Implementazione Parallela e Multipiattaforma di Programmazione Genetica Lineare

Corso di Laurea in Intelligenza artificiale e Data Analytics

Checchin Paolo

Relatore: Prof. Luca Manzoni

26/03/2025

Indice

Capitolo 1

Introduzione

1.1 Contesto e Motivazione

L'intelligenza artificiale rappresenta ad oggi uno dei campi più dinamici e promettenti dell'informatica moderna, con applicazioni che spaziano dal riconoscimento vocale alla guida autonoma, dall'analisi di dati medici alla generazione automatica di contenuti. All'interno di questo vasto panorama, l'informatica evolutiva costituisce un'area di ricerca affascinante, che trae ispirazione dai principi dell'evoluzione biologica per sviluppare algoritmi in grado di risolvere problemi complessi.

La programmazione genetica, in particolare, è una delle espressioni più affascinanti dell'informatica evolutiva. Introdotta da John Koza nei primi anni '90 [8], essa espande il concetto di algoritmo genetico [7] alla generazione automatica di programmi per computer, permettendo di affrontare problemi per i quali non esistono soluzioni algoritmiche convenzionali o dove tali soluzioni risulterebbero troppo complesse da implementare manualmente.

Tra le varie declinazioni della programmazione genetica, la Programmazione Genetica Lineare (LGP) emerge come un paradigma particolarmente interessante, in cui i programmi generati sono rappresentati come sequenze lineari di istruzioni, in modo analogo in cui le istruzioni eseguite dai processori vengono rappresentate in linguaggio macchina tramite una lista di bytecode. A differenza della programmazione genetica tradizionale, che utilizza strutture ad albero, l'approccio lineare offre vantaggi in termini di efficienza computazionale e semplicità di implementazione, risultando particolarmente adatto per applicazioni che richiedono elevate prestazioni.

Il problema che questa tesi affronta è la necessità di strumenti efficienti, scalabili e multipiattaforma per la programmazione genetica lineare. Nonostante negli anni siano stati sviluppati numerosi framework per l'implementazione di algoritmi evolutivi, molti di essi presentano limitazioni in termini di efficienza computazionale, portabilità o facilità d'uso. In un'epoca in cui l'ottimizzazione computazionale e la programmazione parallela

sono diventate essenziali per affrontare problemi complessi, risulta cruciale lo sviluppo di implementazioni che sfruttino appieno le capacità dell'hardware moderno senza sacrificare la portabilità o la facilità d'utilizzo.

1.2 La Programmazione Genetica Lineare

Il concetto fondamentale alla base della programmazione genetica risiede nell'emulazione dei processi evolutivi biologici per la generazione automatica di programmi. Questi processi, originariamente teorizzati da Darwin, si manifestano attraverso la selezione naturale, la variazione genetica e l'ereditarietà. Nella loro trasposizione computazionale, tali meccanismi vengono applicati a popolazioni di programmi candidati a risolvere un problema, che evolvono progressivamente verso soluzioni sempre più efficaci.

La selezione identifica i programmi più performanti in base a una funzione di valutazione predefinita, emulando il principio darwiniano della sopravvivenza del più adatto.

La generazione di nuovi individui avviene mediante ricombinazione e mutazione. La ricombinazione, o crossover, consente lo scambio di segmenti di istruzioni tra programmi diversi, favorendo la diffusione di subroutine efficaci all'interno della popolazione. La mutazione introduce variabilità mediante alterazioni casuali delle istruzioni, permettendo l'esplorazione di nuove regioni dello spazio delle soluzioni e contrastando la convergenza prematura verso ottimi locali.

Un aspetto distintivo della programmazione genetica è la presenza di neutralità genetica. Non tutte le istruzioni di un programma influenzano necessariamente il suo comportamento finale; alcune possono risultare temporaneamente inattive ma potenzialmente significative in seguito a successive modifiche evolutive. Questo fenomeno, analogo alla presenza di introni nel DNA biologico, conferisce robustezza al processo evolutivo facilitando la creazione di nuovi programmi anche molto differenti dai genitori.

La peculiarità della programmazione genetica lineare rispetto ad altre metodologie evolutive risiede nella rappresentazione dei programmi come sequenze lineari di istruzioni imperative che operano su registri, analogamente a quanto avviene nei linguaggi di basso livello. Questa struttura lineare differisce sostanzialmente dalla rappresentazione ad albero tipica della programmazione genetica tradizionale, dove le soluzioni sono codificate come strutture gerarchiche di funzioni e terminali.

Nel contesto della LGP, ogni soluzione candidata costituisce un programma autonomo capace di elaborare dati attraverso una serie di operazioni sequenziali. L'evoluzione di tali programmi avviene mediante un processo ciclico che comprende la valutazione dell'efficacia di ciascun individuo, la selezione delle soluzioni più promettenti e la generazione di nuove varianti attraverso operatori genetici.

1.3 Definizione Formale

Dato un insieme non vuoto di operazioni

$$O = \{OP_1, OP_2, \ldots, OP_k\},$$

che possono essere eseguite su m registri virtuali

$$R = \{r_1, r_2, \ldots, r_m\},$$

un programma in Linear Genetic Programming (LGP), detto anche **individuo**, è rappresentato come una sequenza ordinata di n istruzioni imperative, con $n > 0$:

$$\boldsymbol{I} = (\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_n).$$

Ciascuna istruzione $\boldsymbol{c}_i$ è definita dalla tupla

$$\boldsymbol{c}_i = (r_{\text{res}}, OP_j, \boldsymbol{r}_{\text{args}}),$$

dove:

- $r_{\text{res}} \in R$ è il registro in cui verrà memorizzato il risultato;
- $OP_j \in O$ è l'operazione da eseguire;
- $\boldsymbol{r}_{\text{args}} \in R^{\text{arity}(OP_j)}$ è il vettore dei registri passati come argomenti all'operazione, se essa ne richiede. Con $\text{arity}(OP_j)$ si indica l'arietà dell'operazione OP_j.

