione specifica del protocollo UCI. Lo stesso post contrappone la nuova valutazione a quella classica, basata su concetti scacchistici scritti a mano (handcrafted). La rete usa invece ingressi elementari e sfrutta il fatto che, dopo una mossa tipica, solo una parte di essa va aggiornata. Le reti furono ottimizzate e addestrate sulle valutazioni di milioni di posizioni a profondità di ricerca moderata. Nei test riportati dal post il guadagno fu di oltre 80 Elo [2]. Secondo la Chess Programming Wiki, la versione con NNUE dimezzava circa la velocità di ricerca ma risultava comunque più forte dell'originale [7].
NNUE entrò nella release ufficiale Stockfish 12 del 2 settembre 2020. La valutazione con rete neurale e quella classica restavano entrambe disponibili, e la rete consigliata era incorporata negli eseguibili e usata in modo predefinito. Il post aggiunge che Stockfish resta un motore che lavora solo sulla CPU, perché le reti NNUE si valutano in modo molto efficiente sul processore, e che contro la versione 11 vince tipicamente almeno dieci volte più coppie di partite di quante ne perda [2]. La documentazione

ufficiale indica come SFNNv1, detta anche architettura di Stockfish 12, la prima architettura di rete, in uso dal 6 agosto 2020 [12].
Dopo Stockfish 12 l'architettura della rete è stata modificata più volte. La documentazione ufficiale elenca le architetture da SFNNv1 in avanti, ciascuna con il periodo in cui è stata in uso. Tra SFNNv6 (maggio 2023) e SFNNv9 (aprile 2024) la dimensione del primo strato, indicata con L1, passa da 1024 a 1536, 2048, 2560 e infine 3072 neuroni. SFNNv10 (novembre 2025) aggiunge nuove caratteristiche di ingresso chiamate FullThreats e riporta L1 a 1024 [12]. Il post di Stockfish 16 presenta la rete SFNNv6 e riporta per la release un guadagno di forza fino a 50 Elo contro Stockfish 15 [2]. Per l'addestramento il progetto ufficiale usa NNUE-Pytorch, un'implementazione in PyTorch che lavora nello spazio vittoria, patta, sconfitta e converte le valutazioni espresse in centipawn in probabilità tramite una funzione sigmoide [12].
Il file README del repository di Stockfish afferma che il motore usa reti neurali addestrate su dati forniti dal progetto Leela Chess Zero [11]. La Chess Programming Wiki riporta inoltre che la tecnica NNUE si è diffusa ad altri motori scacchistici, tra cui Igel, Halogen, Seer e Komodo Dragon [7].

2.5 Sintesi del capitolo

Il capitolo ha seguito il rapporto tra scacchi e intelligenza artificiale dalle origini teoriche fino ai motori attuali. Le prime riflessioni risalgono agli anni Quaranta e Cinquanta, quando Turing e Champernowne elaborarono Turochamp [3] e Shannon descrisse una funzione di valutazione abbinata all'algoritmo minimax [1]. Il nome stesso di intelligenza artificiale compare nella proposta per il workshop di Dartmouth [6]. Per alcuni decenni la strada dominante fu la ricerca ad albero, resa efficiente dalla potatura alpha-beta, di cui Knuth e Moore dimostrarono la correttezza [9]. Deep Blue ne è l'esito più noto: un sistema con chip dedicati alla ricerca e alla valutazione che nel 1997 sconfisse il campione del mondo Garry Kasparov [10].
Parallelamente nacquero motori open source come Crafty, Fruit e Glaurung [7]. Dal codice di Glaurung 2.1 Marco Costalba ricavò nel novembre del 2008 la prima versione di Stockfish [2]. La svolta successiva è l'apprendimento automatico. AlphaZero, descritto nel 2017, impara a giocare solo attraverso partite contro se stesso e nel match riportato dagli autori vinse 28 partite e ne pareggiò 72 contro Stockfish 8, senza perderne nessuna. Gli autori presentano il risultato come una messa in discussione della convinzione diffusa che la ricerca alpha-beta sia superiore [13].
Nel 2020 Stockfish adottò NNUE, una rete neurale introdotta da Yu Nasu nello shogi nel 2018 [7]. Stockfish 12, uscito il 2 settembre 2020, la include accanto alla valutazione classica, che resta disponibile [2]. La rete si aggiorna in modo efficiente quando la posizione cambia e viene valutata dal processore, senza acceleratori [12].
Il percorso del capitolo arriva così a Stockfish, il motore che il progetto descritto nei

capitoli successivi usa come motore esterno, interrogato tramite il protocollo UCI.

Capitolo 3

Stockfish 18: architettura e funzionamento

3.1 Introduzione a Stockfish 18

Stockfish 18 è stato rilasciato il 31 gennaio 2026 ed è presentato dal sito ufficiale come una nuova versione maggiore. Nei test contro Stockfish 17 il post di rilascio riporta un guadagno di forza fino a 46 punti Elo e quattro volte più coppie di partite vinte che perse [2].
Le novità principali indicate dal post riguardano quattro aspetti: l'architettura di rete SFNNv10, il cui livello di ingresso è arricchito da caratteristiche sulle minacce (Threat Inputs); una nuova implementazione di memoria condivisa, che permette a processi diversi di Stockfish di condividere i pesi della rete; l'uso della Correction History nella ricerca, che corregge le valutazioni in base a ciò che emerge durante la ricerca stessa; un addestramento delle reti diventato automatizzato e riproducibile. Il post aggiunge che sono stati fatti sforzi per sfruttare meglio l'hardware, adattando il codice alle istruzioni moderne della CPU e rivedendo l'interazione tra i thread durante la ricerca [2]. Questi aspetti sono ripresi nelle sezioni successive. La versione 18 segue Stockfish 17.1, uscito il 30 marzo 2025 con un guadagno dichiarato fino a 20 Elo rispetto al predecessore. Stockfish 17 era uscito il 6 settembre 2024, con un guadagno fino a 46 Elo rispetto alla versione 16 [2].
Stockfish 18 non è l'ultima versione. Il sito ufficiale indica come versione corrente Stockfish 19, rilasciato il 5 settembre 2026. Il post di quella versione riporta un guadagno fino a 44 Elo contro Stockfish 18, l'architettura di rete SFNNv16 e il ritiro della rete secondaria introdotta con Stockfish 16.1 [2]. Questo capitolo descrive la versione 18, ma Stockfish è un progetto in continua evoluzione, a cui contribuiscono sviluppatori e volontari [11]. Per dare un'idea dell'importanza di questi guadagni conviene ricordare che la forza viene misurata con il punteggio Elo, in cui il risultato atteso di una partita dipende dalla differenza di punteggio tra i due giocatori [7].
Nella lista del Computer Chess Rating Lists aggiornata al 2 ottobre 2026 Stockfish 19 è al primo posto con 3650 punti, su un computer con quattro processori [17]. Il punteggio più alto raggiunto da un giocatore umano nella classifica della FIDE è

invece il 2882 di Magnus Carlsen [18]. Le due scale sono calcolate separatamente, la prima sulle partite tra motori e la seconda sulle partite tra giocatori umani, perciò il confronto è indicativo.

3.2 Architettura generale del motore

Stockfish è un motore senza interfaccia grafica propria. Comunica con l'esterno attraverso il protocollo UCI, un protocollo testuale standard in cui l'interfaccia grafica invia i comandi sullo standard input del motore e ne riceve le risposte sullo standard output. I comandi principali sono "uci", che avvia la modalità UCI e fa elencare al motore le opzioni disponibili; "isready", che sincronizza interfaccia e motore; "position", che stabilisce la posizione a partire da quella iniziale o da una stringa FEN; "go", che avvia il calcolo con limiti di profondità, di nodi o di tempo; "stop", che ferma la ricerca e restituisce la migliore mossa trovata fino a quel momento; e "quit", che termina il processo. Stockfish inoltre aggiunge comandi propri tra cui "bench", che esegue un benchmark su posizioni prefissate, ed "eval", che mostra la valutazione statica della posizione corrente [12]. Il comportamento del motore si regola con le opzioni. Le più importanti per questo capitolo sono Threads, il numero di thread usati per la ricerca, con valore predefinito 1, e Hash, la dimensione in megabyte della tabella hash, con valore predefinito 16. Altre opzioni sono MultiPV, che fa produrre più linee migliori, SyzygyPath, la cartella con le tablebase di finale, NumaPolicy, che assegna i thread ai nodi di una macchina con più processori, ed EvalFile, il file della rete neurale da caricare [12].
La documentazione ufficiale definisce i termini della ricerca. L'approfondimento iterativo consiste nell'eseguire ricerche consecutive a profondità sempre maggiore. La tabella di trasposizione è un archivio che conserva i risultati di ricerche già eseguite. La ricerca di quiete è svolta alla fine della ricerca principale e valuta soltanto posizioni tranquille. L'ordinamento delle mosse serve a provare per prime le mosse migliori. La null move pruning si basa sull'idea che, se si raggiunge il valore beta anche senza muovere, è probabile che lo si raggiunga pure muovendo. Le late move reductions cercano a profondità ridotta le mosse da cui ci si aspetta un risultato peggiore [12]. Nel codice sorgente della versione 18 questi elementi compaiono nel file search.cpp, insieme ad altri: le finestre di aspirazione, che partono piccole e vengono allargate quando la ricerca fallisce, la ricerca a variante principale, la consultazione della tabella di trasposizione, le potature razoring, futility pruning e ProbCut, le estensioni singolari, le tabelle di storia delle mosse, la correction history e la gestione del tempo [11].
Nella funzione search ogni nodo è cercato con due limiti, alpha e beta. Alpha è il miglior valore che il giocatore di turno può già ottenere con le varianti viste finora, mentre beta è il valore oltre il quale l'avversario eviterebbe la posizione. Il ciclo sulle mosse si interrompe appena una mossa dà un valore pari o superiore a beta, ed è il

taglio beta, mentre alpha viene aggiornato quando una mossa lo supera. Per cercare la posizione successiva la funzione richiama se stessa con i limiti scambiati e cambiati di segno, cioè con $-\beta$ e $-\alpha$ [11]. La Figura 3.1 mostra il meccanismo su un albero semplificato.

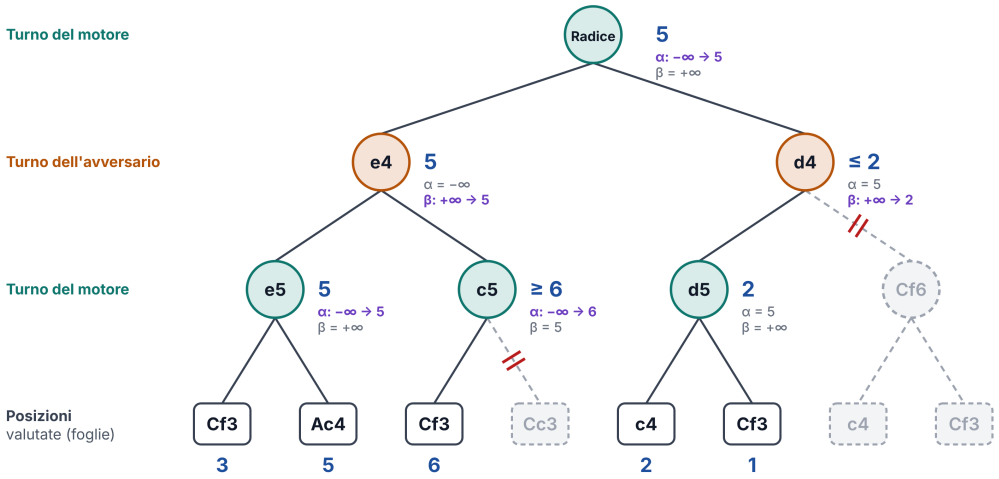


Figura 3.1: Albero di ricerca semplificato con i limiti alpha e beta. Illustrazione dell'autore.

La ricerca non valuta le posizioni da sola. Nel codice la valutazione è affidata alla rete neurale: il file evaluate.cpp calcola il valore di una posizione con una rete grande e una piccola, mentre la cartella nnue contiene l'accumulatore e la definizione dell'architettura [11]. La documentazione ufficiale descrive NNUE come la funzione che valuta le posizioni con una rete neurale addestrata su grandi quantità di dati [12]. La ricerca usa questa valutazione statica, per esempio per saltare del tutto la ricerca di un nodo quando la valutazione è molto bassa, e la corregge con la correction history [11]. La Figura 3.2 mostra il diagramma di flusso semplificato di Stockfish 18.

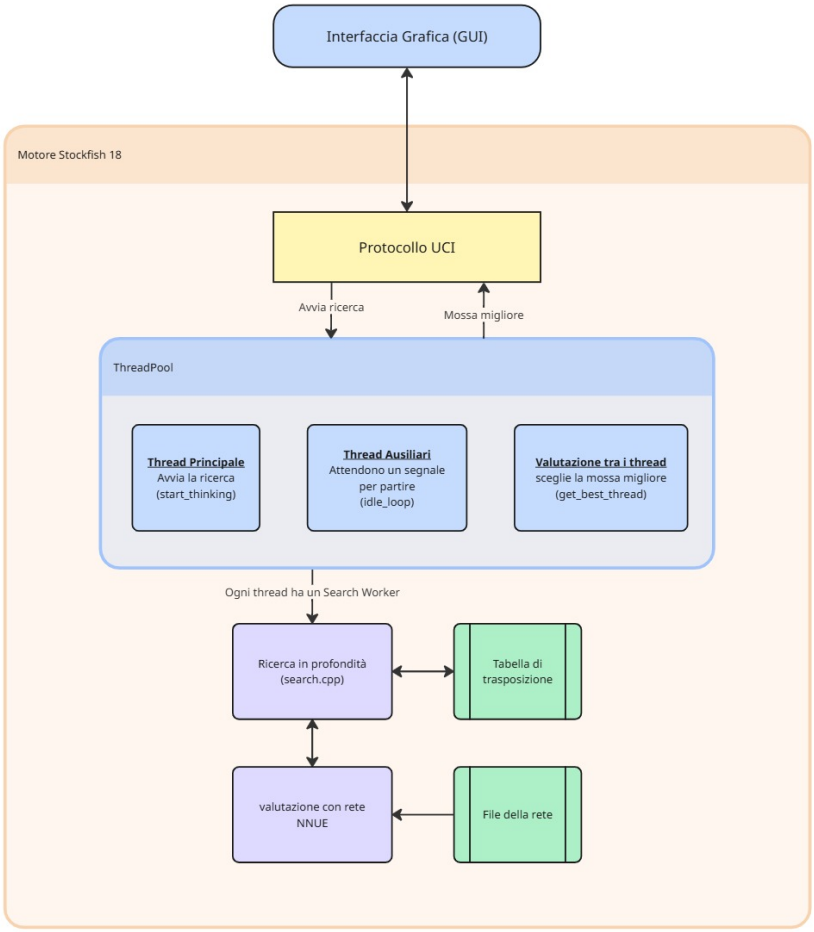


Figura 3.2: Diagramma di flusso di Stockfish 18. Illustrazione dell'autore.

3.3 Rappresentazione della scacchiera

La FEN, Forsyth-Edwards Notation, è il formato testuale con cui un'interfaccia grafica comunica a Stockfish una posizione qualsiasi. Deriva da uno standard dell'Ottocento ideato dal giornalista scozzese David Forsyth, che Steven Edwards ha esteso per l'uso con i programmi di scacchi e inserito nella specifica della Portable Game Notation [19, 7]. Una stringa FEN usa soltanto caratteri ASCII e contiene sei campi separati da uno spazio [19, 11]:

1. la disposizione dei pezzi, dalla traversa 8 alla traversa 1 e, in ogni traversa, dalla colonna a alla colonna h; i pezzi bianchi sono lettere maiuscole, quelli neri minuscole, le case vuote consecutive sono indicate da una cifra e le traverse sono separate da una barra;

2. il colore che deve muovere, w per il bianco e b per il nero;
3. i diritti di arrocco, indicati con K e Q (King side e Queen side) per il bianco e con k e q per il nero, dove un trattino segnala che nessuno può arroccare;
4. la casa di presa en passant, oppure un trattino se non esiste;
5. il contatore delle semimosse trascorse dall'ultima mossa di pedone o dall'ultima cattura, che serve per la regola delle cinquanta mosse;
6. il numero della mossa completa, che parte da 1 e aumenta dopo la mossa del nero.