L'esecuzione di una singola istruzione $\boldsymbol{c}_i$ è formalizzata come:

$$\text{exec}(\boldsymbol{c}_i): \quad r_{\text{res}} = OP_j(\boldsymbol{r}_{\text{args}}).$$

L'esecuzione di un intero programma $\boldsymbol{I}$ è l'applicazione sequenziale delle operazioni che lo compongono:

$$\text{exec}(\boldsymbol{I}): \quad \text{exec}(\boldsymbol{c}_1),\ \text{exec}(\boldsymbol{c}_2),\ \ldots,\ \text{exec}(\boldsymbol{c}_n).$$

L'obiettivo della programmazione genetica lineare è individuare il programma che massimizza (o minimizza) una determinata funzione di fitness.

Ad esempio, dati:

$$(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_h), \quad \boldsymbol{x}_i \in \mathbb{R}^d \quad (d > 0),$$

e gli output desiderati

$$(y_1, y_2, \ldots, y_h), \quad y_i \in \mathbb{R},$$

si può definire una funzione di fitness basata sul Mean Squared Error (MSE) tra l'output atteso y_i e il contenuto del registro r_1 dopo aver caricato i valori del vettore $\boldsymbol{x}_i$ in d registri differenti ed aver eseguito il programma $\boldsymbol{I}$.

La ricerca della soluzione ottimale inizia da una popolazione di programmi generati in maniera casuale, e procede mediante un processo iterativo che ad ogni iterazione, o ciclo evolutivo, prevede:

- **Selezione**: Gli individui con migliore performance (in base alla funzione di fitness) vengono scelti per contribuire alla generazione successiva.
- **Mutazione**: Viene generato un nuovo individuo modificando casualmente una o più istruzioni di un individuo esistente.
- **Crossover**: Vengono creati due nuovi individui scambiando, tra due differenti programmi, un segmento di istruzioni di ognuno.

1.4 Obiettivi della Tesi

Questa tesi si propone di sviluppare un'implementazione efficiente, parallela e multipiattaforma della programmazione genetica lineare in linguaggio C. Gli obiettivi specifici includono:

- Progettare e implementare un framework per LGP che sfrutti il calcolo parallelo attraverso OpenMP.
- Garantire la portabilità del software su diverse piattaforme (Linux, Windows, macOS).
- Confrontare le prestazioni dell'implementazione sviluppata con framework esistenti, in particolare DEAP (Distributed Evolutionary Algorithms in Python).
- Valutare l'efficacia dell'implementazione su problemi standard di sintesi di programmi, utilizzando la suite Program Synthesis Benchmark 2 (PSB2).
- Analizzare le problematiche legate al code bloat e proporre strategie per mitigarle.

Capitolo 2

Stato dell'arte

2.1 Evoluzione della Programmazione Genetica

La programmazione genetica (GP) venne descritta formalmente per la prima volta nel lavoro pionieristico di John Koza nel 1992 [8]. In questa formulazione, Koza, propose un'implementazione in cui ogni programma veniva rappresentato come un albero di istruzioni di un linguaggio funzionale molto simile al Lisp. Questa scelta influenzò profondamente la disciplina, portando allo sviluppo di numerose implementazioni che adottavano la stessa rappresentazione arborea. [10]

Negli anni successivi, la comunità scientifica ha approfondito vari aspetti teorici e pratici della programmazione genetica. Opere fondamentali come "A Field Guide to Genetic Programming" di Poli et al. [13] e "Foundations of Genetic Programming" di Langdon e Poli [9] hanno consolidato le basi teoriche della disciplina, mentre contributi come quelli di Banzhaf et al. [2] hanno ampliato le prospettive applicative.

Con il tempo, sono stati proposti anche diversi approcci innovativi, tra cui la programmazione genetica lineare. A differenza della rappresentazione tradizionale ad albero, nella LGP gli individui vengono rappresentati come sequenze di istruzioni di linguaggio imperativo eseguite su registri, in modo analogo a come i programmi vengono rappresentati nel linguaggio macchina dei computer. Questo approccio, formalizzato nei lavori di Brameier [3] e successivamente approfondito nel libro "Linear Genetic Programming" di Brameier e Banzhaf [4], ha mostrato vantaggi significativi in termini di efficienza computazionale e facilità di implementazione.

2.2 Framework Evolutivi Esistenti

Con la maturazione della disciplina, sono nati molti framework evolutivi open source. Tra i più famosi citiamo lil-gp [15], scritto in ANSI C e fortemente ispirato dal lavoro di Koza, ECJ [10] scritto in Java, supporta tecniche più recenti e complesse di GP [14] e DEAP [5]

scritto in python. Inoltre degno di nota è il linguaggio di programmazione stack-based Push, creato appositamente per essere utilizzato negli algoritmi evolutivi, viene utilizzato in svariati programmi di GP come PushGP, Pushpop [12] e PyshGP [12].

La domanda sempre più crescente di benchmark standardizzati per GP, ha portato alla formulazione della General Program Synthesis Benchmark Suite [6] e, successivamente, della suite aggiornata PSB2, che include vari problemi di coding, tratti da Esercizi di corsi di programmazione e siti web come Code Wars e Advent of Code [6].

2.3 Problematiche Aperte

Nonostante i numerosi framework esistenti abbiano raggiunto livelli di maturità elevati, rimangono alcune criticità:

- **Parallelismo**: I framework attuali, pur essendo efficienti, non sempre sfruttano appieno le capacità offerte dalle architetture multi-core moderne. Questo limita le loro prestazioni in problemi computazionalmente intensivi, dove la parallelizzazione potrebbe portare a significativi miglioramenti.
- **Portabilità**: Alcuni framework sono sviluppati per ambienti specifici, riducendo la loro applicabilità in contesti multipiattaforma. Questa limitazione può rappresentare un ostacolo significativo in progetti che richiedono compatibilità con diverse architetture o sistemi operativi.
- **Leggerezza**: Framework come ECJ e DEAP, pur offrendo funzionalità avanzate e portabilità, richiedono ambienti di runtime pesanti (JVM, interpreti Python) che possono risultare meno performanti o meno adatti a contesti embedded o a basso consumo di risorse.
- **Facilità d'uso**: I framework più complessi, sebbene potenti, possono risultare difficili da integrare o adattare in ambienti di ricerca o applicativi industriali, richiedendo una curva di apprendimento ripida.

2.4 Contributo della Tesi

Il lavoro qui presentato intende colmare tali lacune attraverso lo sviluppo di un framework per Linear Genetic Programming caratterizzato da:

- **Parallelismo nativo**: Utilizzando OpenMP, il framework sfrutta il calcolo parallelo per accelerare le operazioni evolutive (mutazione, crossover, valutazione della fitness), garantendo una migliore scalabilità sui processori multi-core.

- **Multipiattaforma e leggerezza**: Scritto in C, il framework si distingue per un'implementazione leggera ed ottimizzata, con una facile portabilità su diversi sistemi operativi (Linux, Windows, macOS).
- **Semplicità d'uso**: La struttura modulare consente agli utenti di integrarlo in maniera semplice e di adattarlo a differenti problemi, rendendo l'algoritmo accessibile anche a chi non possiede approfondite conoscenze di ingegneria del software.

Con questo approccio, si mira a fornire una soluzione che non solo garantisca performance elevate grazie al parallelismo, ma che sia anche facilmente utilizzabile e adattabile a numerosi scenari applicativi, contribuendo così all'evoluzione degli strumenti per la programmazione genetica lineare.

Capitolo 3

Metodologia

3.1 Architettura dell'implementazione

3.1.1 Operazioni Primitive

Le operazioni disponibili per i programmi sono state rappresentate come struct contenenti informazioni sull'operazione, quali l'arietà o la rappresentazione testuale, e un puntatore al blocco di codice che implementa l'operazione effettiva. L'insieme di operazioni utilizzabili per costruire gli individui è stato definito tramite un array, così da identificare un'operazione tramite la posizione che occupa nell'array. Questa architettura consente di estendere facilmente il set di operazioni disponibili senza modificare la logica evolutiva sottostante, garantendo così la modularità e l'estensibilità del framework.

3.1.2 Rappresentazione degli Individui

L'implementazione proposta in questo lavoro si basa su una rappresentazione degli individui tramite liste di istruzioni, codificate a loro volta in una struttura dati contenente il registro di destinazione, l'indice dell'operazione da eseguire e gli argomenti necessari ad essa. Gli argomenti sono stati implementati mediante una union che consente di gestire operazioni che accettano come argomento sia registri sia valori immediati.

3.1.3 Ambiente di Esecuzione Virtuale

I registri virtuali in cui vengono eseguiti gli individui sono stati implementati tramite un array di double. Il funzionamento dell'ambiente di esecuzione prevede che i dati in input vengano caricati nei registri prima dell'esecuzione di un individuo. Al termine dell'esecuzione, il registro all'indice 0 conterrà l'output del programma, che verrà confrontato con il valore atteso per calcolare il fitness dell'individuo.

3.1.4 Funzione di Fitness

La funzione di fitness adottata è il Mean Squared Error (MSE), calcolato come la media dei quadrati delle differenze tra l'output del programma e l'output atteso:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

dove y_i rappresenta il valore atteso e $\hat{y}_i$ il valore prodotto dal programma per l'i-esimo caso di test.

Questa metrica viene minimizzata durante il processo evolutivo, con valori di MSE più bassi che indicano soluzioni più accurate. L'implementazione include anche un meccanismo di sicurezza che assegna un valore di fitness massimo ('DBL_MAX') ai programmi che producono risultati non finiti o NaN, garantendo che solo soluzioni numericamente stabili vengano considerate nel processo evolutivo.

3.1.5 Parallelizzazione

La parallelizzazione rappresenta un aspetto fondamentale dell'implementazione. Essa è stata realizzata mediante OpenMP per mantenere la portabilità del programma su diverse piattaforme. Questa scelta permette di sfruttare il calcolo parallelo senza vincolarsi a specifiche architetture hardware o sistemi operativi.

La valutazione del fitness, che costituisce tipicamente il collo di bottiglia computazionale negli algoritmi evolutivi, viene parallelizzata attraverso uno scheduling dinamico. Questo approccio consente di distribuire in modo efficiente il carico di lavoro tra i thread disponibili, adattandosi automaticamente a situazioni in cui la valutazione di alcuni individui richiede più tempo rispetto ad altri.

Anche altre operazioni computazionalmente intensive, come le tecniche di breeding e gli algoritmi di ordinamento utilizzati nelle fasi di selezione, beneficiano della parallelizzazione.

3.1.6 Inizializzazione della Popolazione

L'inizializzazione della popolazione è stata implementata attraverso due strategie principali. La prima crea individui con istruzioni e registri selezionati casualmente all'interno del range valido, scartando automaticamente gli individui che producono risultati non finiti (identificati da un MSE pari a 'DBL_MAX'). La seconda è un'estensione della prima, inizializza la popolazione allo stesso modo mantenendo però solo individui non già presenti nella popolazione. Questo secondo metodo si avvale di una versione adattata dell'algoritmo di hashing SipHash [1] per salvare gli ingividui generati in un'HashMap e controllare efficientemente la presenza di duplicati.

3.1.7 Operatori Genetici

Mutazione

L'operatore di mutazione seleziona un segmento casuale del programma e lo sostituisce con nuove istruzioni generate casualmente. La lunghezza del segmento da mutare e del segmento sostitutivo possono differire, consentendo sia l'espansione che la contrazione del programma durante la mutazione.

Crossover

L'operatore di crossover implementa uno scambio a due punti tra programmi genitori. I punti di taglio sono selezionati casualmente in entrambi i genitori, e i segmenti risultanti sono ricombinati per formare due nuovi programmi figli.

Questo approccio permette di combinare efficacemente sottoprogrammi potenzialmente utili proveniente da genitori diversi, facilitando la propagazione di schemi di istruzioni vantaggiosi all'interno della popolazione.