La posizione iniziale corrisponde alla stringa

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

Nel codice è una costante, usata quando il comando position riceve la parola "startpos". Le mosse che seguono la posizione nello stesso comando vengono applicate una alla volta, e ogni stato intermedio viene conservato [11]. Per riconoscere una posizione già incontrata senza confrontare l'intera scacchiera, i motori associano a ogni posizione un numero, detto chiave. Il metodo più diffuso è l'hashing di Zobrist, proposto da Albert Zobrist nel 1970 nel rapporto tecnico A New Hashing Method with Application for Game Playing e ripubblicato nel 1990 sull'ICCA Journal [20, 7]. Il metodo parte da numeri pseudocasuali generati una sola volta all'avvio del programma: uno per ogni pezzo su ogni casa, più altri per il lato che deve muovere, per i diritti di arrocco e per la colonna della presa en passant. La chiave di una posizione si ottiene combinando con l'operazione XOR i numeri che corrispondono agli elementi presenti sulla scacchiera. Poiché lo XOR è l'inverso di se stesso, applicare due volte lo stesso numero lo annulla. La chiave quindi non si ricalcola da zero a ogni mossa, ma si aggiorna [7].
In Stockfish la chiave è un intero senza segno a 64 bit. I numeri casuali vengono preparati nella funzione "Position::init" da un generatore di tipo xorshift64star, inizializzato con un seme fisso. La tabella di Zobrist contiene un numero per ogni pezzo su ogni casa, uno per ciascuna delle otto colonne della presa en passant, uno per ogni combinazione dei diritti di arrocco e uno per il lato che deve muovere. Per i pedoni bianchi i valori delle case dell'ottava traversa, e per i pedoni neri quelli della prima, sono posti a zero, perché il codice li annota come case in cui un pedone viene promosso [11]. La chiave è un riassunto della posizione, di dimensione fissa, e non la posizione stessa. Per questo due posizioni diverse possono ricevere la stessa chiave, un evento detto collisione. Con chiavi a 64 bit la collisione è rara: la Chess Programming Wiki stima che se ne possa attendere una dopo circa quattro miliardi di posizioni [7]. Non è però impossibile, e i suoi effetti sulla ricerca sono trattati nella sezione sulle tabelle di trasposizione.
La chiave viene calcolata in due momenti. Quando la posizione è letta da una FEN, la funzione "set_state" parte da zero e applica lo XOR del valore di ogni pezzo sulla

sua casa, della colonna en passant se presente, del lato quando muove il nero e dei diritti di arrocco. Dopo ogni mossa eseguita con la funzione "do_move", invece, la chiave si aggiorna a partire da quella precedente, con gli XOR del lato, del pezzo spostato sulla casa di partenza e su quella di arrivo, dell'eventuale pezzo catturato e delle variazioni di en passant e di arrocco [11]. Con lo stesso metodo Stockfish mantiene altre chiavi parziali: una per i soli pedoni, una per i pezzi minori, cioè cavalli e alfieri, una per i pezzi non pedoni di ciascun colore e una per il materiale. Il codice usa le prime tre come indice per la storia dei pedoni e per le tabelle di correzione della valutazione, tra le quali figura la correction history, già citata [11]. La chiave serve anche a riconoscere le ripetizioni. Dopo aver aggiornato la chiave, la funzione "do_move" cerca all'indietro tra gli stati conservati uno con la stessa chiave, a distanze pari di semimosse e fino al limite dato dal contatore delle cinquanta mosse o dall'ultima mossa nulla. Se lo trova, memorizza la distanza, con segno negativo quando la posizione si è già ripetuta due volte. La funzione "is_draw" considera patta la posizione quando il contatore supera 99 semimosse, a meno che il lato che deve muovere sia sotto scacco e non abbia mosse legali, oppure quando la ripetizione è ritenuta valida. Il commento del codice precisa che basta una ripetizione se la posizione precedente è successiva alla radice della ricerca, mentre ne servono due se la ripetizione avviene prima della radice o nella radice stessa [11]. La funzione key, che fornisce la chiave alla ricerca, non restituisce sempre il valore memorizzato. Quando il contatore delle semimosse raggiunge 14 o più, la chiave viene combinata con un valore calcolato da quel contatore, che cambia ogni otto semimosse. Questa chiave è quella con cui la ricerca interroga la tabella di trasposizione, descritta nella sezione seguente [11].

3.3.1 Le tabelle di trasposizione

La tabella di trasposizione conserva i risultati di ricerche già eseguite. Serve perché la stessa posizione può essere raggiunta con sequenze di mosse diverse, dette trasposizioni, e senza la tabella il motore la cercherebbe ogni volta da capo. Una tabella di questo tipo fu usata per la prima volta nel programma Mac Hack VI di Richard Greenblatt [7]. In Stockfish esiste una sola tabella globale, condivisa dal motore e da tutti i suoi thread. La ricerca la consulta all'inizio di ogni nodo con la chiave di Zobrist della posizione. Se trova una voce, ne ricava il valore, la profondità con cui è stato calcolato, il tipo di limite e la mossa migliore; alla fine del nodo scrive il risultato ottenuto. Nei nodi che non appartengono alla variante principale, se la profondità della voce è sufficiente e il suo limite è compatibile con il valore beta, la ricerca può usare il valore memorizzato senza esplorare il nodo [11]. La mossa memorizzata serve invece per l'ordinamento delle mosse [7].
La Chess Programming Wiki indica come contenuto tipico di una voce la chiave di verifica, la profondità, il valore con il tipo di limite (esatto, inferiore o superiore), la mossa migliore e l'età (da quanto tempo la voce è nella tabella, conta quante

ricerche sono "passate"), usata per decidere quali voci sovrascrivere [7]. In Stockfish 18 una voce occupa 10 byte e contiene 16 bit della chiave, 8 bit di profondità, 5 bit di generazione, 1 bit che segnala un nodo della variante principale, 2 bit per il tipo di limite, 16 bit per la mossa, 16 per il valore e 16 per la valutazione statica della posizione. La profondità è salvata con uno scostamento, perché il codice deve poter memorizzare anche le profondità negative della ricerca di quiete, e il valore zero della profondità indica una voce vuota [11].
Le voci sono raggruppate in cluster di tre, con due byte di riempimento, per una dimensione totale di 32 byte. Il commento del codice spiega che la dimensione del cluster dovrebbe dividere quella di una linea di cache, perché la linea viene caricata in anticipo quando possibile. Il cluster si individua moltiplicando la chiave per il numero di cluster e prendendo i 64 bit alti del prodotto, mentre all'interno del cluster la voce giusta si riconosce confrontando i 16 bit bassi della chiave [11]. La Chess Programming Wiki descrive invece come soluzione comune l'uso dei bit bassi della chiave mascherati, quando la dimensione della tabella è una potenza di due [7]. La stessa fonte distingue due tipi di collisione: quelle di chiave, in cui posizioni diverse hanno la stessa chiave, e quelle di indice, in cui chiavi diverse puntano alla stessa posizione della tabella. Le collisioni di indice si controllano memorizzando nella voce una parte della chiave, da confrontare durante la consultazione [7]. Il commento in testa a tt.h riconosce che nel codice le collisioni sono possibili e possono provocare errori di gioco, come mosse sbagliate o segnalazioni errate di matto, che ridurli costerebbe punti Elo e che il rischio diminuisce rapidamente con tabelle più grandi [11].
Nella funzione "do_move", appena la nuova chiave è calcolata, il codice richiede alla memoria di caricare in anticipo il cluster corrispondente (prefetch). Il commento di "first_entry", la funzione che calcola l'indice, precisa che il suo unico uso esterno è proprio il prefetch [11]. La dimensione della tabella si imposta con l'opzione Hash, espressa in megabyte. Il codice ricava il numero di cluster dividendo i byte richiesti per la dimensione di un cluster, alloca la memoria con la funzione "aligned_large_pages_alloc" e poi azzera la tabella, lasciando che ogni thread azzeri una propria porzione. La funzione "hashfull" stima l'occupazione in millesimi, contando le voci della generazione corrente nei primi mille cluster [11]. Quando la posizione non è presente, la funzione "probe" sceglie la voce del cluster da sostituire. Il commento del codice definisce il valore di sostituzione di una voce come la sua profondità meno otto volte la sua età relativa, e sostituisce quella con il valore più basso. La funzione "save", invece, sovrascrive una voce se il nuovo limite è esatto, se la chiave è diversa, se la nuova profondità non è troppo inferiore a quella memorizzata, con un margine favorevole ai nodi della variante principale, oppure se la voce appartiene a una ricerca precedente. Se non c'è una mossa nuova, la mossa già memorizzata viene conservata. L'età è misurata con un numero di generazione, che la funzione "new_search" aumenta all'inizio di ogni ricerca dalla radice [11].
La Chess Programming Wiki elenca tra gli schemi di sostituzione la sostituzione

immediata, la preferenza per la profondità maggiore, il sistema a due livelli attribuito a Ken Thompson e Joe Condon, i bucket e l'invecchiamento [7]:

- Sostituzione immediata: quando una nuova voce finisce in una posizione della tabella già occupata, quella vecchia viene cancellata subito e la nuova prende il suo posto. È lo schema più semplice e dà peso solo alla novità.
- Preferenza per la profondità maggiore: la nuova voce prende il posto di quella vecchia solo se è stata calcolata con una profondità maggiore. Questo schema dà peso al lavoro svolto, perché protegge i risultati ottenuti guardando più mosse in avanti.
- Sistema a due livelli, attribuito a Ken Thompson e Joe Condon: per ogni posizione della tabella ci sono due voci, una che segue la regola della profondità maggiore e una che segue la sostituzione immediata. Unisce quindi i due schemi precedenti.
- Bucket: è un'estensione del sistema a due livelli. Invece di due voci, ogni posizione della tabella ne contiene un gruppo, chiamato bucket, il cui numero di solito dipende dalla dimensione di una linea di cache, cioè il blocco di memoria che il processore legge in un solo accesso. La nuova voce sostituisce quella del bucket con la profondità più bassa.
- Invecchiamento: ogni voce ricorda quanto è vecchia, così il motore evita di conservare voci di ricerche passate che dalla posizione di partenza attuale potrebbero non comparire più. Una voce vecchia può essere sostituita anche se la sua profondità la proteggerebbe.

Sulla condivisione tra i thread la Chess Programming Wiki afferma che una tabella globale, usata da più thread o processi, è essenziale per una ricerca parallela efficace sulle CPU multicore moderne [7]. Nel codice gli aggiornamenti tra thread non sono sincronizzati: il commento in testa a tt.h dichiara che le scritture concorrenti sono permesse di proposito, perché sincronizzare l'accesso costerebbe tempo di calcolo e quindi punti Elo. Lo stesso commento avverte che i dati copiati dalla funzione "probe" possono risultare incoerenti per le corse in lettura. Nella ricerca, un commento segnala che un valore non valido può comparire quando la voce non è stata trovata o per una corsa nell'accesso [11].

3.4 La ricerca in parallelo e i thread multipli

Per scegliere una mossa un motore esplora le posizioni che possono nascere da quella attuale. La posizione di partenza, quella di cui si vuole trovare la mossa migliore, si chiama radice, e le posizioni incontrate durante l'esplorazione sono i nodi dell'albero di ricerca. Questo lavoro può essere distribuito tra più thread. Un thread è la più piccola unità di elaborazione che un sistema operativo può pianificare, e i

thread di uno stesso programma possono condividere dei dati globali [7]. Stockfish può cercare la mossa migliore con più thread contemporanei. Il numero si sceglie con l'opzione "Threads", che per impostazione predefinita vale 1. La documentazione ufficiale definisce i thread come le unità di calcolo che il motore userà, chiamate anche core, e osserva che di solito più thread danno risultati migliori [12].
Il valore massimo dell'opzione Threads è calcolato a partire dal numero di processori del computer, che il codice ricava dal sistema. Quando l'opzione cambia, il motore ricrea i thread e rialloca la tabella di trasposizione [11]. La tabella di trasposizione è un database che conserva i risultati di ricerche già svolte [7]. Il metodo con cui i thread collaborano si chiama "Lazy SMP": tutti eseguono la funzione di ricerca e condividono la tabella di trasposizione [12]. Nella terminologia della Chess Programming Wiki, SMP search indica una ricerca parallela con una tabella hash condivisa [7]. Le sezioni seguenti descrivono questo metodo e la sua realizzazione nel codice.

3.4.1 Lazy SMP: l'idea e la sua storia

Lazy SMP è il metodo di ricerca parallela usato da Stockfish. L'idea è far eseguire la stessa funzione di ricerca a più thread, che si scambiano i risultati attraverso la tabella di trasposizione condivisa [12]. Ogni voce della tabella indica anche con quale profondità, cioè a quanti livelli di mosse in avanti, la posizione è stata esaminata [7]. In pratica, quando un thread esplora una posizione ne salva il risultato nella tabella di trasposizione. Se un secondo thread arriva alla stessa posizione, può riusare quel risultato invece di ricalcolarlo. Secondo la Chess Programming Wiki, le istanze della ricerca esaminano tutte la stessa posizione radice, ma partono con profondità diverse o provano le mosse alla radice in un ordine diverso. Il vantaggio nasce dal poter usare le voci della tabella scritte dalle altre istanze. L'ordine in cui il motore prova le mosse è importante, perché l'algoritmo alpha-beta funziona bene quando le mosse migliori sono cercate per prime [7]. La Figura 3.3 descrive il comportamento della ricerca multithread con tabella di trasposizione condivisa.

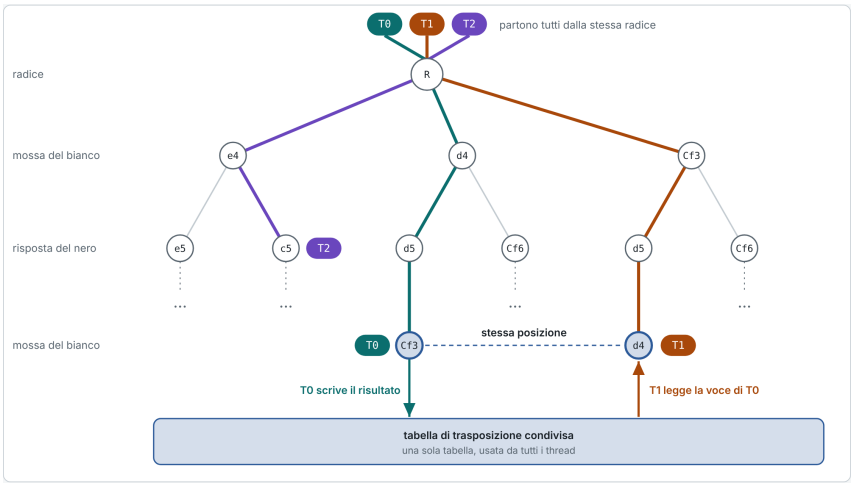


Figura 3.3: Diagramma del comportamento multithread in Lazy SMP. Illustrazione dell'autore.

Nel codice di Stockfish l'ordine delle mosse è deciso dalla classe "MovePicker". Il commento la descrive come lo strumento che restituisce una mossa alla volta, cercando di dare per prime quelle che hanno più probabilità di rendere inutile la ricerca delle altre. Il motore ordina le mosse con un punteggio calcolato da regole veloci, senza analizzarle a fondo. L'ordinamento avviene in quattro passi [11]:

1. Le mosse vengono generate senza essere cercate: il motore ricava l'elenco delle mosse possibili, senza esplorare le posizioni che ne derivano.
2. A ogni mossa viene assegnato un punteggio con regole semplici. Per le catture contano il valore del pezzo catturato e una tabella che ricorda come sono andate le catture precedenti. Per le mosse che non catturano si usano le tabelle di storia, che ricordano quanto spesso una mossa ha permesso di non cercare le altre (la sezione 3.4.3 le descrive meglio), con un bonus per le mosse che danno scacco e una penalità se il pezzo finisce su una casella minacciata da un pezzo di valore inferiore.
3. La mossa associata alla posizione nella tabella di trasposizione, se esiste, viene provata per prima.
4. Le mosse arrivano a fasi, non tutte insieme: dopo la mossa della tabella vengono le catture, poi le mosse che non catturano, che il motore genera e ordina solo a questo punto, e per ultime le catture considerate svantaggiose e le mosse con il punteggio più basso. Se la ricerca si interrompe prima, le mosse non ancora raggiunte, per esempio quelle che non catturano, non vengono nemmeno generate.

Il termine Lazy SMP è stato coniato da Julien Marcel alla fine del 2012. Secondo la Chess Programming Wiki, in termini di forza di gioco Lazy SMP scala sorprendentemente bene fino a 8 core e oltre, cioè la forza del motore continua a

crescere aggiungendo core [7]. Nel codice della versione 18 i thread ausiliari entrano direttamente nella stessa funzione di approfondimento iterativo usata dal thread principale [11]. L'approfondimento iterativo è il ciclo che alla radice aumenta via via la profondità della ricerca [12]. Il thread principale ha indice 0: oltre a cercare, tiene traccia del tempo e conserva i dati che riguardano solo lui [11].
Una differenza tra i thread compare nella finestra di aspirazione iniziale. Una finestra di aspirazione è un intervallo di punteggi attorno al valore che ci si aspetta, usato come limiti dell'algoritmo alpha-beta: se il punteggio vero cade fuori dalla finestra, bisogna ripetere la ricerca, con un costo elevato [7]. Nel codice l'ampiezza iniziale dipende anche dall'indice del thread, che in search.cpp viene usato soltanto qui per variare la ricerca, oltre che per riconoscere il thread principale [11]. I thread condividono anche un segnale, chiamato "increaseDepth", che il thread principale imposta in base al tempo trascorso: è vero se si sta facendo "ponder" oppure se il tempo trascorso non supera la metà di quello previsto. Ponder significa sfruttare il tempo di riflessione dell'avversario: il motore cerca la mossa successiva mentre l'avversario sta ancora pensando [7]. Quando il segnale è falso, ogni thread aumenta un contatore che riduce la profondità effettiva della ricerca successiva [11].