3.1.8 Controllo del Code Bloat

Il code bloat è un fenomeno ben documentato nella programmazione genetica che consiste nella crescita incontrollata della dimensione dei programmi evoluti senza un corrispondente miglioramento del loro fitness. Questo problema si manifesta tipicamente nelle fasi avanzate dell'evoluzione, quando gli individui tendono ad accumulare istruzioni non essenziali o ridondanti, creando strutture sempre più complesse.

Il code bloat non solo aumenta il costo computazionale della valutazione, rallentando significativamente il processo evolutivo, ma può anche ostacolare la generalizzazione e la comprensibilità delle soluzioni trovate. Risulta quindi essenziale implementare strategie efficaci per contrastare questo fenomeno.

Un controllo del code bloat è stato implementato attraverso la funzione `remove_trash`. Essa opera eliminando da ogni nuovo individuo creato le istruzioni che non contribuiscono, direttamente o indirettamente, al calcolo del valore nel registro di output.

L'algoritmo funziona come segue:

1. Viene creato un array di flag, lungo quanto il numero di registri nell'ambiente virtuale, dove solo il bit corrispondente al registro di output viene impostato a 1, indicando che questo registro è "attualmente rilevante" nel calcolo finale.

2. L'algoritmo analizza le istruzioni del programma in ordine inverso (dall'ultima alla prima), identificando quelle che modificano un registro marcato come "attualmente rilevante".

3. Quando viene trovata un'istruzione che modifica un registro rilevante, questa viene preservata nella nuova versione dell'individuo. Il flag del registro di destinazione viene azzerato (poiché il suo valore precedente viene sovrascritto) e tutti i registri utilizzati come argomenti dall'istruzione vengono marcati come "attualmente rilevanti" (poiché il loro valore influenza il risultato finale).

4. Il processo continua fino a esaminare tutte le istruzioni, producendo un nuovo programma che contiene solo istruzioni che contribuiscono, direttamente o indirettamente, al valore finale del registro di output.

Questo approccio garantisce che il programma risultante contenga esclusivamente istruzioni rilevanti, eliminando automaticamente codice ridondante o ininfluente, istruzioni il cui effetto viene sovrascritto o operazioni i cui risultati non vengono mai utilizzati.

3.1.9 Metodi di Selezione

L'implementazione include diversi metodi di selezione, ciascuno con caratteristiche distintive:

- **Elitismo**: Seleziona un numero prefissato di individui migliori (in base al fitness) dalla popolazione corrente per passare direttamente alla generazione successiva. Questo metodo garantisce che le soluzioni migliori non vengano perse durante l'evoluzione.

- **Elitismo percentuale**: Variante dell'elitismo che seleziona una frazione configurabile della popolazione, invece di un numero fisso di individui. Questo approccio offre maggiore flessibilità, adattandosi automaticamente a popolazioni di dimensioni diverse.

- **Selezione a torneo**: Divide la popolazione in gruppi di dimensione stabilita, selezionando da ciascun gruppo l'individuo con il miglior fitness. Questo metodo combina casualità e pressione selettiva, permettendo anche a individui non ottimali di contribuire alle generazioni successive.

- **Selezione a roulette**: Implementa un approccio probabilistico in cui la probabilità di selezione è proporzionale al fitness relativo nella popolazione; poiché il fitness (MSE) deve essere minimizzato, viene applicata un'inversione per convertirlo in un valore da massimizzare.

Inoltre, sono state implementate varianti con fitness sharing, che applicano uno scaling al fitness basato sulla densità nello spazio dei programmi. Questi metodi utilizzano una distanza di edit tra individui per misurare la similarità tra programmi. Questo approccio promuove la diversità genetica penalizzando individui simili tra loro, contribuendo al controllo del code bloat ed a prevenire la convergenza prematura su minimi locali.

3.2 Dataset e Problemi di Test

Per valutare l'implementazione sono stati selezionati i problemi dalla suite PSB2 focalizzati su dati in virgola mobile. Per ciascun problema è stato definito un insieme di istruzioni primitive; ognuno di questi insiemi include le operazioni aritmetiche di base: addizione (+), sottrazione (-), moltiplicazione (*) e divisione sicura (/), quest'ultima implementata per gestire la divisione per zero. Per ogni problema è stato generato dataset di 100 istanze.

3.2.1 Vector Distance

Data una coppia di vettori n-dimensionali $\boldsymbol{x}$ e $\boldsymbol{y}$ composti da numeri in virgola mobile, calcolare la distanza euclidea che li separa nello spazio n-dimensionale.

$$\text{dist}(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{3.1}$$

Nel nostro dataset, i vettori erano a 3 dimensioni e le componenti sono state generate casualmente nell'intervallo $[-100, 100]$. Per questo problema viene inclusa anche l'operazione di radice quadrata $\sqrt{x}$ all'insieme di istruzioni primitive.

3.2.2 Shopping List

Data una lista di k prodotti con prezzi $\{p_1, p_2, \ldots, p_k\}$ e corrispondenti percentuali di sconto $\{d_1, d_2, \ldots, d_k\}$, calcolare il costo totale della spesa dopo aver applicato gli sconti.

$$\text{Costo Totale} = \sum_{j=1}^{k} p_j \cdot \left(1 - \frac{d_j}{100}\right) \tag{3.2}$$

Nel dataset è stata fissata la lunghezza della lista della spesa a 2 articoli, i prezzi $p_j \in [0, 50]$ e gli sconti $d_j \in [0, 100]$. In questo caso è stata introdotta un'operazione specifica per il calcolo dello sconto, definita come $\text{discount}(x, y) = x \cdot (1 - \frac{y}{100})$

3.2.3 Dice Game

Peter possiede un dado a n facce e Colin uno a m facce. Se entrambi lanciano i loro dadi simultaneamente, calcolare la probabilità P che il risultato di Peter sia strettamente maggiore rispetto a quello di Colin.

$$P = \begin{cases} 1 - \frac{m+1}{2n} & \text{se } n > m \\ \frac{n-1}{2m} & \text{se } n \leq m \end{cases} \tag{3.3}$$

Nel dataset, $n, m \in [1, 1000]$. Per questo problema sono state utilizzate le operazioni di minimo $\min(x, y)$ e massimo $\max(x, y)$ tra due valori

3.2.4 Bouncing Balls

Data un'altezza iniziale h_0 da cui una palla viene lasciata cadere, l'altezza h_1 raggiunta dopo il primo rimbalzo, e il numero totale di rimbalzi r, calcolare la distanza totale verticale percorsa dalla palla.

$$d = h_0 + \sum_{j=1}^{r} 2h_0 \cdot \left(\frac{h_1}{h_0}\right)^j \tag{3.4}$$

Il rapporto $\frac{h_1}{h_0}$ rappresenta il coefficiente di rimbalzo della palla. Nel dataset, $h_0 \in [1.01, 100.0]$, $h_1 \in [1.0, h_0 - 0.01]$ e $r \in [1, 20]$. All'insieme di istruzioni è stata aggiunta l'operazione potenza x^y.