3.4.2 L'organizzazione dei thread nel codice

Il codice descrive i thread con due classi, definite in thread.h. La classe "Thread" rappresenta un singolo thread: contiene un puntatore all'oggetto che esegue la ricerca, chiamato Worker, e un thread nativo, cioè il thread vero e proprio che il sistema operativo pianifica. Dopo la creazione il thread nativo parte dalla funzione "idle_loop" e aspetta un segnale per iniziare a cercare. Quando la ricerca finisce, torna in attesa di un nuovo segnale. La classe "ThreadPool", cioè l'insieme dei thread, si occupa di tutto ciò che li riguarda, dalla creazione all'avvio della ricerca, e ogni accesso ai thread passa da questa classe [11].
La funzione "ThreadPool::set" distrugge i thread esistenti, se ce ne sono, e ne crea il numero richiesto con l'opzione Threads. I thread creati vanno subito in attesa in "idle_loop", e se serve vengono associati a un nodo NUMA [11]. La Chess Programming Wiki definisce NUMA (Non-uniform memory access) come un'organizzazione della memoria in cui la memoria principale è divisa tra i processori, e precisa che la memoria locale a un processore è più veloce di quella remota [7]. Un nodo NUMA è quindi una di queste porzioni di memoria insieme ai processori che la usano come memoria locale. Associare i thread a un nodo significa legarli a quel nodo [12]; la sezione 3.4.5 descrive come il motore decide se farlo.
Il primo thread del pool, con indice 0, è il thread principale. Solo a questo il codice assegna un gestore completo della ricerca, il "SearchManager", che secondo il commento gestisce la ricerca dal thread principale, tiene traccia del tempo e conserva i dati propri di quel thread. Agli altri thread assegna un gestore nullo, il "NullSearchManager", un oggetto che non fa nulla: il commento lo descrive come un

Null Object, che serve a dare a tutti i gestori un'interfaccia comune [11]. Quando l'interfaccia invia il comando go, la funzione "start_thinking" prepara la ricerca. Attende la fine della ricerca precedente, azzera i segnali di arresto, ricava le mosse legali della posizione e, per ogni thread, imposta i limiti e ricostruisce la posizione radice a partire dalla stringa FEN. Solo dopo questa preparazione viene svegliato il thread principale [11].
Il thread principale avvia gli altri thread e poi comincia a cercare anch'esso. Due modalità cambiano il momento in cui la ricerca si conclude: la modalità ponder, in cui il motore riflette durante il turno dell'avversario [7], e la modalità infinita, richiesta con go infinite, in cui la ricerca prosegue finché non arriva il comando stop [12]. In entrambi i casi il thread principale non stampa la mossa finché l'interfaccia non invia "stop" o "ponderhit" [11]. Il comando "ponderhit" segnala che l'avversario ha giocato la mossa prevista e fa passare il motore dalla riflessione a una ricerca normale [12]. Quando la ricerca termina, il thread principale ferma gli altri thread con il segnale di arresto, aspetta che finiscano e prepara la mossa da restituire. La scelta del thread da cui prenderla è descritta nella sezione 3.4.4 [11].

La Figura 3.4 descrive come i thread si attivano e come si comportano durante la ricerca.

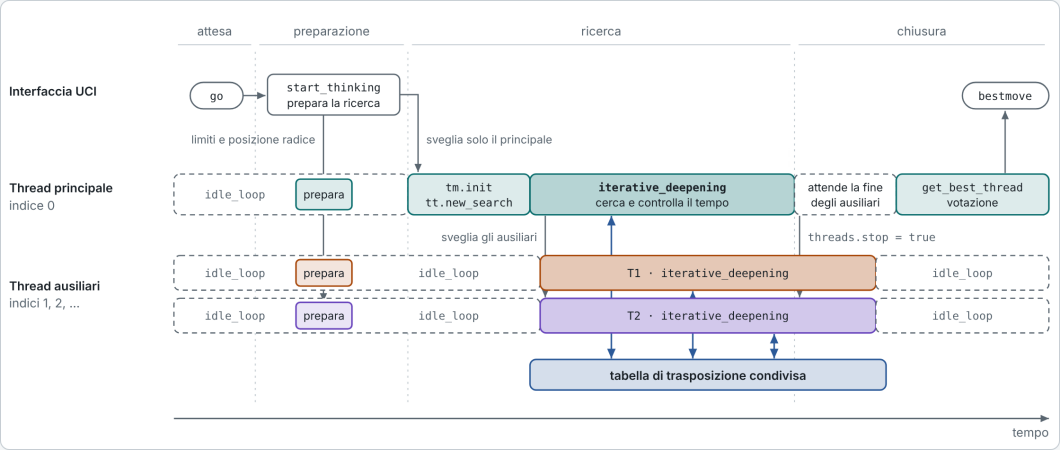


Figura 3.4: Diagramma temporale del comportamento dei thread durante una ricerca. Illustrazione dell'autore.

Per coordinarsi i thread usano due strumenti della libreria standard del C++, presenti in ogni oggetto Thread: un mutex e una variabile di condizione [11]. Un mutex può essere posseduto da un solo thread alla volta: chi chiama lock resta bloccato finché ottiene la proprietà e la mantiene fino a unlock, perciò gli altri thread devono aspettare. Una variabile di condizione permette a un thread di restare bloccato in attesa finché un altro lo risveglia, uno solo con "notify_one" o tutti insieme con "notify_all" [21].
Nella funzione "idle_loop" il thread segna di non essere in ricerca, avvisa chi attende

la fine della ricerca e si addormenta sulla variabile di condizione finché non riceve un compito. Quando riceve un compito lo esegue e poi torna in attesa. Il compito può essere l'avvio della ricerca oppure un'altra funzione, per esempio l'azzeramento di una parte della tabella di trasposizione; il commento chiama lancio l'esecuzione di una funzione in un thread. Durante la ricerca solo il thread principale controlla il tempo e alza il segnale di arresto quando il tempo è esaurito, mentre gli altri thread proseguono finché il segnale non è alzato [11].

3.4.3 La tabella di trasposizione condivisa e gli accessi senza blocchi

I dati condivisi arrivano a ogni thread attraverso una struttura chiamata "SharedState", che contiene le opzioni: il ThreadPool, la tabella di trasposizione, le storie condivise e le reti neurali. Ogni Worker ne conserva i riferimenti. Esiste quindi una sola tabella di trasposizione per tutti i thread. Una storia (history) è una tabella che registra per ogni mossa quanti tagli ha provocato, e serve a ordinare le mosse che non catturano. Un taglio è il caso in cui, trovata una mossa sufficiente, non occorre cercare le altre [7]. Altre tabelle di storia, per esempio la storia principale e quella delle catture, sono invece proprie di ogni thread [11].
Nella ricerca parallela i thread si scambiano i risultati attraverso la tabella di trasposizione, come descritto nella sezione precedente. Se più thread usano la stessa tabella, però, possono scrivere nella stessa voce nello stesso momento. La Chess Programming Wiki spiega che le voci di 16 byte o più non possono essere lette o scritte in modo atomico, cioè indivisibile: due scritture simultanee producono quindi dati corrotti. La soluzione classica indicata dalla stessa pagina è un lock per ogni voce, cioè un blocco che consente a un solo thread alla volta di accedervi. Si preferiscono però approcci senza blocchi, detti lockless [7].
In Stockfish la scrittura passa da un oggetto dedicato, "TTWriter", che il commento descrive come lo strumento usato per fare scritture non protette sulla tabella globale, cioè senza una protezione che impedisca gli accessi simultanei. La lettura restituisce invece una copia dei dati, in modo che gli oggetti locali restino separati da quelli globali. I commenti ammettono che l'aggiornamento di una voce non è atomico e può subire corse, cioè accessi simultanei di più thread, e che i dati copiati, pur restando definitivi, possono essere incoerenti al proprio interno. Le ragioni per cui il codice permette gli aggiornamenti non sincronizzati sono nella sezione 3.3.1. Lo stesso criterio compare nel conteggio degli accessi alle tablebase: il codice scrive il contatore con una lettura e una scrittura relaxed e il commento spiega che così si evitano istruzioni di blocco [11].
La storia dei pedoni e le tabelle di correzione sono invece condivise solo tra i thread di uno stesso nodo NUMA. Le tabelle di correzione registrano le differenze tra la valutazione statica di una posizione e il suo punteggio di ricerca, e servono a migliorare la valutazione statica usata da alcune euristiche. Il commento del codice spiega che la condivisione è limitata a un nodo per evitare un eccesso di trasferimenti di dati tra

i nodi. Quando i thread sono creati, la funzione "ThreadPool::set" prepara per ogni nodo NUMA un insieme di storie condivise, la cui dimensione è la prima potenza di due non inferiore al numero di thread del nodo; il commento indica che la dimensione è scelta così per rendere l'indicizzazione più efficiente. Al momento dell'azzeramento, ogni thread è responsabile della propria parte delle storie condivise [11].

3.4.4 La votazione tra i thread per scegliere la mossa

Alla fine della ricerca ogni thread ha un proprio elenco di mosse radice, cioè delle mosse legali nella posizione di partenza, e per ciascuna un punteggio e una variante principale. La variante principale è la sequenza di mosse che il motore considera migliori e che quindi si aspetta siano giocate [12]. La mossa restituita al comando "go" può quindi provenire da un thread diverso dal principale: in certe condizioni la funzione "get_best_thread" sceglie il thread da cui prenderla [11]. La scelta avviene soltanto se valgono insieme quattro condizioni: l'opzione "MultiPV" vale 1, il comando "go" non ha limiti di profondità né di matto, la limitazione della forza di gioco non è attiva e il thread principale ha trovato almeno una mossa. In tutti gli altri casi la mossa è quella del thread principale [11]. "MultiPV" indica quante varianti migliori il motore stampa durante la ricerca. La limitazione della forza di gioco fa giocare il motore in modo più debole. Si ottiene con l'opzione "Skill Level", che assume valori da 0 a 20, oppure con "UCI_LimitStrength", che punta a un punteggio Elo impostato dall'utente; con il valore 20 e senza "UCI_LimitStrength" la limitazione non è attiva [12, 11].
Ogni thread vota per la propria mossa migliore. Il peso del voto cresce con il punteggio del thread, misurato rispetto al punteggio più basso tra i thread, e con la profondità che il thread ha completato. I pesi si sommano per ogni mossa, perciò una mossa scelta da più thread, o da thread che hanno raggiunto una profondità maggiore, ottiene più voti [11].
Il thread scelto si ricava confrontando, uno alla volta, tutti i thread con quello che al momento è il migliore, che all'inizio è il thread principale. Nel confronto un punteggio più alto è considerato migliore. Una vittoria dimostrata è un punteggio che raggiunge la soglia riservata ai matti a favore e alle vittorie garantite dalle tablebase, mentre una sconfitta dimostrata è il caso opposto. Le regole del confronto sono tre, applicate in quest'ordine [11]:

1. Se il thread migliore ha una vittoria dimostrata, cioè un matto o una conversione con le tablebase, se ne sceglie un altro solo se ha un punteggio più alto. Il commento indica lo scopo: ottenere il matto o la conversione più breve.
2. Se il thread migliore ha una sconfitta dimostrata, se ne sceglie un altro solo se ha anch'esso una sconfitta dimostrata e un punteggio più basso. Il commento parla anche qui della variante più breve.

3. In tutti gli altri casi vince il thread la cui mossa ha ricevuto più voti. A parità di voti conta il peso del voto del thread. Il codice tratta a parte i thread con una vittoria o una sconfitta dimostrata.

Se il thread scelto non è il principale, il motore stampa di nuovo le informazioni sulla variante principale, con la profondità completata da quel thread, e restituisce la sua mossa [11].

3.4.5 I nodi NUMA e la memoria condivisa

La sezione 3.4.2 ha introdotto il nodo NUMA come una porzione della memoria principale con i processori che la usano come memoria locale, più veloce di quella remota [7]. La documentazione di Stockfish descrive l'opzione "NumaPolicy" come lo strumento che associa i thread a un nodo NUMA per migliorare le prestazioni nei sistemi con più CPU o più domini NUMA. L'opzione "NumaPolicy" ha per impostazione predefinita il valore auto e accetta questi valori [12]:

- none: si assume un solo nodo NUMA e i thread non vengono associati a nessun processore.
- system: si usano le informazioni NUMA fornite dal sistema operativo e i thread sono associati di conseguenza.
- auto: il motore sceglie tra system e none in base al sistema.
- hardware: si usano le informazioni NUMA dell'hardware, ignorando le affinità impostate in precedenza. La documentazione lo consiglia quando Stockfish non usa tutti i thread.
- una configurazione personalizzata, in cui si indicano a mano i processori di ogni nodo.

Nel codice la configurazione è rappresentata dalla classe "NumaConfig", che elenca i processori di ogni nodo [11]. Per processore si intende un processore logico: l'Hyper-Threading permette di eseguire più thread su ogni core, che il sistema vede perciò come due core logici [22].
Una cache è una memoria intermedia tra i core e la memoria principale: ogni core ha di solito una cache L1 privata, e lungo il percorso verso la memoria principale si trovano altre cache, private o condivise [22]. Nel codice un dominio L3 è un gruppo di processori che condividono la stessa cache L3, e un nodo con più cache L3 viene suddiviso per impostazione predefinita in più nodi logici, uno per ogni dominio. I domini vengono poi raggruppati fino a un numero massimo di processori logici. Il commento spiega che questo limite bilancia il guadagno di forza dovuto alla maggiore condivisione delle storie con la perdita di velocità dovuta ai maggiori accessi tra cache diverse, e che l'utente può comunque cambiarlo [11]. Quando crea i thread, la funzione "ThreadPool::set" decide se associarli ai nodi. Con la politica none non li associa mai,

con system o con una configurazione personalizzata li associa sempre, con auto lo fa solo se la funzione "suggests_binding_threads" lo consiglia. Il commento spiega che di default non si associano i thread, a meno che non si sappia con certezza che si estendono su più nodi, perché con più istanze di Stockfish in esecuzione tutte quelle con un solo thread finirebbero sul primo nodo. La funzione decide così. Se l'affinità impostata dall'utente non coincide con quella del sistema operativo, consiglia sempre l'associazione. Altrimenti non la consiglia per un solo thread. Con più thread la consiglia solo se i nodi sono più di uno e i thread sono numerosi rispetto ai processori del nodo più grande o al numero di nodi non piccoli. Un nodo è piccolo se ha al più il 60 % dei processori del nodo più grande [11]. Quando i thread vanno associati, la funzione "distribute_threads_among_numa_nodes" assegna ogni thread al nodo che risulta meno occupato in proporzione al numero dei suoi processori [11].
Le reti neurali sono contenute in un oggetto replicato per nodo, cioè con una copia per ogni nodo: la copia del nodo 0 viene creata subito e le altre solo quando un thread di quel nodo ne ha bisogno [11]. Le copie sono tenute in memoria condivisa, e il commento del codice precisa che il valore è condiviso a livello di sistema per un singolo utente, cioè più istanze di Stockfish sullo stesso computer possono usare la stessa copia. Un processo è un'istanza di un programma, con il proprio codice e i propri dati in memoria, perciò due istanze di Stockfish sono due processi distinti, e tra processi diversi la memoria condivisa richiede un'interfaccia del sistema operativo [7]. Se la memoria condivisa non è disponibile, il codice usa un'alternativa con memoria locale [11].

3.4.6 La scalabilità con il numero di thread

La documentazione ufficiale dedica una pagina di dati, intitolata "Useful data", all'efficienza dei thread nello schema Lazy SMP. La pagina esprime l'efficienza come percentuale e la ricava in due modi, descritti di seguito. Afferma che in linea di principio Lazy SMP scala molto bene con il numero di core per quanto riguarda i nodi al secondo, cioè le posizioni esaminate in un secondo, e osserva che le misure reali sono influenzate da variazioni di frequenza, da SMT (hyperthreading) e dai limiti dell'hardware [12]. SMT è la tecnologia per cui ogni core esegue più thread e viene visto come due core logici [22].
Il primo modo di misura usa l'opzione "nodestime" di Stockfish, che indica al motore di contare il tempo trascorso con i nodi esaminati invece che con il tempo reale. Le partite si giocano quindi con un budget fisso di nodi. Si fanno giocare un giocatore con un thread e uno con più thread, e si modifica il numero di nodi di quest'ultimo finché le due forze si equivalgono. Il risultato è chiamato nodestime equivalente. La pagina riporta che cresce con il numero di thread, ma non troppo rapidamente, e che diminuisce quando aumenta il numero di nodi a disposizione. Con 64 thread e un budget di 240+2,4 milioni di nodi per partita il nodestime equivalente è circa il 200 % e l'efficienza è indicata come ancora attorno al 50 % [12]. I controlli di tempo

si scrivono come base più incremento: nei test Stockfish usa il controllo breve 10+0.1 e il controllo lungo 60+0.6, cioè i secondi per la partita più i secondi aggiunti a ogni mossa, indicati con le sigle STC e LTC [12]. Nel secondo modo l'efficienza è un rapporto tra guadagni in Elo. Elo è il punteggio con cui si valutano i giocatori: la differenza di Elo tra due giocatori è la differenza dei loro punteggi, e da essa si ricava la probabilità di vittoria attesa [7]. Si confronta il guadagno di otto thread contro uno con quello di un thread che dispone di un tempo otto volte maggiore. Se i due guadagni fossero uguali, l'efficienza varrebbe il 100 %. Con il controllo di tempo lungo la pagina ne ricava un'efficienza di circa l'82 % [12].
La pagina propone infine un modello, che estrapola l'efficienza a budget di nodi molto più grandi, cioè la stima oltre l'intervallo misurato. Secondo l'estrapolazione, con più di 300 thread ai controlli di tempo dei campionati TCEC l'efficienza potrebbe arrivare all'80 % o più, a condizione che i nodi al secondo crescano con il numero di thread [12]. TCEC è una sigla che la Chess Programming Wiki scioglie in Top Chess Engines Competition e descrive come una lista di valutazione e un torneo per motori [7].

3.5 La valutazione con la rete NNUE

La valutazione di una posizione è una funzione euristica che ne stima il valore relativo, e il codice di Stockfish la chiama valutazione statica [11]. NNUE è la sigla di Efficiently Updatable Neural Network, rete neurale aggiornabile in modo efficiente. Una rete neurale è formata da neuroni, ciascuno dei quali riceve più ingressi e produce una sola uscita. Gli ingressi sono moltiplicati per numeri chiamati pesi, sommati e passati a una funzione non lineare, detta funzione di attivazione [7].
I neuroni di una rete NNUE sono organizzati in strati lineari. Uno strato riceve dei numeri in ingresso. Ognuno viene moltiplicato per un peso, cioè un valore fisso della rete, e i risultati si sommano. Al totale si aggiunge un altro valore fisso, il bias. Pesi e bias sono i parametri dello strato. La documentazione indica tre principi di progetto: gli ingressi diversi da zero devono essere pochi, devono cambiare poco da una valutazione alla successiva, e la rete deve restare abbastanza semplice da funzionare con l'aritmetica intera. Il secondo principio permette di aggiornare una parte costosa della rete invece di rivalutarla per intero. Per usare i numeri interi al posto di quelli con parte decimale occorre la quantizzazione, cioè il passaggio del modello dai numeri in virgola mobile agli interi [12].
In Stockfish 18 la valutazione statica di una posizione è affidata a reti neurali di tipo NNUE. La funzione "Eval::evaluate" riceve: la posizione, le reti, la pila degli accumulatori e un valore di ottimismo, e restituisce un punteggio dal punto di vista del lato che deve muovere. Nello stesso file non restano termini scacchistici scritti a mano: ci sono la chiamata alle reti, una stima del solo materiale, cioè la differenza di valore dei pezzi tra i due lati, e le correzioni descritte nelle sezioni seguenti [11].

L'accumulatore è il risultato del primo strato per la posizione corrente. Poiché quello strato somma colonne della matrice di pesi, il risultato si può conservare come parte dello stato della posizione e aggiornare dopo ogni mossa, invece di ricalcolarlo a ogni valutazione [12]. Gli accumulatori delle posizioni raggiunte durante la ricerca sono tenuti in una pila, descritta successivamente [11]. L'ottimismo è un numero che il motore somma al punteggio di una posizione per rendere il suo gioco un po' più rischioso [11].
La ricerca chiede la valutazione quando entra in una posizione in cui il re non è sotto scacco. Se la tabella di trasposizione contiene già una valutazione valida per quella posizione, la ricerca la riusa. Altrimenti chiama la funzione di valutazione e scrive nella tabella il valore grezzo, prima di applicare la correzione della correction history, la tabella che registra le differenze tra la valutazione statica e il punteggio trovato dalla ricerca [11]. Le reti sono due, una grande e una piccola, ciascuna contenuta in un file con estensione ".nnue" [11]. Le sezioni seguenti seguono il percorso dei dati: il passaggio dalla valutazione classica alle sole reti, le caratteristiche in ingresso, l'accumulatore, gli strati e la quantizzazione, il ruolo delle due reti e il calcolo del punteggio finale.

3.5.1 Dalla valutazione classica alla sola rete neurale

La valutazione classica si basa su euristiche (come il vantaggio materiale e il posizionamento dei pezzi) ed è più veloce ma meno accurata della valutazione con rete neurale [12]. Dopo l'ingresso di NNUE nel 2020 le due valutazioni sono rimaste insieme nel codice per alcuni anni. Secondo la Chess Programming Wiki, nell'agosto 2020 una modifica detta hybrid fece usare la rete solo nelle posizioni con materiale abbastanza equilibrato, e la valutazione classica nelle altre [7]. La documentazione ufficiale aggiunge che i concetti della valutazione classica erano scritti da esperti e poi collaudati e ottimizzati con Fishtest, una coda distribuita di task per provare i motori scacchistici, mentre la valutazione NNUE calcola lo stesso valore con una rete neurale che parte da ingressi elementari [12].
La documentazione ufficiale colloca la rimozione della valutazione classica dal codice nel luglio 2023 [12]. Il post di Stockfish 16.1, del 24 febbraio 2024, la presenta come novità della versione [2]. Nel codice del tag "sf_18" il file "evaluate.cpp" non contiene più termini scacchistici scritti a mano. Oltre alla funzione di valutazione con le reti, contiene soltanto "simple_eval", che stima il solo materiale, "use_smallnet", che sceglie la rete, e "trace", che stampa il dettaglio della valutazione per il debug. La stima del materiale non serve a valutare la posizione: serve a scegliere quale delle due reti usare, come descritto nella sezione 3.5.5 [11].

3.5.2 Le caratteristiche in ingresso nella rete

L'ingresso di una rete NNUE è un vettore molto lungo in cui sono attive poche caratteristiche. Una caratteristica è un'informazione sulla posizione che la rete riceve come ingresso, per esempio la presenza di un certo pezzo su una certa casella. Il codice lo rappresenta come l'elenco degli indici delle caratteristiche attive e lo calcola due volte, una per ciascuna prospettiva, quella del Bianco e quella del Nero [11]. La documentazione parte dall'insieme più semplice, di 768 caratteristiche, una per ogni combinazione di casella, tipo di pezzo e colore (64 per 6 per 2), di cui in una posizione legale al massimo 32 sono attive [12].
Nel codice di Stockfish 18 le caratteristiche dei pezzi sono quelle della classe "HalfKAv2_hm". Il commento la descrive come la combinazione della posizione del re proprio e della posizione dei pezzi, con la scacchiera specchiata in modo che il re stia sempre sulle colonne da e a h. L'indice di una caratteristica dipende dalla prospettiva, dalla casella e dal tipo del pezzo e dalla casella del re. Le caselle sono ribaltate per il Nero e specchiate in orizzontale quando il re sta sulle colonne da a a d, e la casella del re determina uno di 32 gruppi. Il vantaggio dello specchio sta nel numero di ingressi della rete: poiché il re può trovarsi soltanto in 32 caselle invece che in 64, gli ingressi si dimezzano, da 45.056 a 22.528 [11]. La novità di Stockfish 18 è l'aggiunta di caratteristiche sulle minacce, chiamate "Threat Inputs" nel post di rilascio, che permettono alla rete di riconoscere più direttamente quali pezzi sono minacciati [2]. Nel codice la classe "FullThreats" definisce una caratteristica come un pezzo che attacca un altro pezzo: l'indice dipende dal pezzo attaccante e dalla sua casella, dal pezzo attaccato e dalla sua casella. La casella del re serve soltanto a orientare le caselle. Le caratteristiche attive si trovano scorrendo, per ogni pezzo, gli attacchi che cadono su caselle occupate, e una tabella di corrispondenza esclude alcune coppie di tipi di pezzo. Le dimensioni sono 79.856 e le caratteristiche attive al massimo 128 [11].
Le minacce non sono usate da entrambe le reti. Il trasformatore delle caratteristiche, la parte della rete che trasforma le caratteristiche attive negli accumulatori e questi nel vettore che entra negli strati, le prende in considerazione solo quando ha la dimensione della rete grande, quindi la rete piccola riceve soltanto le caratteristiche dei pezzi [11]. L'insieme delle minacce è già cambiato dopo Stockfish 18. Il post di Stockfish 19 parla della rimozione di caratteristiche di minaccia ridondanti e dell'introduzione di nuove caratteristiche sulle coppie di pedoni [2].

3.5.3 L'accumulatore e l'aggiornamento incrementale

L'accumulatore serve a non rifare da capo il calcolo del primo strato della rete a ogni valutazione. È il risultato di quello strato per la posizione corrente, e poiché lo strato somma colonne della tabella di pesi, il risultato si può conservare e aggiornare dopo ogni mossa [12]. Nel codice è la struttura "Accumulator", che per ciascun colore

contiene i valori del primo strato e una componente aggiuntiva chiamata "psqt". Il commit che la introdusse chiama PSQT i valori associati a ciascuna caratteristica e spiega che permettono alla rete di imparare il valore dei pezzi in modo più diretto [11]. Nella Chess Programming Wiki una piece-square table è un modo semplice di assegnare valori a pezzi specifici su caselle specifiche [7].
Durante la ricerca gli accumulatori delle posizioni raggiunte sono tenuti in una pila, cioè un elenco in cui si aggiunge e si toglie sempre l'ultimo elemento. Le pile sono due, una per le caratteristiche dei pezzi e una per le minacce, e ogni thread di ricerca ha le proprie. A ogni mossa eseguita la pila aggiunge uno stato e vi registra i pezzi e le minacce che sono cambiati, a ogni mossa annullata lo toglie. Il nuovo stato non viene calcolato subito: resta segnato come non calcolato finché la ricerca non chiede una valutazione. A quel punto il codice risale la pila fino all'ultimo stato già calcolato e applica in avanti le variazioni delle mosse intermedie. Se una di quelle mosse impone il ricalcolo, ricalcola invece lo stato corrente e ricostruisce all'indietro quelli intermedi [11].
Un aggiornamento dopo una mossa è piccolo. Per le caratteristiche dei pezzi si tolgono e si aggiungono al massimo due caratteristiche per prospettiva, e il contributo di ognuna si sottrae o si somma al valore già calcolato. Il commento del codice spiega che il limite è 2 perché si aggiorna una mossa alla volta. Il ricalcolo completo è necessario quando si muove il re della prospettiva considerata, perché tutte le caratteristiche dei pezzi dipendono dalla casella del re [11].
Anche il ricalcolo dei pezzi evita di ripartire da zero. La struttura "AccumulatorCaches", di cui ogni thread ha una copia, conserva per ogni casella del re e per ogni colore l'ultimo accumulatore calcolato, insieme alla disposizione dei pezzi con cui era stato calcolato. Al ricalcolo il codice confronta la posizione attuale con quella memorizzata e applica soltanto le differenze. Le minacce si ricalcolano invece a partire dall'elenco completo di quelle attive [11].

3.5.4 Gli strati della rete e la quantizzazione

Il primo passo trasforma gli accumulatori nel vettore che entra negli strati. Per ciascuna prospettiva, a partire da quella di chi muove, il codice somma l'accumulatore dei pezzi e quello delle minacce, limita i valori a un intervallo fisso e li combina a coppie. Il risultato è un vettore di numeri interi da 8 bit, che nella rete grande ha 1024 valori [11]. Gli strati sono definiti nella classe "NetworkArchitecture". Nella rete grande il primo strato porta i 1024 valori a 16 uscite, cioè 15 più una aggiuntiva. Il primo strato è scritto per ingressi in cui la maggior parte dei valori è zero: una funzione cerca gli indici dei valori diversi da zero, e la documentazione spiega che questo tipo di strato conviene perché salta le colonne il cui ingresso è nullo [11, 12]. L'intera sequenza di strati esiste in otto copie, e il motore ne usa una sola, scelta in base al numero di pezzi sulla scacchiera. Lo stesso criterio sceglie la componente

"psqt" tra le otto disponibili [11]. La Figura 3.5 rappresenta una versione semplificata della rete neurale NNUE di Stockfish 18.

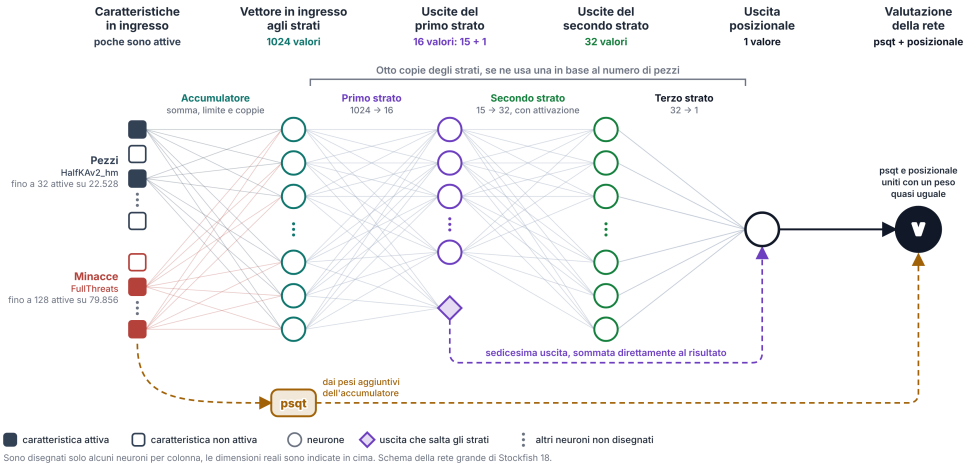


Figura 3.5: Schema semplificato della rete NNUE grande di Stockfish 18. Illustrazione dell'autore.

La rete lavora con numeri interi. Il passaggio dai numeri con la virgola a quelli interi si chiama quantizzazione, e la documentazione ne descrive lo schema generale: i valori del trasformatore e i pesi degli strati lineari vengono moltiplicati per costanti fisse che li portano nell'intervallo degli interi usati. La documentazione osserva anche che reti più grandi valutano con più precisione, ma che il costo in velocità può annullare il guadagno nel gioco reale [12]. Nel codice parti diverse della rete usano interi di dimensioni diverse, da 8 a 32 bit [11].

3.5.5 La rete grande e la rete piccola

Stockfish 18 usa due reti con la stessa forma e dimensioni diverse. La rete grande trasforma le caratteristiche in un vettore di 1024 valori, seguito da strati di 15 e 32 neuroni, ed è alimentata sia dalle caratteristiche dei pezzi sia dalle minacce. La rete piccola produce un vettore di 128 valori, con gli stessi strati successivi, e riceve soltanto le caratteristiche dei pezzi [11]. Il post di Stockfish 16.1 presenta per la prima volta una rete secondaria, usata per valutare rapidamente le posizioni già facilmente decise, e il post di Stockfish 19 ne annuncia il ritiro [2].
A decidere quale rete usare è una stima del solo materiale, la stessa descritta nella sezione sulla valutazione classica. Se la differenza di materiale tra i due lati supera una soglia pari a circa 4,6 volte il valore di un pedone, si usa la rete piccola, altrimenti la grande. In questo modo la rete piccola è riservata alle posizioni con un forte squilibrio materiale [11].
Se è stata usata la rete piccola e la stima ottenuta è, in valore assoluto, inferiore a 277, la posizione viene valutata di nuovo con la rete grande e il risultato della piccola

viene scartato. Il commento nel codice dice che la rivalutazione avviene quando una maggiore accuratezza giustifica il tempo speso. Ogni stato della pila contiene un accumulatore per la rete grande e uno per la piccola, e le cache di ricalcolo sono due, una per rete. Poiché gli accumulatori si aggiornano solo quando la ricerca chiede una valutazione, ciascuna rete porta avanti i propri soltanto quando viene interrogata [11].

3.5.6 Dalla uscita della rete alla valutazione finale

Ogni rete restituisce due valori. Il primo, chiamato "psqt", nasce dai pesi aggiuntivi dell'accumulatore ed esprime la differenza di valore tra il lato che muove e l'avversario. Il secondo, chiamato "posizionale", è l'uscita degli strati. La funzione "Eval::evaluate" li unisce in un'unica stima della rete, in cui i due valori hanno un peso quasi uguale [11]. Nel calcolo entra anche l'ottimismo, un piccolo valore che la pull request che lo ha introdotto descrive come sommato alla valutazione di Stockfish. Secondo la stessa pull request fa giocare Stockfish in modo un po' più rischioso: tiene più materiale sulla scacchiera quando ritiene di avere il vantaggio e mantiene la tensione più a lungo quando si difende. La pull request lo definisce simile nello spirito al vecchio "contempt" [11]. Il contempt factor è una stima della superiorità o inferiorità del programma rispetto all'avversario, usata come punteggio di patta per evitare le patte precoci contro avversari apparentemente più deboli [7].
Nel codice l'ottimismo è calcolato dalla ricerca. Alla radice, cioè nella posizione da cui parte la ricerca, il codice prende il punteggio medio della mossa radice e lo usa come centro della finestra di aspirazione iniziale. La finestra di aspirazione è un intervallo attorno a una stima del punteggio, di solito quella dell'iterazione precedente, usato come limiti della ricerca per ridurne lo spazio [7]. Prima di avviare la ricerca con quella finestra, il codice imposta l'ottimismo del lato che muove a partire da quel punteggio medio: il valore ha il segno del punteggio, cresce con esso e non supera un massimo. L'ottimismo dell'avversario è lo stesso valore con segno opposto [11]. La stima viene poi corretta in base alla complessità della rete, cioè a quanto i due valori, psqt e posizionale, differiscono tra loro. Quando la complessità è alta la stima diminuisce e l'ottimismo aumenta. Il valore finale combina stima e ottimismo pesandoli in base al materiale rimasto sulla scacchiera [11]. Il valore è poi ridotto in proporzione al contatore delle cinquanta mosse, che conta le mezze mosse dall'ultima cattura o dall'ultimo avanzamento di pedone. Il commento del codice descrive questo passo come uno smorzamento lineare della valutazione in caso di shuffling [11]. La regola delle cinquanta mosse dichiara patta la partita dopo cinquanta mosse complete consecutive senza catture né mosse di pedone [7].
Infine il valore è limitato in modo da non entrare nell'intervallo riservato alle tablebase, come dichiara il commento del codice [11]. Le tablebase di finale sono tabelle precalcolate, generate con un'analisi retrograda esaustiva, che per ogni posizione permettono di sapere se è patta, vinta o persa [7]. Il valore che esce dalla funzione è la valutazione statica grezza. La ricerca la scrive così nella tabella di trasposizione e ne

usa una versione corretta dalla correction history, la tabella che serve a migliorare la valutazione statica usata da alcune euristiche della ricerca [11]. Il punteggio resta in unità interne fino all'interfaccia UCI. Per mostrarlo in "centipawn", unità pari a circa un centesimo di pedone [7], il motore lo converte con un modello della probabilità di vittoria che si adatta con buona precisione alle statistiche delle partite di Fishtest a tempo lungo [11, 12].