3.2.5 Snow Day

Data un'intera quantità di ore t e tre numeri in virgola mobile che rappresentano, rispettivamente, la quantità di neve iniziale s_0, la velocità di caduta della neve v e la proporzione m di neve che si scioglie ogni ora, determinare la quantità di neve s_f presente al termine delle ore indicate.

$$s_f = s_0 + \sum_{j=1}^{t} v \cdot (1 - m)^j \tag{3.5}$$

Ogni ora viene considerata come un evento discreto: prima si aggiunge la neve con velocità v, poi si procede allo scioglimento con proporzione m. Nel dataset, $t \in [0, 20]$, $s_0 \in [0, 20]$, $v \in [0, 10]$ e $m \in [0, 1]$.

3.3 Metriche di Valutazione

Per valutare le prestazioni dell'implementazione proposta e confrontarla con il framework DEAP, sono state utilizzate diverse metriche:

- **MSE (Mean Squared Error)**: Rappresenta la qualità della soluzione trovata. Valori più bassi indicano soluzioni più accurate.
- **Numero di valutazioni**: Indica quante volte è stato calcolato l'MSE di un individuo durante un processo evolutivo. Questa metrica è utile per valutare l'efficienza del processo di ricerca.
- **Tempo di esecuzione**: Misura il tempo impiegato nell'esecuzione di un ciclo evolutivo completo, permettendo di valutare l'efficienza computazionale dell'implementazione.

- **Valutazioni al secondo**: Calcolato come il rapporto tra il numero di valutazioni e il tempo di esecuzione, fornisce un'indicazione diretta della velocità di elaborazione.

Queste metriche sono state utilizzate in modo complementare per ottenere una visione completa delle prestazioni dell'implementazione, considerando sia la qualità delle soluzioni trovate sia l'efficienza computazionale.

Capitolo 4

Risultati

I risultati ottenuti dall'implementazione proposta sono stati confrontati con quelli del framework DEAP su ciascuno dei problemi di test descritti. DEAP è stato eseguito in modalità milti-processing, in modo da sfruttare a pieno la macchina in cui sono stati eseguiti i test ed ottenere un confronto più equo. In questa sezione, vengono presentati in dettaglio i risultati per ogni problema, evidenziando sia gli aspetti prestazionali sia la qualità delle soluzioni trovate.

4.1 Panoramica Generale

In generale, l'implementazione LGP in C ha mostrato un significativo vantaggio in termini di velocità di esecuzione rispetto a DEAP, riuscendo ad eseguire un numero maggiore di valutazioni di fitness nello stesso intervallo di tempo. Tuttavia questo vantaggio prestazionale non è sempre presente; questo accade principalmente a causa del fenomeno del code bloat, che, in un numero ristretto di test, ha influenzato negativamente l'efficienza dell'implementazione proposta.

4.2 Risultati per Problema

4.2.1 Vector Distance

Per il problema Vector Distance, l'implementazione ha mostrato buone capacità di trovare soluzioni accurate. La natura relativamente semplice del problema, che richiede principalmente operazioni aritmetiche di base e l'uso della radice quadrata, ha permesso all'algoritmo di convergere rapidamente verso soluzioni con basso MSE.

4.2.2 Shopping List

Il problema Shopping List ha rappresentato un caso favorevole per l'implementazione proposta. La natura additiva del problema e l'introduzione dell'operazione specifica per il calcolo dello sconto hanno permesso all'algoritmo di trovare rapidamente soluzioni accurate. Questo problema ha evidenziato come l'adattamento dell'insieme delle operazioni primitive alle specifiche esigenze del problema possa migliorare significativamente le prestazioni dell'algoritmo evolutivo.

4.2.3 Dice Game

Il problema Dice Game ha evidenziato una delle limitazioni dell'implementazione attuale: la difficoltà nel gestire efficacemente problemi che richiedono strutture condizionali. La formula matematica che definisce il problema richiede una decisione basata sul confronto tra i valori di n e m, che idealmente verrebbe implementata con un costrutto if-else. Infatti l'implementazione non riesce a trovare la soluzione per questo problema, ma, con l'aggiunta delle operazioni min e max all'insieme delle primitive, ha trovato soluzioni approssimate con un MSE accettabile arrivando anche ad un valore nell'ordine di $10^{-}12$.

4.2.4 Bouncing Balls

Il problema Bouncing Balls ha rappresentato una sfida maggiore per l'implementazione. La sua formulazione, che coinvolge una sommatoria dipendente dal numero di rimbalzi, richiede idealmente l'uso di strutture di controllo come cicli, che non sono direttamente supportate nel modello LGP implementato. L'algoritmo, inoltre, fatica anche a trovare delle soluzioni approssimate al problema, mantenendo un MSE nell'ordine di 10^2.

4.2.5 Snow Day

Analogamente al problema Bouncing Balls, anche Snow Day ha presentato sfide legate alla necessità di strutture di controllo per implementare la sommatoria richiesta dalla formula matematica che risolve il problema. Anche qui l'implementazione non riesce a trovare delle soluzioni approssimate accettabili, anche se l'MSE a fine ciclo evolutivo è mediamente più basso di quello calcolato per Bouncing Balls, con un ordine di grandezza di 10^1.