3.6 Il machine learning nell'addestramento delle reti

L'addestramento è la fase in cui una rete neurale ottiene i propri parametri. I valori non vengono scritti da un programmatore: sono regolati a partire da esempi, con i metodi del machine learning, cioè dell'apprendimento automatico. La documentazione di NNUE-Pytorch descrive l'addestramento come la minimizzazione di una funzione di perdita, ottenuta con la discesa del gradiente [12]. I due termini sono spiegati nella sezione 3.6.2.
La sezione 3.5 ha descritto com'è fatta la rete e come viene usata durante la ricerca. Qui si vede da dove nascono i suoi parametri, in quattro passi: i dati su cui la rete impara, con il ruolo del progetto Leela Chess Zero (3.6.1); il programma che esegue l'addestramento e il modo in cui misura l'errore (3.6.2); la sequenza automatica che collega le fasi (3.6.3); la prova della rete con partite giocate (3.6.4).

3.6.1 I dati di addestramento e il ruolo di Leela Chess Zero

Per imparare, una rete ha bisogno di esempi. Nel caso di Stockfish un esempio è una posizione di scacchi con un valore associato, che la rete deve imparare a restituire. La documentazione di NNUE-Pytorch precisa che ogni esempio contiene le caratteristiche della posizione viste dai due lati, il lato che muove e un punteggio di valutazione, e che il punteggio è il valore che si vuole insegnare alla rete [12]. Una parte importante di questi esempi viene dal progetto Leela Chess Zero, abbreviato Lc0. Lc0 è un motore scacchistico a codice aperto basato su una rete neurale. Il sito del progetto lo presenta come un'implementazione open source ispirata ad AlphaZero, il sistema di DeepMind descritto nella sezione 2.4.1, e dice che la sua rete ha imparato a giocare con partite contro se stessa [23]. Il README del repository lo definisce un motore compatibile con il protocollo UCI, progettato per giocare a scacchi con una rete neurale [24]. Maharaj, Polson e Turk scrivono che il progetto nacque come tentativo di riprodurre il lavoro di AlphaZero con il calcolo collettivo (crowd computing) e che usa in gran parte la stessa ricerca e la stessa valutazione [25]. Il progetto Leela ha anche un secondo ruolo: produce dati. Nel suo blog scrive di generare una grande quantità di dati e di volerli rendere aperti e disponibili a tutti. Con quei dati si creano le reti di Lc0 e di altri motori, e i file sono elencati in un archivio pubblico. La raccolta è distribuita con la licenza Open Database License [23].

Il blog di Stockfish documenta l'uso di questi dati in tre occasioni. Il post su Stockfish 14, del 2021, dice che il gruppo di Leela ha messo a disposizione miliardi di posizioni valutate da Leela, unite a miliardi di posizioni valutate da Stockfish per addestrare la rete di quella versione, e che poter usare e combinare liberamente i due insiemi fu essenziale per il progresso ottenuto. Il post su Stockfish 17 afferma che Leela continua a fornire dati aperti, essenziali per addestrare le reti NNUE. Il post su Stockfish 18 parla di oltre 100 miliardi di posizioni con valutazioni di Lc0 e rimanda al post di Leela sui dati aperti [2].

3.6.2 Il trainer NNUE-Pytorch e la funzione di perdita

Il programma che esegue l'addestramento si chiama trainer. Per le reti di Stockfish è NNUE-Pytorch, un repository ufficiale descritto come trainer in PyTorch per le reti di valutazione NNUE [11]. Il post di Stockfish 18 lo indica come il trainer del progetto [2]. PyTorch è una libreria per l'apprendimento automatico: calcola da sola le correzioni da applicare ai parametri, con la differenziazione automatica [12].
La lettura dei dati è affidata a una libreria in C++ che interpreta rapidamente i file di addestramento. Alcuni thread preparano in modo asincrono interi gruppi di esempi, detti batch, che vengono poi convertiti nei tensori di PyTorch, cioè nelle tabelle di numeri che la libreria elabora [12]. L'addestramento è un ciclo. La rete valuta una posizione, il risultato viene confrontato con il valore atteso e una funzione di perdita misura quanto i due sono distanti. I parametri vengono poi spostati un poco nella direzione che riduce questa distanza: è la discesa del gradiente. La documentazione riassume dicendo che addestrare una rete significa minimizzare una funzione di perdita, che deve essere liscia e avere il minimo nel valore atteso [12].
Il valore atteso può essere di due tipi. Il primo è la valutazione che un motore ha dato alla posizione, il secondo è il risultato della partita. La documentazione introduce un parametro, chiamato lambda, che regola quanto peso dare all'uno e all'altro, e osserva che quale scelta funzioni meglio dipende dal caso [12]. Le due grandezze hanno scale diverse. La valutazione è espressa in centipawn, l'unità usata dai motori, o in una scala proporzionale; il risultato vale 0 per la sconfitta, 0,5 per la patta e 1 per la vittoria. La documentazione chiama la prima scala CP e la seconda WDL, dall'inglese win, draw, loss, e porta le valutazioni nella seconda con una funzione sigmoide. Osserva infatti che di solito la corrispondenza tra valutazione e risultato si approssima bene con una curva di questo tipo. Un fattore di scala, che dipende dal motore e dai dati, regola la curva [12]. Per misurare la distanza la pagina descrive due funzioni di perdita. L'errore quadratico medio prende il quadrato della differenza ed è presentato come molto semplice. L'entropia incrociata è usata di solito per problemi di classificazione continua. La pagina non indica quale delle due sia usata per le reti ufficiali di Stockfish, ma osserva che nell'errore quadratico l'esponente può essere maggiore di 2 e che le reti di Stockfish hanno dato buoni risultati con un esponente di 2,6 [12].

Il comando principale, "train.py", richiede due insiemi di dati: uno di addestramento, da cui si calcolano le correzioni, e uno di validazione, che serve a controllare le prestazioni tra una fase e l'altra. L'addestramento finisce dopo un numero di epoche fissato dall'utente, dove un'epoca è un passaggio completo sui dati; per maturare, le reti richiedono di solito circa 400 epoche. Il risultato è un file di checkpoint (.ckpt), che i motori non leggono: serve un file .nnue, ottenuto con il programma serialize.py, e la pagina precisa che la conversione comporta una perdita. La stessa pagina avverte che la perdita di validazione dipende dal dataset e dal calcolo della perdita, quindi confrontare due addestramenti su quel valore può trarre in inganno. In questi casi, scrive, bisogna far giocare le reti prodotte [12]. Le partite di collaudo sono nella sezione 3.6.4.

3.6.3 La pipeline automatizzata e riproducibile di Stockfish 18

Il post di Stockfish 18, pubblicato il 31 gennaio 2026, dedica una parte al flusso di addestramento rifatto. Dice che l'addestramento delle reti è passato a un modello automatizzato e riproducibile, che questo quadro permette di usare ricette standardizzate per concatenare fasi complesse, e che il passaggio facilita l'addestramento con oltre 100 miliardi di posizioni con valutazioni di Lc0 [2]. Una pipeline è una catena di fasi in cui il risultato di una è l'ingresso della successiva; qui le fasi sono quelle dell'addestramento descritte nella sezione 3.6.2. La documentazione di NNUE-Pytorch descrive invece strumenti di automazione che si possono leggere. Lo script "easy_train.py" gestisce addestramento e collaudo come un unico flusso, in modo organizzato e autonomo. Ogni esecuzione è un esperimento, con il proprio trainer, i propri giocatori, i propri registri e i propri modelli di partenza, e può essere ripresa. Lo script addestra la rete, converte i checkpoint in file .nnue, fa giocare un motore di riferimento contro un motore con la nuova rete usando il programma "c-chess-cli" e libri di aperture, e classifica i risultati con il programma "ordo" [12].
Il repository dello stesso progetto contiene lo script "run_games.py", che converte in .nnue i checkpoint trovati nella cartella dell'esecuzione, controlla la comparsa di nuovi checkpoint e fa giocare le reti per trovare la migliore [11]. La pagina di "easy_train.py" avverte inoltre che i file di esempio non sono pronti per la produzione e contiene parti ancora da scrivere. I due strumenti vanno quindi letti come esempi documentati di automazione, non come la descrizione del flusso di Stockfish 18 [12].

3.6.4 Fishtest: il collaudo distribuito di reti e modifiche

Fishtest è una coda distribuita di task per provare i motori scacchistici; l'istanza principale prova Stockfish [11]. La documentazione di NNUE-Pytorch dice che, per giudicare le reti prodotte, bisogna farle giocare (sezione 3.6.2). Per provare una rete nuova la documentazione di Fishtest indica proprio Fishtest, come per le modifiche

al codice [12]. Il lavoro è svolto da volontari. Chi vuole contribuire installa un worker sul proprio computer, che gioca partite in sottofondo. Il worker scarica da solo un libro di aperture, il programma "fastchess" che gestisce le partite e il codice sia di Stockfish attuale sia della modifica da provare, li compila e poi invia i risultati al server. Il server gestisce una coda di test con priorità regolabili, calcola valori probabilistici dai risultati ricevuti e li aggiorna, interrompe un test quando il risultato è statisticamente significativo e ne pubblica l'esito [11].
Il criterio statistico è descritto nella pagina sulla matematica di Fishtest. Il progetto usa il GSPRT, una generalizzazione del test sequenziale SPRT (Sequential Probability Ratio Test), e i limiti dei test sono espressi in Elo normalizzato, il che rende la durata attesa dipendente solo dai limiti scelti [12]. Il nome SPRT indica un test che valuta i dati man mano che arrivano le partite. Nel procedimento standard si esegue prima un test a controllo di tempo breve (STC) e, se lo supera, uno a controllo di tempo lungo (LTC). La documentazione raccomanda di non fermare un test standard, salvo un errore grave all'inizio, perché Fishtest esegue tutte le partite necessarie a ottenere un risultato statisticamente definitivo [12].
Per una rete la procedura ha alcuni passi in più. La rete va caricata su Fishtest, con un account, e il caricamento la rende disponibile con licenza CC0 (Creative Commons Zero, un tipo di licenza che serve per perdere tutti i diritti d'autore sull'opera, cosicché tutti possono utilizzarla e modificarla). Il file prende il nome "nn-SHA.nnue", dove SHA sono le prime 12 cifre dell'impronta sha256 del file. Su un ramo del codice si sostituisce il nome predefinito della rete, indicato in "evaluate.h", con quello della nuova rete, poi si eseguono i test STC e LTC come al solito. La pagina esclude un'opzione EvalFile nei test e dice che la forza della rete viene misurata nei normali test di progresso e regressione [12].

3.7 Ricerca con rete neurale e approccio di AlphaZero a confronto

La sezione 2.4.1 ha descritto AlphaZero e la sezione 3.5 la rete NNUE di Stockfish. Qui i due approcci si confrontano su cinque punti: dove sta la rete, come si cerca, quanto si calcola, come si addestra e che cosa mostrano i confronti diretti. Leela Chess Zero, definito nella sezione 3.6.1, usa in gran parte la stessa ricerca e la stessa valutazione di AlphaZero e viene quindi trattato insieme ad AlphaZero [25]. In Stockfish la rete ha un solo compito: dare il valore di una posizione. La ricerca resta di tipo alpha-beta, descritta nella sezione 2.3.2, e la rete prende il posto della valutazione classica scritta a mano [25]. Anche l'ordine in cui le mosse vengono provate non dipende dalla rete: nel codice del tag "sf_18" lo calcola la classe "MovePicker" con tabelle di storia e regole semplici, come nella sezione 3.4.1 [11].
In AlphaZero e in Leela la rete ha invece due uscite, come nella sezione 2.4.1. La descrizione tecnica di Leela le chiama politica, un elenco ordinato delle mosse da

considerare, e valore, un numero compreso tra meno uno e uno che stima l'esito della partita [23]. La politica indica quindi quali mosse esplorare, un compito che in Stockfish svolge "MovePicker". La ricerca di Leela si chiama PUCT, che molti chiamano Monte Carlo Tree Search. La stessa pagina la dice uguale a quella del lavoro su AlphaGo Zero, con una differenza: al posto delle simulazioni fino alla fine della partita, dette rollout, c'è una rete che stima il loro esito [23]. Maharaj et al. descrivono Stockfish come una ricerca alpha-beta e AlphaZero come una ricerca Monte Carlo che individua le linee migliori con campionamenti ripetuti [25]. Una differenza misurabile è il numero di posizioni esaminate. Nel 2017 AlphaZero ne esaminava circa 80.000 al secondo e Stockfish circa 70 milioni (sezione 2.4.1). Maharaj et al. riportano le velocità medie della finale della TCEC Cup 8 del marzo 2021: circa 150 milioni di nodi al secondo per Stockfish e circa 140.000 per LCZero. Gli stessi autori scrivono che NNUE è circa due volte più lenta della valutazione classica di Stockfish [25]. Sul punto in cui sta la competenza del motore, gli autori riassumono: la magia di Stockfish è programmata nella ricerca, quella di LCZero nella valutazione [25].
Anche l'addestramento parte da punti diversi. AlphaZero produce i propri dati giocando contro se stesso a partire da gioco casuale (sezione 2.4.1), e la rete di Leela ha imparato con partite contro se stessa [23]. Le reti NNUE di Stockfish imparano invece da posizioni già valutate, che in parte vengono da Leela (sezione 3.6.1). I due approcci non sono quindi separati: l'addestramento di Stockfish usa dati prodotti da un motore del secondo tipo. Cambia anche l'hardware. Stockfish lavora solo sulla CPU [2], mentre il README di Lc0 parla di accelerazione su GPU e di molti supporti hardware [24]. Per un confronto diretto Maharaj et al. usano il problema scacchistico di Plaskett, uno studio di finale composto nel 1970 e diventato celebre nel 1987, e riportano che Stockfish lo risolve meglio di LCZero. Il risultato riguarda quel solo problema [25].

3.8 Sintesi del capitolo

Il capitolo ha descritto Stockfish 18, rilasciato il 31 gennaio 2026, nelle parti che servono a capire come sceglie una mossa [2]. Il motore comunica con l'esterno attraverso il protocollo UCI e si regola con opzioni come Threads e Hash [12]. Non è l'ultima versione: il sito ufficiale indica come corrente Stockfish 19, uscito il 5 settembre 2026 [2]. La ricerca esplora l'albero delle mosse a profondità crescente e usa la tabella di trasposizione per non rifare il lavoro sulle posizioni già incontrate [12]. Nel codice la tabella è una sola ed è condivisa da tutti i thread. I thread possono scriverla senza sincronizzazione: il commento del codice spiega che sincronizzare costerebbe tempo e quindi punti Elo, che le collisioni possono causare errori e che il rischio diminuisce con tabelle più grandi [11].
I thread cooperano con Lazy SMP: eseguono tutti la stessa funzione di ricerca e

condividono la tabella [12]. Il thread principale gestisce il tempo e avvia gli altri, e di norma la mossa finale è scelta con una votazione tra i thread [11]. La valutazione è affidata alle sole reti NNUE: in "evaluate.cpp" non restano termini scritti a mano, e le reti sono due, una grande e una piccola [11]. La rete può girare sul processore perché conserva l'accumulatore e lo corregge dopo ogni mossa, e perché usa numeri interi [12]. Stockfish 18 aggiunge caratteristiche in ingresso sulle minacce, mentre Stockfish 19 ritira la rete piccola [2].
I parametri delle reti non sono scritti da un programmatore: sono ottenuti con un addestramento che minimizza una funzione di perdita [12]. Una parte dei dati viene dal progetto Leela Chess Zero, con oltre 100 miliardi di posizioni nel caso di Stockfish 18 [2]. Le reti e le modifiche al codice sono provate con partite giocate da volontari su Fishtest [11]. Il confronto con AlphaZero e Leela mostra due modi di usare una rete. Stockfish la impiega per valutare le posizioni dentro una ricerca alpha-beta, mentre AlphaZero e Leela la usano anche per indicare le mosse da esplorare, in una ricerca di tipo Monte Carlo [25, 23]. I due approcci non sono separati, perché Stockfish si addestra anche con dati prodotti da Leela (sezione 3.6.1). Il motore descritto in questo capitolo è quello che il progetto presentato nei capitoli successivi interroga tramite il protocollo UCI [7].

Capitolo 4

Stockfish Board

4.1 Presentazione del progetto

Stockfish è uno dei motori di scacchi più forti al mondo [17], ma funziona solo da terminale: non ha una parte grafica e interagisce con l'utente soltanto attraverso messaggi di testo. Il protocollo che regola questi messaggi si chiama UCI, Universal Chess Interface [12].
Per questo motivo è nato Stockfish Board, un progetto che realizza in Unity un'interfaccia grafica per gli scacchi. L'interfaccia è affiancata dal motore esterno Stockfish, nella versione 18, usato come strumento di analisi e di controllo sulle mosse giocate. La differenza principale rispetto ad altre interfacce grafiche è l'attenzione alle capacità del motore stesso. L'applicazione ha due modalità principali, tra cui il giocatore può passare liberamente: un editor di posizione, che permette di comporre a mano una disposizione dei pezzi o di partire dalla posizione di apertura standard, e una modalità di analisi, in cui il giocatore gioca la posizione impostata mossa per mossa con il supporto in tempo reale del motore. In questo modo Stockfish funziona da "allenatore": per la posizione corrente suggerisce la mossa migliore che ha trovato.
Nella modalità di analisi il giocatore muove i pezzi con il click. Ogni mossa è convalidata rispetto alle regole ufficiali degli scacchi, comprese le mosse speciali (arrocco, cattura en passant, promozione del pedone) e le condizioni di fine partita (scacco matto, stallo, patta per la regola delle cinquanta mosse o per triplice ripetizione). Intanto un pannello mostra l'analisi di Stockfish per la posizione corrente: la mossa migliore suggerita, la valutazione numerica (lo Score), la profondità di ricerca raggiunta, il tempo impiegato in millisecondi e la linea di gioco principale prevista (principal variation). Una barra di valutazione mostra in forma grafica il vantaggio tra i due colori. Il sistema permette anche di navigare nella cronologia della partita. Con le frecce della tastiera il giocatore torna a una posizione precedente e può continuare con una mossa diversa, creando una nuova variante. Il comportamento è simile a quello delle principali piattaforme scacchistiche online, così da risultare intuitivo per gli utenti esperti e accessibile ai principianti.
La scelta di Unity è stata naturale. Il suo modello a componenti (i MonoBehaviour) ha permesso di scomporre un sistema complesso come una scacchiera in moduli

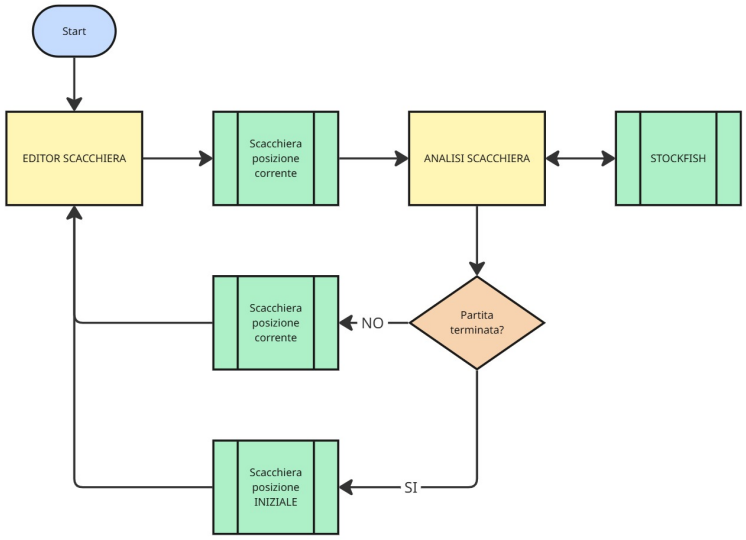


Figura 4.1: Diagramma di flusso di StockfishBoard. Illustrazione dell'autore.

indipendenti, testabili separatamente. Le strutture principali sono tre: la generazione della griglia, che funge da scacchiera; la gestione del movimento del singolo pezzo, con la cattura e le mosse speciali (en passant e arrocco); la comunicazione con il motore esterno, affidata a un thread dedicato che la rende asincrona. Il ciclo di aggiornamento Update, che Unity esegue a ogni frame, permette inoltre di gestire in modo efficace l'input dell'utente.
Per l'analisi si è scelto Stockfish, un motore open source tra i più forti oggi disponibili. Comunica con il protocollo UCI, un'interfaccia testuale documentata e indipendente dal linguaggio e dalla piattaforma che la ospita [12]. Per questo si può trattare come un processo esterno, del tutto separato da Unity, e dialogare con lui tramite gli stream standard di input e output, senza integrazioni a livello di libreria. La comunicazione passa da un thread diverso da quello principale di Unity: così programma e motore non si bloccano a vicenda e non devono aspettarsi.

4.2 Architettura del sistema

Il programma è diviso in sistemi distinti. L'input dell'utente li attraversa uno dopo l'altro fino ad arrivare al motore scacchistico esterno, come mostra la Figura 4.1.

Alla base c'è la scacchiera, rappresentata dagli script BoardGenerator, PieceSpawner, CellData e DraggablePiece, che si occupano della sua creazione fisica. Come supporto grafico c'è un'immagine SVG semplice, che mostra all'utente com'è disposta la scacchiera e dove stanno i pezzi. Sopra l'immagine, BoardGenerator e CellData

creano una griglia di box collider, ciascuno posizionato esattamente su una casella. L'utente crede di premere, per esempio, il cavallo, ma il click del mouse seleziona in realtà la casella, mentre i pezzi sono una semplice rappresentazione grafica. Questo funziona grazie alla classe CellData, che contiene la riga e la colonna della casella (int), il nome della casella (stringa), il pezzo che la occupa (GameObject) e un booleano che indica se è occupata. Ogni casella ha inoltre due sprite figli, un cerchio e un anello semitrasparenti. Vengono attivati solo quando il pezzo selezionato può spostarsi su quella casella: il cerchio pieno se la casella è vuota, l'anello se nella casella di arrivo c'è un pezzo che si può catturare. PieceSpawner posiziona i pezzi sulla scacchiera, o nella posizione iniziale o in una posizione modificata.
Il cuore del programma è MoveInput. Questo script gestisce l'input del giocatore e l'intero stato della partita: mosse giocate, turno corrente, cronologia navigabile e cache delle posizioni analizzate. Di conseguenza decide anche come deve apparire la scacchiera in ogni istante. Il livello più esterno è StockfishManager, che gestisce la comunicazione con il processo Stockfish tramite il protocollo UCI e mantiene un thread dedicato alla lettura asincrona delle risposte.
MoveInput e StockfishManager, i due componenti centrali, hanno responsabilità ben separate. StockfishManager non conosce nulla della scacchiera: invia comandi nel formato UCI, interpreta le righe di testo ricevute e le traduce in eventi tipizzati (mosse legali disponibili, aggiornamenti di punteggio, mossa suggerita e così via). MoveInput non comunica mai direttamente con il processo esterno: riceve gli eventi e decide cosa farne in base allo stato della partita che possiede. Le risposte di Stockfish arrivano in modo asincrono, perciò un evento relativo a una richiesta ormai superata può arrivare dopo l'invio di una richiesta nuova. Per questo ogni evento emesso da StockfishManager porta un'etichetta testuale che identifica in modo univoco la posizione a cui si riferisce. MoveInput la usa per riconoscere e scartare le risposte obsolete, invece di applicarle per errore allo stato corrente.
A livello di interfaccia l'applicazione alterna due schermate, realizzate come due oggetti Canvas di Unity attivati in modo esclusivo: l'editor di posizione (CanvasBoardEditor) e la schermata di analisi (CanvasAnalysis). Il passaggio è coordinato da BoardEditorManager, che con un semplice bottone ne controlla l'attivazione ed espone lo stato corrente con la proprietà IsEditModeActive. Ogni componente che riceve input dal giocatore sulla scacchiera, in particolare CellData e DraggablePiece, consulta questa proprietà per decidere a chi passare un click. Se l'editor è attivo, l'interazione è gestita interamente da BoardEditorManager (piazzamento o rimozione di un pezzo). Altrimenti il click segue il percorso consueto verso MoveInput, per la selezione e l'esecuzione delle mosse. In questo modo la logica di gestione del click non è duplicata in due componenti, e CellData e DraggablePiece restano indipendenti dalla modalità attiva. Quando il giocatore preme "Inizia analisi", BoardEditorManager genera una rappresentazione FEN della posizione composta e la passa a MoveInput, che inizializza una nuova partita a partire da essa. Nel percorso inverso, il ritorno all'editor, lo stato di gioco accumulato (mosse giocate, cronologia, cache

delle analisi) viene azzerato del tutto, perché la sessione di analisi successiva partirà da una posizione diversa.

4.2.1 Rappresentazione della scacchiera e dei pezzi

All'avvio la scacchiera non è un asset già assemblato, ma viene generata a runtime dallo script BoardGenerator, che, come detto, crea dei box collider e li affianca a formare una griglia. La generazione avviene nel metodo Awake() e non in Start(), perché Unity esegue tutti i metodi Awake() della scena prima di qualunque Start(). Questo è necessario perché PieceSpawner, per collocare i pezzi, ha bisogno di conoscere le coordinate delle 64 caselle.
Ogni cella è un GameObject autonomo, posizionato secondo coordinate calcolate dalla coppia (colonna, riga). Ha un BoxCollider2D impostato come trigger, che serve a intercettare i click del giocatore senza introdurre una collisione fisica vera e propria, e il componente CellData, che conserva la notazione scacchistica della cella (ad esempio "e4") e il riferimento al pezzo eventualmente presente. Le 64 celle sono raccolte in una matrice bidimensionale (squaresGrid), che ogni altro componente consulta per trovare una casella a partire dalle sue coordinate.
Ogni pezzo è descritto dal componente PieceData, che ne conserva il tipo (pedone, cavallo, alfiere, torre, donna, re), il colore e la posizione corrente sulla griglia. La disposizione iniziale della partita è codificata come un array di otto stringhe in formato FEN, una per riga, e questo rende immediato modificarla o estenderla. L'istanziazione di un singolo pezzo non è duplicata nei vari punti del codice che ne hanno bisogno, ma centralizzata nel metodo SpawnPieceFromSymbol, che riceve un simbolo in stile FEN e la cella di destinazione. Il metodo è richiamato da tre contesti indipendenti: lo schieramento iniziale della partita, il piazzamento manuale di un pezzo nell'editor e la ricostruzione silenziosa della scacchiera durante la navigazione nella cronologia. Una modifica al modo in cui un pezzo viene posizionato si fa così in un solo punto del codice.
Un problema emerso durante lo sviluppo riguarda la sovrapposizione fisica tra un pezzo e la sua casella. Entrambi hanno un collider alla stessa profondità (coordinata Z), quindi il raggio generato da Unity per interpretare un click poteva colpire l'uno o l'altro in modo non deterministico. A volte il click era intercettato dalla cella invece che dal pezzo, e il pezzo diventava saltuariamente non selezionabile, senza alcun errore in console che ne spiegasse la causa. La soluzione è tenere ogni pezzo a una profondità leggermente più vicina alla camera rispetto alla propria cella ($Z = +0.1$ anziché $Z = 0$), così che il raggio del click incontri sempre per primo il pezzo, quando c'è.
Lo scarto va imposto in ogni punto in cui un pezzo è istanziato o spostato: spawn iniziale, mosse ordinarie, spostamento della torre durante l'arrocco, sostituzione in caso di promozione, piazzamento nell'editor, ricostruzione durante la navigazione. Va fatto ogni volta perché copiare la posizione di una cella così com'è copierebbe anche

la sua coordinata Z e annullerebbe l'effetto voluto. La Figura 4.2 mostra l'ordine dei layer usati.

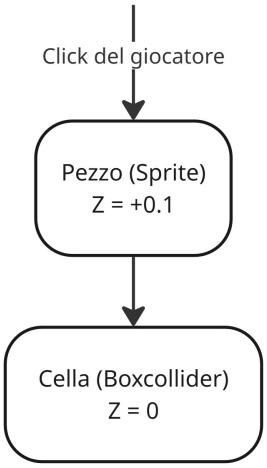


Figura 4.2: Ordine di stratificazione dell'input dell'utente. Illustrazione dell'autore.

4.3 Integrazione con il motore Stockfish

Stockfish è lanciato come processo esterno con la classe Process di .NET. Gli stream standard di input e output sono reindirizzati (RedirectStandardInput, RedirectStandardOutput), così da scrivere i comandi e leggere le risposte a livello di codice, senza un terminale interattivo. Subito dopo l'avvio StockfishManager esegue l'handshake iniziale previsto dal protocollo [12]: invia uci per chiedere l'identificazione del motore, setoption name Hash value 32 per raddoppiare la tabella hash rispetto al valore predefinito (16 megabyte) e infine isready. Il motore è considerato pronto a ricevere comandi operativi solo quando risponde readyok.
Il metodo StandardOutput.ReadLine() legge una riga dallo standard output di Stockfish ed è una chiamata bloccante: il thread che la esegue resta sospeso finché il processo esterno non produce una riga. L'attesa non è costante, perché dipende dal comando inviato e dalla profondità di ricerca configurata. Va da pochi millisecondi, per un semplice comando di conteggio delle mosse legali, a diversi secondi, quando si chiede un'analisi approfondita. Unity esegue quasi tutta la sua logica, compresi l'aggiornamento della scena, l'input del giocatore e il rendering, su un unico thread principale. Se la lettura bloccante girasse su questo thread, l'intera applicazione resterebbe sospesa per tutta l'attesa: nessun fotogramma verrebbe disegnato e nessun click registrato. Per un'applicazione interattiva anche uno o due secondi di blocco

sarebbero inaccettabili.
Per evitarlo la lettura è affidata a un thread separato, dedicato solo a questo compito. Il thread parte una volta sola, all'apertura del processo esterno, e resta attivo per tutta la sessione di gioco. Esegue un ciclo continuo di lettura e deposita ogni riga ricevuta in una coda ConcurrentQueue<string>. Una coda concorrente permette al thread di lettura di scrivere e al thread principale di leggere nello stesso istante in sicurezza, senza lock o mutex espliciti, la cui gestione manuale aggiungerebbe complessità e rischio di errori.
Il metodo Update() del thread principale di Unity svuota questa coda a ogni fotogramma ed elabora le righe accumulate nel frattempo. L'operazione non è mai bloccante: se la coda è vuota, il ciclo termina subito e Unity prosegue con le sue attività. La reattività dell'applicazione non dipende quindi dai tempi di risposta di Stockfish. Il giocatore può continuare a usare l'interfaccia mentre l'analisi procede in sottofondo.

4.3.1 Meccanismo a stati e validazione delle mosse

Le righe prodotte da Stockfish non descrivono da sole il proprio significato: una riga come "e2e4: 1" assume un significato diverso a seconda che sia attesa come mossa legale o arrivi in un altro contesto. StockfishManager risolve l'ambiguità con un'unica variabile di stato, che indica in ogni momento "cosa ci si aspetta che arrivi": Idle, WaitingPerft, WaitingBoardInfo o WaitingBestMove. Prima di inviare un comando che genera una risposta strutturata, il codice imposta lo stato corrispondente. Il metodo ProcessLine, eseguito su ogni riga estratta dalla coda, consulta lo stato per decidere come interpretarla e lo aggiorna quando la fase finisce.
La Figura 4.3 mostra il meccanismo a stati creato per gestire le risposte di Stockfish.

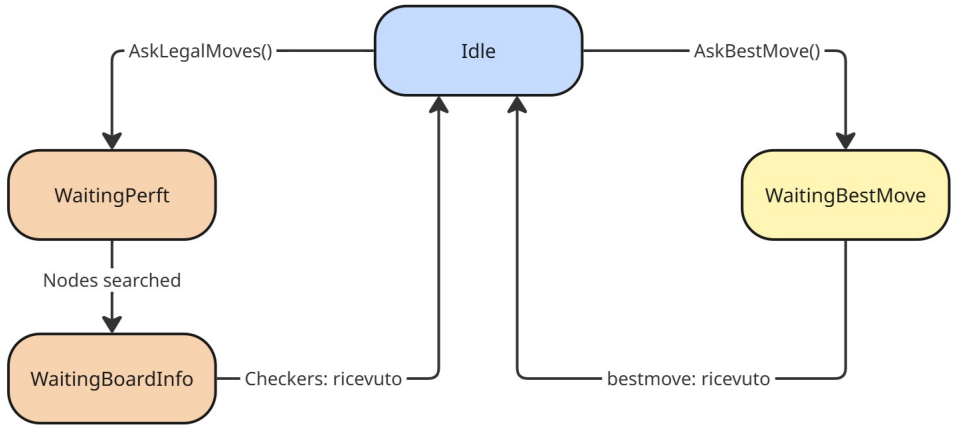


Figura 4.3: Meccanismo a stati che gestisce la risposta di Stockfish. Illustrazione dell'autore.