4.3 Analisi Comparativa

La Figura 4.1 mostra un confronto tra l'implementazione proposta e DEAP in termini di valutazioni di MSE eseguite al secondo per il problema Shopping List. Come si può osservare, l'implementazione in C esegue un numero significativamente maggiore di

valutazioni nello stesso intervallo di tempo, evidenziando il vantaggio prestazionale del linguaggio C e dell'approccio adottato.

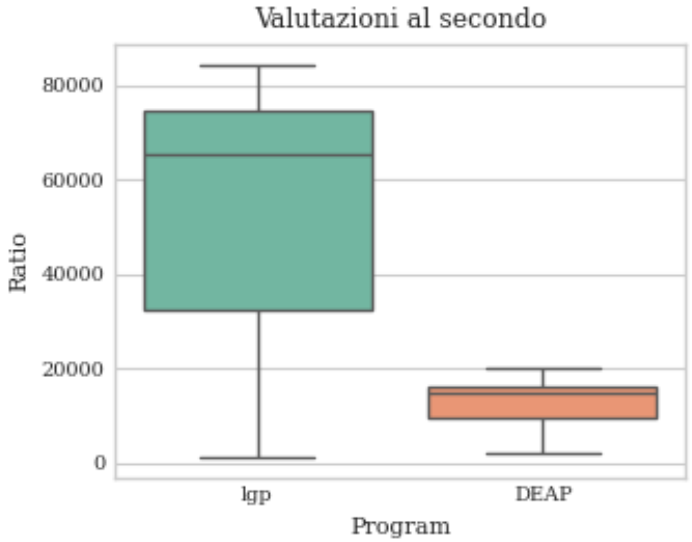

Figura 4.1: Valutazioni di MSE eseguite al secondo per il problema Shopping List

Si può notare, però, che anche l'LGP in C, in certi casi, dimostra performance paragonabili a DEAP. Questo, molto probabilmente, avviene a causa del code bloat che rende le valutazioni di MSE computazionalmente costose e rallenta l'intero processo evolutivo.

In Figura 4.2 sono mostrati vari processi evolutivi, eseguiti con diversi parametri. Dal grafico si evince che, per cicli evolutivi più dispendiosi in termini di tempo, DEAP trova la soluzione metre l'implementazione proposta tende a sbagliare. Questo può suggerire ulteriormente un problema di code bloat.

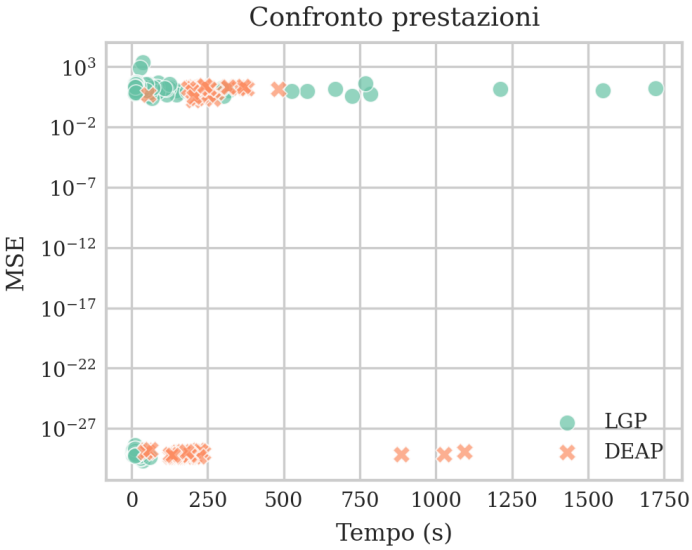

Figura 4.2: MSE rispetto al tempo di esecuzione per il problema Shopping List

L'implementazione proposta risulta, quindi, più performante del framework scritto in python. Nonostante ciò, l'implementazione in C sembrerebbe mostrare una maggiore vulnerabilità al code bloat, il quale in un numero ristretto di test, a causa delle dimensioni eccessive raggiunte dagli individui, ha rallentato l'esecuzione ed ostacolato l'esplorazione efficace dello spazio delle soluzioni.

Capitolo 5

Discussione

I risultati ottenuti offrono spunti interessanti per una riflessione sulle potenzialità e sulle limitazioni dell'implementazione, nonché sulle caratteristiche generali della programmazione genetica lineare.

5.1 Efficienza Computazionale

L'efficienza computazionale rappresenta un aspetto cruciale per l'applicabilità degli algoritmi evolutivi a problemi complessi, dove il costo della valutazione del fitness può essere elevato. In questo senso, l'implementazione proposta offre un contributo significativo, permettendo di eseguire un gran numero di valutazioni per intervallo di tempo.

Tuttavia, è importante sottolineare che l'efficienza computazionale non rappresenta l'unico fattore determinante per il successo di un algoritmo evolutivo. L'efficacia del processo di ricerca, la capacità di evitare minimi locali e la robustezza rispetto a fenomeni come il code bloat sono altrettanto importanti, come evidenziato dai risultati ottenuti.

5.2 Controllo del Code Bloat

Il fenomeno del code bloat emerge come una delle limitazioni dell'implementazione attuale. Nonostante l'introduzione della funzione `remove_trash`, che elimina le istruzioni non influenti sul risultato finale e riduce quindi la lunghezza degli individui, essi, in alcune esecuzioni del programma, tendono comunque a crescere considerevolmente di dimensione durante l'evoluzione.

A differenza dell'approccio proposto in questo lavoro, DEAP implementa un meccanismo di controllo più stringente attraverso l'imposizione di una lunghezza massima che gli individui possono raggiungere durante l'evoluzione. Questo vincolo esplicito previene efficacemente la crescita incontrollata dei programmi, mantenendo costante il costo com-

putazionale massimo della valutazione del fitness, anche nelle fasi avanzate del processo evolutivo.

La sensibilità più alta al code bloat può essere attribuita a diversi fattori:

- La funzione `remove_trash` opera solo sulle istruzioni che non contribuiscono al risultato finale, ma non limita la crescita delle parti effettivamente utilizzate del programma, che possono comunque espandersi attraverso la ripetizione di schemi di istruzioni simili o la creazione di percorsi di calcolo ridondanti.
- Gli operatori genetici implementati, in particolare la mutazione che può sostituire un segmento con uno di lunghezza maggiore, favoriscono naturalmente l'espansione dei programmi in assenza di una pressione selettiva specifica contro la crescita.
- Il fitness basato unicamente sull'MSE non penalizza direttamente la complessità o la dimensione dei programmi, portando a una situazione in cui programmi più complessi ma con lo stesso MSE hanno la stessa probabilità di essere selezionati rispetto a programmi più semplici.

I risultati suggeriscono che un approccio più efficace al controllo del code bloat potrebbe includere:

- L'introduzione di un limite esplicito alla dimensione degli individui, come implementato in DEAP.
- L'adozione di una funzione di fitness sharing che bilanci l'accuratezza della soluzione con la complessità del programma.
- L'implementazione di operatori genetici specificamente progettati per favorire la parsimonia, come proposto in letteratura [11].

La gestione efficace del code bloat rappresenta una sfida importante non solo per l'implementazione proposta, ma per la programmazione genetica in generale, e il bilanciamento tra espressività e parsimonia rimane un'area di ricerca attiva.

5.3 Espressività del Modello

I risultati ottenuti sui diversi problemi evidenziano come l'espressività del modello LGP implementato influenzi significativamente la capacità di trovare soluzioni accurate. Problemi come Vector Distance e Shopping List, che richiedono principalmente operazioni aritmetiche dirette, sono stati affrontati con successo, mentre problemi che beneficerebbero di strutture di controllo come cicli (Bouncing Balls, Snow Day) o condizionali (Dice Game) hanno presentato maggiori difficoltà.

Questa osservazione suggerisce che, per ampliare lo spettro di problemi affrontabili efficacemente, potrebbe essere utile estendere il modello per includere primitive che simulano strutture di controllo o, alternativamente, adottare un approccio ibrido che combini la programmazione genetica lineare con altri paradigmi più espressivi per determinati tipi di problemi.

È interessante notare come la personalizzazione dell'insieme delle operazioni primitive per specifici problemi (come l'introduzione dell'operazione di sconto per il problema Shopping List) abbia migliorato significativamente le prestazioni. Questo evidenzia l'importanza di un'attenta selezione delle primitive in base alle caratteristiche del problema affrontato.

5.4 Scalabilità e Parallelismo

L'approccio parallelo adottato nell'implementazione ha mostrato buoni risultati in termini di scalabilità, permettendo di sfruttare efficacemente le architetture multi-core. Questo aspetto è particolarmente rilevante considerando l'evoluzione dell'hardware moderno, caratterizzato da un numero crescente di core e da architetture parallele sempre più sofisticate.

La parallelizzazione della valutazione del fitness, in particolare, ha permesso di ottenere significativi miglioramenti prestazionali, confermando l'ipotesi che questa rappresenti effettivamente il collo di bottiglia computazionale negli algoritmi evolutivi.

5.5 Confronto con DEAP

Il confronto con DEAP ha evidenziato sia i punti di forza sia le limitazioni dell'implementazione proposta. Da un lato, l'efficienza computazionale del linguaggio C e l'approccio parallelo hanno permesso di ottenere un significativo vantaggio in termini di valutazioni al secondo. Dall'altro, la maggiore vulnerabilità al code bloat in alcuni cicli evolutivi, ha limitato il vantaggio ottenuto in termini di prestazioni.

È importante considerare che il confronto è stato effettuato su problemi relativamente semplici, dove il costo della valutazione della fitness è modesto. In scenari più complessi, dove questa valutazione rappresenta il fattore limitante dominante, il vantaggio dell'implementazione in C potrebbe risultare più determinante.

Capitolo 6

Conclusioni e Sviluppi Futuri

6.1 Conclusioni

Questo lavoro ha presentato un'implementazione parallela e multipiattaforma della programmazione genetica lineare in C, con l'obiettivo di offrire uno strumento efficiente, scalabile e facilmente integrabile per la soluzione automatica di problemi attraverso tecniche evolutive.

I risultati ottenuti hanno evidenziato sia i punti di forza sia le limitazioni dell'approccio proposto:

- L'implementazione in C, combinata con la parallelizzazione tramite OpenMP, ha dimostrato un significativo vantaggio prestazionale rispetto al framework Python DEAP, permettendo di eseguire un numero maggiore di valutazioni di fitness nello stesso intervallo di tempo.
- Il framework ha mostrato buone capacità di trovare soluzioni accurate per problemi che richiedono principalmente operazioni aritmetiche dirette, come Vector Distance e Shopping List.
- L'architettura modulare e la flessibilità nell'estensione dell'insieme delle operazioni primitive hanno permesso di adattare efficacemente l'implementazione a diversi tipi di problemi.
- Il controllo del code bloat è emerso come una delle principali sfide, con l'implementazione attuale che risulta più vulnerabile a questo fenomeno rispetto a DEAP, limitando l'efficacia del processo evolutivo nei cicli più lunghi.
- L'espressività del modello LGP implementato ha mostrato limitazioni nell'affrontare problemi che beneficerebbero di strutture di controllo come cicli o condizionali.

Nel complesso, l'implementazione proposta rappresenta un contributo significativo al panorama degli strumenti disponibili per la programmazione genetica, offrendo un

framework leggero, efficiente e multipiattaforma che può essere facilmente integrato in diverse applicazioni e contesti di ricerca.

La combinazione di efficienza computazionale, parallelismo nativo e portabilità multipiattaforma rende il framework particolarmente adatto per applicazioni che richiedono elevate prestazioni o che operano in ambienti con risorse limitate, dove framework più pesanti come ECJ o DEAP potrebbero risultare meno appropriati.

6.2 Sviluppi Futuri

I risultati ottenuti e le limitazioni evidenziate suggeriscono diverse direzioni promettenti per sviluppi futuri:

- **Miglioramento del controllo del code bloat**: L'implementazione di strategie più efficaci per il controllo del code bloat rappresenta una priorità per migliorare l'efficacia dell'evoluzione. Questo potrebbe includere l'introduzione di limiti espliciti alla dimensione degli individui o l'implementazione di operatori genetici specificamente progettati per favorire la parsimonia.