Il ciclo tipico è questo: da Idle si passa a WaitingPerft quando si invia una richiesta di mosse legali. Al termine del perft si passa automaticamente a WaitingBoardInfo, per interrogare lo stato della posizione (scacco, FEN). Ricevuta questa informazione si torna a Idle e, se nel frattempo è stata accodata una richiesta di analisi, si passa a WaitingBestMove, da cui si rientra in Idle alla ricezione della mossa suggerita.
Il protocollo UCI offre comandi per impostare una posizione (position), avviare una ricerca (go) e ricevere la mossa migliore (bestmove), ma non ne ha uno per chiedere quali mosse siano legali [12]. Serviva quindi un altro modo di ottenere questa informazione, indispensabile per un'applicazione che deve validare ogni mossa del giocatore prima di eseguirla. La soluzione usa il comando go perft N, un parametro di go previsto da Stockfish e nato per un altro scopo: verificare il generatore di mosse del motore, contando le posizioni raggiungibili esplorando l'albero delle mosse fino alla profondità N [12]. Con N uguale a 1 il comando diventa utile qui. Invece di un solo numero complessivo, Stockfish elenca ogni mossa legale della posizione corrente, ciascuna seguita dal conteggio delle posizioni raggiungibili con quella mossa, che è sempre uno perché la profondità è di una sola mossa [11]. StockfishManager legge queste righe, ne estrae la notazione della mossa e le raccoglie in una lista con tutte le mosse legali della posizione.
È un uso diverso da quello per cui il comando è stato pensato, ma l'informazione ottenuta è esatta: l'elenco restituito da un perft di profondità 1 è per definizione completo e corretto, perché per contare le mosse Stockfish deve generare tutte le mosse legali. Questo comando permette anche di mantenere reattiva l'interfaccia.

Un perft di profondità 1, a differenza di una ricerca vera e propria, richiede un tempo di calcolo trascurabile, qualunque sia la profondità impostata per l'analisi, che può arrivare a valori per cui una ricerca completa richiederebbe diversi secondi. MoveInput sfrutta questa differenza e fa le richieste a Stockfish in due fasi, in un ordine preciso: chiede sempre per prime le mosse legali con il perft e solo dopo, in background, l'analisi vera e propria con go depth N. Poiché la prima fase finisce quasi subito, il giocatore può selezionare e muovere un pezzo appena termina, senza aspettare la ricerca. L'analisi continua da sola e aggiorna l'interfaccia (punteggio, mossa suggerita, linea di gioco) man mano che i risultati arrivano.

4.4 Logica di gioco e validazione delle mosse

MoveInput non verifica mai la legalità di una mossa applicando le regole degli scacchi. Il compito è lasciato a Stockfish, che con il perft a profondità 1 (Sezione 4.3) restituisce l'elenco completo delle mosse ammesse nella posizione corrente. L'elenco tiene conto di tutte le condizioni del regolamento: pezzi inchiodati, re sotto scacco, diritti di arrocco ancora disponibili e così via.
Quando il giocatore tenta una mossa, IsLegalMove costruisce la notazione UCI corrispondente, ad esempio "e2e4", e verifica che compaia come prefisso in una delle stringhe della lista legalMoves ricevuta da Stockfish. Il confronto usa StartsWith invece di un'uguaglianza esatta per un motivo preciso: Stockfish restituisce una promozione come stringa di cinque caratteri, ad esempio "e7e8q", mentre il giocatore seleziona la cella di arrivo prima di aver scelto il pezzo di promozione, e quindi produce solo i primi quattro caratteri. La stessa lista serve anche a un secondo scopo. Quando il giocatore seleziona un pezzo, SelectCell filtra legalMoves cercando le stringhe che iniziano con il nome della cella selezionata e ne estrae la cella di arrivo, per attivare l'evidenziazione. Distingue una mossa semplice da una cattura a seconda che la cella di destinazione sia occupata. Mosse disponibili e mosse valide derivano così dalla stessa fonte, e la logica di evidenziazione non deve calcolare da sé quali destinazioni siano raggiungibili.
Poiché dipende da una risposta asincrona, finché Stockfish non ha inviato la lista aggiornata, segnalata dal flag legalMovesReady, SelectCell rifiuta ogni selezione di pezzo. Il perft a profondità 1 richiede un tempo trascurabile (Sezione 4.3), quindi l'attesa è di norma impercettibile, ma il controllo serve a evitare che una mossa sia valutata con una lista non ancora aggiornata alla posizione corrente. Le responsabilità sono quindi divise: stabilire se una mossa è legale spetta al motore, eseguirla (aggiornare lo stato interno della partita e la scacchiera) spetta a MoveInput. Le mosse speciali descritte qui sotto seguono lo stesso principio. Stockfish le include, in quanto legali, nella lista del perft, quindi il codice locale non deve stabilire se un arrocco, una cattura en passant o una promozione siano permessi. Deve solo riconoscerli, per applicarne gli effetti sulla scacchiera.

Negli scacchi standard le mosse di arrocco sono quattro, ciascuna con una notazione UCI fissa: "e1g1" e "e1c1" per il bianco, "e8g8" e "e8c8" per il nero. MoveInput confronta la notazione della mossa eseguita con queste quattro stringhe, invece di calcolare se il pezzo mosso sia un re spostato di due colonne. Se coincide, HandleCastling sposta anche la torre coinvolta, scegliendo colonna di partenza, colonna di arrivo e riga in base alla stringa riconosciuta.
La cattura en passant richiede invece un riconoscimento basato su un pattern, perché non esiste un insieme finito di notazioni da confrontare. Il pattern è questo: il pezzo che si muove è un pedone, lo spostamento è in diagonale e la cella di arrivo è vuota al momento della mossa. Un pedone può muoversi in diagonale solo per catturare, quindi queste tre condizioni insieme identificano senza ambiguità una cattura en passant, dato che Stockfish ha già accertato la legalità della mossa. Il pedone avversario da rimuovere non sta sulla cella di arrivo, ma su quella adiacente, sulla stessa riga del pedone che si muove e sulla colonna di destinazione. HandleEnPassant individua questa cella a partire da queste coordinate e vi rimuove il pezzo.
La promozione è più complessa, perché richiede una scelta del giocatore e interrompe il normale flusso sincrono di una mossa. Il riconoscimento segue lo stesso principio delle altre mosse speciali: il pezzo mosso è un pedone e la riga di arrivo è l'ultima della scacchiera per il suo colore. A questo punto ExecuteMove non può completare la mossa. Chiama PromotionManager.PromotePawn, passandogli la cella di destinazione, il colore del giocatore e una funzione di callback da eseguire quando la scelta è disponibile. FinishMove, il metodo che registra la mossa e notifica Stockfish, viene richiamato solo dentro questa callback, dopo che il giocatore ha scelto il pezzo dal popup.
La scelta si fa con due prefab, uno per colore, ciascuno con quattro pulsanti per donna, torre, alfiere e cavallo. Due prefab separati evitano di scambiare a codice le immagini dei pulsanti in base al colore: PromotionManager attiva il prefab del colore giusto sopra la cella di promozione e lo disattiva appena il giocatore sceglie. La Figura 4.4 mostra i due prefab.

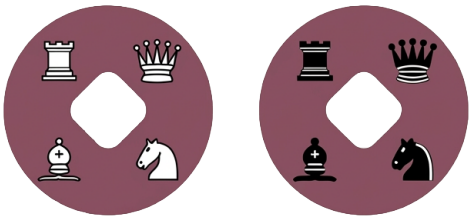


Figura 4.4: Pulsanti di scelta del pezzo di promozione. A sinistra il prefab per il bianco, a destra quello per il nero. Illustrazione dell'autore.

Quando il giocatore sceglie un pulsante, PromotionManager sostituisce il pedone con il pezzo scelto, richiamando SpawnPieceFromSymbol (Sezione 4.2.1) con il

simbolo FEN corrispondente. La lettera legata al pulsante torna a ExecuteMove tramite la callback e viene aggiunta in coda alla notazione della mossa. Si ottiene così la stringa a cinque caratteri che Stockfish si aspetta, ad esempio "e7e8q" per una promozione a donna, identica a una voce della lista delle mosse legali.
Ogni volta che Stockfish invia le mosse legali per la posizione corrente, MoveInput ne controlla il numero prima di ogni altra elaborazione. Se la lista è vuota la partita è finita. La differenza tra scacco matto e stallo si stabilisce interrogando di nuovo Stockfish con il comando di debug "d", il cui output ha una riga "Checkers" con gli eventuali pezzi che danno scacco al re di turno [11]. Se la riga non è vuota è scacco matto, e il vincitore è il colore opposto a quello rimasto senza mosse. Se è vuota è stallo, quindi patta.
La partita può finire in patta anche quando esistono mosse legali, per due regole lette dal FEN restituito da Stockfish. La regola delle cinquanta mosse usa il contatore delle semimosse senza catture né mosse di pedone, contenuto nel FEN [19]: a cento semimosse la partita è dichiarata patta. La triplice ripetizione si verifica con uno storico delle posizioni già incontrate, che usa come chiave i primi quattro campi del FEN (disposizione dei pezzi, turno, diritti di arrocco, possibilità di cattura en passant). Questi campi identificano la posizione a prescindere dai contatori, che cambiano a ogni mossa. Quando la stessa chiave compare per la terza volta, la partita termina in parità.

4.5 Interfaccia utente e feedback al giocatore

L'interfaccia ha due schermate, come visto nella Sezione 4.2. Qui si descrivono le loro parti.
Il testo della cronologia è ricostruito da UpdateMoveListDisplay ogni volta che si esegue una mossa o il giocatore cambia la posizione mostrata. Le mosse sono numerate a coppie, secondo la convenzione della notazione scacchistica, con la mossa del bianco e quella del nero sulla stessa riga, e indicano la casa di partenza e quella di arrivo del pezzo mosso. La mossa della posizione visualizzata è sottolineata con i tag di rich text di TextMeshPro, così il giocatore vede subito dove si trova nella cronologia. Lo scroll automatico verso il fondo del testo, che serve a mostrare sempre l'ultima mossa giocata, scatta solo quando la cronologia cresce perché è stata eseguita una mossa. Quando il testo si aggiorna per un altro motivo lo scroll non è forzato, per non spostare la vista lontano da ciò che il giocatore sta guardando. Quando la partita termina, il risultato in notazione standard ("1-0", "0-1" oppure "1/2-1/2", a seconda che si tratti di scacco matto o di una delle condizioni di patta della Sezione 4.4) è aggiunto come ultima riga del testo, associato alla posizione in cui la partita è finita.
Oltre alla cronologia l'interfaccia mostra le informazioni dell'analisi di Stockfish: profondità raggiunta, profondità selettiva, nodi esplorati, tempo impiegato, nodi

al secondo, linea di gioco principale e punteggio della posizione dal punto di vista del Bianco. Come descritto nella Sezione 4.2, decidere cosa mostrare non spetta a StockfishManager, che si limita a notificare gli aggiornamenti con eventi, ma a MoveInput. MoveInput aggiorna i campi di testo solo quando l'aggiornamento riguarda la posizione che il giocatore sta guardando.
La barra di valutazione mostra il punteggio numerico come vantaggio tra i due colori. Il punteggio, in centesimi di pedone, è convertito in un valore compreso tra zero e uno con una tangente iperbolica. Questa funzione comprime i vantaggi molto ampi: la barra non va all'estremo per differenze che, pur grandi in valore numerico, non corrispondono a una vittoria già decisa. Quando Stockfish segnala uno scacco matto forzato invece di un punteggio in centesimi di pedone, la barra è saturata quasi del tutto verso il colore che ha il matto. La barra si aggiorna con un unico metodo pubblico, SetScore, chiamato ogni volta che è disponibile un nuovo punteggio, così il comportamento è lo stesso comunque si sia ottenuto il punteggio.
Il giocatore può cambiare la profondità di ricerca con un campo di input testuale, validato prima di essere accettato. Un valore non numerico o fuori dall'intervallo consentito è respinto con un avviso, e il campo torna a mostrare la profondità in uso, così il giocatore vede che la modifica non ha avuto effetto. Solo se il valore supera il controllo, StockfishManager lo applica e notifica il cambiamento.

4.5.1 Editor di posizioni personalizzate

Quando apre l'applicazione, il giocatore si trova nella schermata dell'editor con la posizione iniziale già impostata. Se vuole modificarla, sceglie il pezzo da piazzare con uno dei dodici pulsanti dell'editor, uno per ogni combinazione di pezzo e colore, ciascuno associato al proprio simbolo in notazione FEN. Con un simbolo selezionato, il click su una cella lo piazza lì e sostituisce l'eventuale pezzo già presente. Se la cella contiene già esattamente il pezzo selezionato, lo stesso click lo rimuove: questo comportamento a interruttore permette di correggere un piazzamento senza un pulsante di cancellazione. Un pulsante separato attiva comunque una modalità di sola cancellazione, utile per togliere un pezzo senza sapere in anticipo quale sia o senza selezionare prima il suo simbolo. Per svuotare del tutto la scacchiera e inserire una posizione da zero c'è il pulsante con la croce. L'ultimo pulsante, rosso e con il simbolo di "ricomincia", cancella tutti i pezzi e reimposta la posizione iniziale.
A differenza della validazione delle mosse, delegata a Stockfish (Sezione 4.4), l'editor applica alcuni vincoli direttamente, perché la posizione composta non è ancora passata a nessun motore. Il primo riguarda il numero di re: una partita regolare ne ha esattamente uno per colore, quindi piazzare un secondo re dello stesso colore rimuove automaticamente quello già presente, ovunque si trovi, invece di rifiutare l'operazione. I diritti di arrocco del FEN generato, descritto nel paragrafo seguente, non sono impostati a mano né sempre disponibili per default, ma calcolati dalla posizione dei pezzi: un lato ottiene il diritto di arrocco solo se il re e la torre corrispondente stanno

ancora sulla casa originale. In questo modo non si dichiara disponibile un arrocco che la disposizione dei pezzi non potrebbe più permettere.
Quando il giocatore preme il pulsante che avvia l'analisi (quello con il simbolo di Stockfish), la scacchiera composta è tradotta in una stringa FEN. La griglia è letta una riga alla volta, dall'ottava alla prima, e le celle vuote consecutive sono scritte come un numero invece che una per una, secondo la convenzione standard [19]. Il turno di partenza è sempre il bianco e non è prevista alcuna cattura en passant all'avvio, perché una posizione composta a mano non ha una mossa precedente da cui possa derivare.
Il FEN è passato al metodo StartGameFromFen di MoveInput, che inizializza lo stato della partita da questa posizione, invece che dalla disposizione iniziale standard, e azzera la cronologia e la cache di eventuali partite precedenti.

4.5.2 Navigazione nella cronologia e sistema di cache

Quando il giocatore si sposta nella cronologia con le frecce della tastiera, la scacchiera è ricostruita interamente in locale, senza usare Stockfish per la sola rappresentazione visiva. La ricostruzione parte dal FEN di apertura della partita e riapplica in sequenza, fino all'indice scelto, le mosse già registrate. Usa una versione semplificata dell'esecuzione di una mossa, che non controlla la legalità, perché ogni mossa è già stata validata quando è stata giocata, e non fa richieste al motore. Poiché il meccanismo è tutto locale, la navigazione è istantanea, qualunque sia il tempo che Stockfish impiega ad analizzare la posizione raggiunta.
Ogni posizione già incontrata è conservata in una voce di cache, indicizzata dalla sequenza di mosse che porta a essa dall'inizio della partita. Una voce contiene due gruppi di informazioni, ottenuti in momenti diversi. Il primo comprende le mosse legali, lo stato di scacco e l'eventuale esito di fine partita, disponibili appena Stockfish restituisce il perft. Il secondo comprende i dati di analisi (mossa suggerita, punteggio, profondità effettivamente raggiunta, linea di gioco principale), che arrivano più tardi e in modo incrementale durante la ricerca.
In modalità di analisi, quando il giocatore raggiunge una posizione già presente in cache, il sistema distingue tre situazioni, come mostra la Figura 4.5.

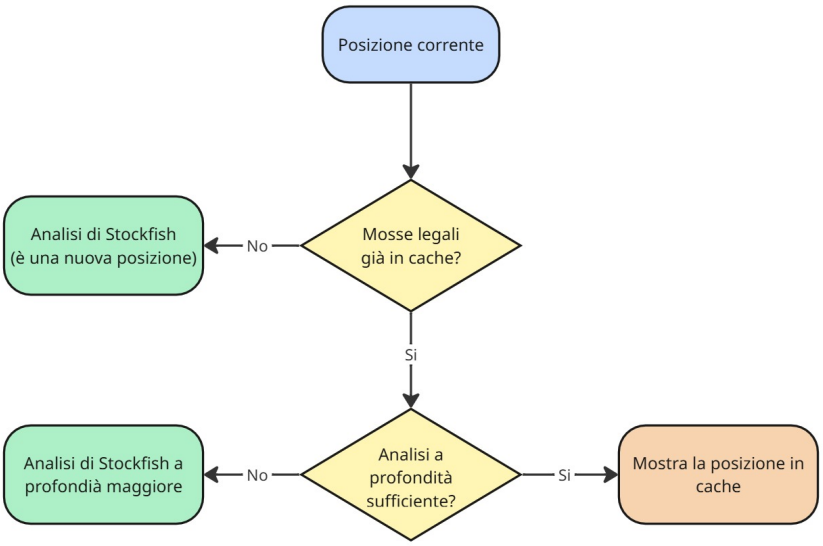


Figura 4.5: Sistema che gestisce la cache. Illustrazione dell'autore.

Il primo controllo chiede se la posizione è in cache con le mosse legali già calcolate. Se la risposta è no, la posizione non è mai stata incontrata e la richiesta è completa: MoveInput chiede sia le mosse legali sia l'analisi, come in una partita giocata dal vivo (Sezione 4.3). In questo caso il secondo controllo non si fa, perché non ha senso chiedersi se l'analisi basti quando della posizione non si sa ancora nulla.
Se le mosse legali sono già note, il secondo controllo verifica se la cache contiene anche un'analisi di profondità almeno pari a quella impostata (Sezione 4.5). Se l'analisi manca, o c'è ma è troppo poco profonda perché nel frattempo il giocatore ha aumentato il valore configurato, si chiede soltanto una nuova analisi, senza ripetere il perft, perché le mosse legali restano valide qualunque sia la profondità di ricerca. Il giocatore può comunque muovere subito, perché la legalità di una mossa non dipende da quanto a fondo Stockfish abbia cercato la risposta migliore.
Se entrambe le condizioni sono soddisfatte, cioè le mosse legali sono note e l'analisi è abbastanza profonda, la posizione è mostrata solo con i dati della voce di cache, senza inviare comandi a Stockfish. È l'unico dei tre casi in cui il motore non viene interpellato.
La navigazione non aspetta mai Stockfish, quindi il giocatore può spostarsi su una nuova posizione mentre è ancora in corso una richiesta per quella precedente. Senza precauzioni, la risposta in ritardo aggiornerebbe l'interfaccia con dati che non riguardano la posizione mostrata. Per questo ogni evento notificato da StockfishManager (Sezione 4.2) porta con sé la sequenza di mosse a cui si riferisce. MoveInput aggiorna sempre la cache con il dato ricevuto, ovunque sia il giocatore, ma applica l'aggiornamento ai pannelli di analisi e alla barra di valutazione solo se la sequenza

coincide con quella della posizione visualizzata. Una risposta in ritardo per una posizione abbandonata è quindi conservata per un uso futuro, ma non cambia ciò che il giocatore vede.
Se il giocatore torna indietro nella cronologia e gioca una mossa diversa da quella registrata in quel punto, le mosse successive della diramazione abbandonata sono eliminate insieme alle loro voci di cache, che resterebbero orfane e non sarebbero più raggiungibili. Anche le regole di patta della Sezione 4.4 tengono conto di questa distinzione: ai fini della triplice ripetizione conta solo una mossa realmente giocata, mentre rivisitare la stessa posizione navigando nella cronologia non la fa scattare, anche se Stockfish dà lo stesso tipo di risposta. Tutta la cache è svuotata quando il giocatore torna all'editor (Sezione 4.5.1) o avvia una nuova partita da una posizione diversa, perché le posizioni analizzate appartengono a una linea di gioco che non esiste più.

4.6 Principali problemi incontrati e soluzioni

Il primo problema di concorrenza riguardava la sequenza della Sezione 4.3: mosse legali richieste per prime, analisi in background subito dopo. Nella prima versione la richiesta di analisi veniva inviata subito dopo l'avvio del perft, senza aspettare che finisse. Poiché AskBestMoveFromMoves interrompe qualunque elaborazione in corso prima di avviarne una propria, la seconda richiesta interrompeva il perft ancora in corso. La catena di comandi successiva, cioè l'interrogazione dello stato della posizione con il comando di debug "d" e la notifica dell'evento con le mosse legali, non si completava mai. Il sintomo era che la mossa suggerita non veniva mai calcolata, senza alcun errore visibile.
La soluzione è stata incatenare le fasi invece di lasciarle indipendenti. La richiesta di analisi è ora accodata internamente a StockfishManager e parte solo quando la catena (perft, interrogazione dello stato, notifica dell'evento) è conclusa, mai prima.
Un secondo problema è comparso dopo l'introduzione della navigazione nella cronologia (Sezione 4.5.2). Scorrere avanti e indietro tra le stesse posizioni, senza giocare alcuna mossa, faceva scattare la triplice ripetizione (Sezione 4.4) dopo poche decine di passaggi. Ogni volta che una posizione veniva osservata, anche di passaggio, il suo conteggio nello storico aumentava come se fosse stata raggiunta con una mossa reale. La correzione distingue le due situazioni: solo le posizioni raggiunte con una mossa effettivamente eseguita sono contate ai fini della ripetizione. Rivisitarle navigando nella cronologia non cambia il conteggio, anche se Stockfish dà la stessa risposta.
Il problema più sottile riguardava l'interazione tra la navigazione, volutamente mai bloccante, e l'associazione tra risposte e posizione osservata (Sezione 4.5.2). Se il giocatore si spostava su una nuova posizione mentre la ricerca per la precedente era ancora in corso, il codice interrompeva la ricerca e ne avviava subito una nuova, sovrascrivendo nello stesso momento la variabile che registra a quale posizione si

riferisce la ricerca attiva. Le righe residue della ricerca interrotta potevano però arrivare dopo, perché la comunicazione è asincrona (Sezione 4.3). Venivano allora lette quando la variabile indicava già la posizione nuova e finivano nella voce di cache sbagliata.
L'effetto visibile era una linea di gioco suggerita senza senso per la posizione mostrata, perché apparteneva a una ricerca su un'altra posizione. La correzione ha eliminato la sovrascrittura immediata: se arriva una nuova richiesta di analisi mentre una precedente è ancora attiva, la nuova resta in attesa e parte solo quando la ricerca in corso ha restituito per intero la propria risposta. Così nessuna riga può essere letta nel momento sbagliato.

4.7 Sintesi del capitolo

Il progetto descritto in questo capitolo è un'applicazione Unity per giocare e analizzare partite di scacchi, costruita attorno a Stockfish come motore esterno di validazione e valutazione. Oltre all'esecuzione delle mosse, il sistema gestisce le mosse speciali meno immediate da implementare (arrocco, cattura en passant, promozione) e le condizioni di fine partita: scacco matto, stallo, regola delle cinquanta mosse e triplice ripetizione. A questo nucleo si aggiungono funzioni per studiare una posizione: un editor che permette di comporre una disposizione dei pezzi, una barra di valutazione e pannelli di analisi con il giudizio di Stockfish in tempo reale, e la navigazione nella cronologia, che permette di rivedere e ramificare una partita già giocata. La navigazione è resa praticabile da una cache che evita di chiedere al motore un'analisi già calcolata.
Alcuni aspetti del sistema sono meno robusti di quanto potrebbero essere, per scelta o per i limiti di tempo del progetto. Il processo Stockfish è chiuso correttamente quando l'applicazione termina in modo ordinato, ma non è protetto da un arresto improvviso dell'intera applicazione, per esempio un crash: in quel caso il processo esterno può restare attivo in background senza che il giocatore lo sappia. Per proteggerlo bisognerebbe legare il ciclo di vita del processo a quello dell'applicazione a livello di sistema operativo, con un meccanismo disponibile su Windows ma non ancora integrato. Un secondo limite riguarda la validazione delle posizioni composte nell'editor (Sezione 4.5.1). I controlli applicati, un solo re per colore e diritti di arrocco coerenti con la posizione dei pezzi, riducono ma non eliminano la possibilità di generare una posizione non valida, ad esempio con un pedone sull'ultima riga o con il re del lato che non deve muovere già sotto scacco. Le versioni più recenti di Stockfish, a partire dalla 19, controllano la posizione ricevuta in modo più severo: davanti a una posizione non valida terminano il processo, invece di comportarsi in modo imprevedibile come prima [11]. È un motivo in più per rafforzare questi controlli prima di aggiornare il motore usato dal progetto.
Oltre a rafforzare questi due aspetti, il progetto ha alcune estensioni naturali. Salvare e

caricare una partita, funzioni oggi assenti, permetterebbe di riprendere l'analisi in una sessione successiva senza ricomporre la posizione nell'editor. Il supporto alle varianti Chess960, già previsto dal protocollo UCI [12], richiederebbe modifiche contenute rispetto all'integrazione esistente con Stockfish e allargherebbe l'uso dello strumento oltre gli scacchi tradizionali. Infine l'architettura a eventi correlati (Sezioni 4.2 e 4.5.2), nata per la navigazione nella cronologia, offre già una base per analizzare più posizioni in parallelo, ad esempio per confrontare due varianti affiancate. Il progetto attuale non esplora questa direzione, ma la struttura del codice non impedirebbe di percorrerla.

Capitolo 5

Conclusioni

La tesi partiva da due difficoltà. La prima era capire come funziona un motore moderno come Stockfish 18, le cui informazioni sono sparse tra il blog del progetto, la documentazione e il codice sorgente. La seconda era usarlo in un'applicazione, cosa che richiede di gestire un processo esterno e di ottenere dal motore dati che il protocollo UCI non offre tra i suoi comandi.
Per la prima difficoltà il Capitolo 2 ha ripercorso la storia dei motori scacchistici, da Turing e Shannon fino ad AlphaZero e all'adozione di NNUE da parte di Stockfish. Il Capitolo 3 ha poi descritto Stockfish 18 a partire dal post di rilascio, dalla documentazione ufficiale e dal codice del tag `sf_18`. Per la seconda difficoltà il Capitolo 4 ha presentato StockfishBoard, un'applicazione Unity scritta in C# che avvia Stockfish come processo esterno e lo interroga tramite UCI per validare le mosse e analizzare le posizioni.
Lo studio di Stockfish 18 ha mostrato che il motore cerca la mossa esplorando l'albero a profondità crescente, con una tabella di trasposizione condivisa da tutti i thread [11]. I thread cooperano con Lazy SMP e di norma la mossa finale è scelta con una votazione tra loro [11]. La valutazione è affidata alle sole reti NNUE, una grande e una piccola, che conservano un accumulatore aggiornato dopo ogni mossa [11, 12]. Le reti si addestrano anche con dati di Leela Chess Zero, oltre 100 miliardi di posizioni nel caso di Stockfish 18 [2], e le modifiche sono collaudate da volontari su Fishtest [11]. Il confronto con AlphaZero e Leela ha chiarito la differenza principale: Stockfish usa la rete per valutare le posizioni dentro una ricerca alpha-beta, mentre AlphaZero e Leela la usano anche per indicare le mosse da esplorare, in una ricerca di tipo Monte Carlo [25].
Per l'applicazione il risultato è un programma completo per giocare e analizzare posizioni. La legalità delle mosse è affidata a Stockfish, che con il comando perft a profondità 1 restituisce l'elenco delle mosse ammesse, e la lettura delle sue risposte avviene su un thread dedicato, così l'interfaccia resta reattiva durante l'analisi. Una macchina a stati interpreta le risposte, e una cache indicizzata dalla sequenza di mosse permette di navigare nella cronologia senza richiedere di nuovo le analisi già calcolate. Durante lo sviluppo sono emersi e sono stati risolti tre problemi di concorrenza: la sovrapposizione tra la richiesta delle mosse legali e quella di analisi,

il conteggio delle ripetizioni durante la sola navigazione e le risposte tardive associate alla posizione sbagliata.
Il valore della tesi sta nel mettere insieme, con le fonti indicate, informazioni che il progetto Stockfish distribuisce tra blog, documentazione e codice. L'applicazione offre inoltre un esempio funzionante di integrazione di un motore esterno tramite UCI, con soluzioni a problemi che può incontrare chi sviluppa un'interfaccia simile: la lettura bloccante delle risposte, le richieste sovrapposte e le risposte che arrivano in ritardo. Il contributo ha due limiti. Il primo è che la descrizione riguarda Stockfish 18, mentre la versione corrente è la 19, che ha una nuova architettura di rete e ritira la rete piccola [2]. Il secondo è che alcune parti della documentazione e del codice, quelle sull'addestramento e sui test, non sono state esaminate, e che manca un confronto di forza aggiornato tra Stockfish e Leela.

5.1 Sviluppi futuri

Per l'applicazione, gli sviluppi più immediati riguardano la robustezza e le funzionalità:

- legare la vita del processo di Stockfish a quella dell'applicazione a livello di sistema operativo, meccanismo disponibile su Windows ma non ancora integrato, così che il motore non resti attivo dopo un arresto improvviso;
- rendere più severi i controlli sulle posizioni composte nell'editor, per esempio con un pedone sull'ultima riga o con il re del lato che non muove già sotto scacco, anche perché Stockfish, a partire dalla versione 19, verifica la posizione ricevuta in modo più rigoroso [11];
- aggiungere il salvataggio e il caricamento delle partite, il supporto alle varianti Chess960, già previsto da UCI [12], e l'analisi di due varianti affiancate, che l'architettura a eventi non impedirebbe;
- aggiornare l'applicazione a Stockfish 19 e provarla con più utenti e in scenari diversi, come l'analisi delle proprie partite da parte di giocatori, per verificarne l'uso reale.

Per lo studio del motore, la descrizione può essere estesa a Stockfish 19. I punti rimasti aperti si potrebbero chiarire leggendo le parti della documentazione e del codice sull'addestramento e sui test che non sono state esaminate, e cercando o producendo un confronto di forza tra le versioni attuali di Stockfish e di Leela.

Bibliografia

[1] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314):256–275, 1950.

[2] Stockfish. Sito ufficiale e blog del progetto Stockfish. `https://stockfishchess.org`.

[3] Garry Kasparov and Frederic Friedel. Reconstructing turing's "paper machine". EasyChair Preprint no. 3, 2017.

[4] Alan M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[5] Jimmy Soni and Rob Goodman. A man in a hurry: Claude Shannon's new york years. IEEE Spectrum, `https://spectrum.ieee.org/a-man-in-a-hurry-claude-shannons-new-york-years`, July 2017.

[6] John McCarthy, Marvin L. Minsky, Nathaniel Rochester, and Claude E. Shannon. A proposal for the dartmouth summer research project on artificial intelligence, 1955. Testo datato 31 agosto 1955, pubblicato in AI Magazine, 27(4), 12, 2006.

[7] Chess Programming Wiki. Chess Programming Wiki. `https://www.chessprogramming.org`.

[8] Chess.com. Computers and chess: A history. `https://www.chess.com/article/view/computers-and-chess---a-history`.

[9] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[10] Murray Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002. Copia consultata: `https://www.cefns.nau.edu/~edo/Classes/CS470-570_WWW/Readings/Campbell2002.pdf`.

[11] Stockfish. Stockfish, repository GitHub del progetto (codice sorgente, readme, pull request, Fishtest, NNUE-Pytorch). `https://github.com/official-stockfish`, 2026. Codice sorgente: versione con tag sf_18, `https://github.com/official-stockfish/Stockfish`.

[12] Stockfish. Documentazione di Stockfish (comprende Fishtest e NNUE-Pytorch). `https://official-stockfish.github.io/docs/`.

[13] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Mat-

thew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[14] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[15] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, 2017.

[16] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[17] Computer Chess Rating Lists. CCRL 40/40, lista di tutti i motori. `https://computerchess.org.uk/ccrl/4040/rating_list_all.html`, 2026. Lista aggiornata al 2 ottobre 2026.

[18] FIDE. Open chess museum: Magnus Carlsen. `https://museum.fide.com/champions/magnus-carlsen`.

[19] Steven J. Edwards. Standard: Portable game notation specification and implementation guide. `https://www.saremba.de/chessgml/standards/pgn/pgn-complete.htm`, 1994. Sezione 16.1; copia consultata online.

[20] Albert L. Zobrist. A new hashing method with application for game playing. Technical Report 88, Computer Sciences Department, University of Wisconsin, 1970.

[21] Bozza dello standard C++. `https://eel.is/c++draft/`.

[22] Intel Corporation. Pagine su Hyper-Threading Technology e Memory Performance in a Nutshell. `https://www.intel.com`.

[23] Leela Chess Zero. Sito e blog del progetto Leela Chess Zero. `https://lczero.org`.

[24] Leela Chess Zero. Lc0, repository GitHub del progetto Leela Chess Zero. `https://github.com/LeelaChessZero/lc0`.

[25] Shiva Maharaj, Nick Polson, and Alex Turk. Chess AI: Competing paradigms for machine intelligence, 2021.

Ringraziamenti

Questa tesi conclude il mio percorso di studi e mi ha permesso di unire una passione di sempre, gli scacchi, a ciò che ho imparato in questi anni. Gli scacchi sono stati una costante della mia vita e mi hanno sempre affascinato per il mistero che portano con sé. Poterli studiare dal punto di vista dell'intelligenza artificiale è stato per me un traguardo importante.

Lavorare a questo progetto mi ha fatto crescere. Ho dovuto leggere fonti ufficiali e articoli scientifici, capire come funziona un motore scacchistico e costruire un'applicazione in Unity. Sento che questo lavoro, insieme agli studi che l'hanno preceduto, mi ha formato.

Ringrazio il Prof. Aldo Franco Dragoni per la fiducia che mi ha dato e per avermi permesso di sviluppare questo lavoro su un argomento a cui tengo molto.

Civitanova Marche, Ottobre 2026

Piero Ciccalé