- **Estensione del supporto a diversi tipi di dati**: Attualmente, l'implementazione supporta principalmente operazioni su numeri a virgola mobile. Un'estensione naturale sarebbe l'inclusione del supporto per altri tipi di dati, come interi, booleani o stringhe, ampliando così lo spettro di problemi affrontabili.

- **Implementazione di primitive per strutture di controllo**: Per migliorare l'espressività del modello, sarebbe utile implementare primitive che simulano strutture di controllo come cicli o condizionali, o integrare approcci come la programmazione genetica automata che permettono l'evoluzione di strutture più complesse.

- **Introduzione di altre metriche di Fitness**: L'implementazione attualmente supporta solamente l'MSE come funzione di fitness e l'introduzione di ulteriori metriche o dare la possibilità all'utilizzatore di definire una metrica di fitness amplierebbe in modo significativo lo spazio dei problemi risolvibili tramite questo approccio

- **Integrazione con tecniche di machine learning**: Un'interessante direzione di ricerca potrebbe essere l'integrazione della programmazione genetica lineare con tecniche di machine learning, utilizzando modelli predittivi per guidare il processo evolutivo o per apprendere automaticamente operatori genetici più efficaci.

- **Estensione a problemi multi-output**: L'implementazione attuale è focalizzata su problemi con un singolo output. Un'estensione naturale sarebbe il supporto per problemi multi-output, dove il programma deve produrre diversi valori correlati.

- **Sviluppo di interfacce utente e strumenti di visualizzazione**: Per facilitare l'utilizzo del programma in contesti didattici o applicativi, sarebbe utile sviluppare interfacce utente e strumenti di visualizzazione che permettano di monitorare e analizzare il processo evolutivo in modo più intuitivo.

Questi sviluppi potrebbero contribuire a migliorare ulteriormente l'efficacia e l'applicabilità del framework, rendendolo uno strumento ancora più versatile per la soluzione automatica di problemi attraverso tecniche evolutive.

In conclusione, la programmazione genetica lineare in C parallela e multipiattaforma rappresenta un approccio promettente per la generazione automatica di programmi, combinando l'efficienza del linguaggio C con la potenza degli algoritmi evolutivi. Nonostante le sfide identificate, i risultati ottenuti suggeriscono che questo approccio possa offrire vantaggi significativi in termini di prestazioni e flessibilità, aprendo la strada a numerose applicazioni in ambiti dove l'efficienza computazionale e la portabilità rappresentano requisiti fondamentali.

Bibliografia

[1] Jean-Philippe Aumasson e Daniel J. Bernstein. *SipHash: a fast short-input PRF*. Cryptology ePrint Archive, Paper 2012/351. 2012. URL: https://eprint.iacr.org/2012/351.

[2] W. Banzhaf. *Genetic Programming: An Introduction*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 1998. ISBN: 9781558605107. URL: https://books.google.it/books?id=1697qefFdtIC.

[3] Markus Brameier. «On linear genetic programming». In: (feb. 2004), p. 272. DOI: 10.17877/DE290R-253.

[4] Markus Brameier e Wolfgang Banzhaf. *Linear Genetic Programming*. English. Netherlands: Springer, 2007.

[5] Félix-Antoine Fortin et al. «DEAP: evolutionary algorithms made easy». In: *J. Mach. Learn. Res.* 13.1 (lug. 2012), pp. 2171–2175. ISSN: 1532-4435.

[6] Thomas Helmuth e Peter Kelly. «PSB2: The Second Program Synthesis Benchmark Suite». In: *2021 Genetic and Evolutionary Computation Conference*. GECCO '21. Lille, France: ACM, 2021. DOI: 10.1145/3449639.3459285. URL: https://dl.acm.org/doi/10.1145/3449639.3459285.

[7] John H. Holland. «Genetic Algorithms». In: *Scientific American* 267.1 (1992), pp. 66–73. ISSN: 00368733, 19467087. URL: http://www.jstor.org/stable/24939139 (visitato il 19/03/2025).

[8] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0262111705. URL: http://www.genetic-programming.com/jkpdf/scjournallong.pdf.

[9] William Langdon e Riccardo Poli. *Foundations of Genetic Programming*. Gen. 2002. ISBN: 978-3-540-42451-2. DOI: 10.1007/978-3-662-04726-2.

[10] Sean Luke. «ECJ then and now». In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '17. Berlin, Germany: Association for Computing Machinery, 2017, pp. 1223–1230. ISBN: 9781450349390. DOI: 10.1145/3067695.3082467. URL: https://doi.org/10.1145/3067695.3082467.

[11] Sean Luke e Liviu Panait. «A Comparison of Bloat Control Methods for Genetic Programming». In: *Evolutionary Computation* 14.3 (2006), pp. 309–344. DOI: 10.1162/evco.2006.14.3.309.

[12] Edward Pantridge e Lee Spector. «PyshGP: PushGP in Python». In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '17. Berlin, Germany: Association for Computing Machinery, 2017, pp. 1255–1262. ISBN: 9781450349390. DOI: 10.1145/3067695.3082468. URL: https://doi.org/10.1145/3067695.3082468.

[13] Riccardo Poli, William Langdon e Nicholas Mcphee. *A Field Guide to Genetic Programming*. Gen. 2008. ISBN: 978-1-4092-0073-4.

[14] Garnett Wilson, Andrew Mcintyre e Malcolm Heywood. «Resource Review: Three Open Source Systems for Evolving Programs–Lilgp, ECJ and Grammatical Evolution». In: *Genetic Programming and Evolvable Machines* 5 (mar. 2004), pp. 103–105. DOI: 10.1023/B:GENP.0000017053.10351.dc.

[15] Douglas Zongker e Bill Punch. *lilgp 1.01 User's Manual*. Rapp. tecn. USA: Michigan State University, 1996. URL: http://citeseer.ist.psu.edu/cache/papers/cs/14524/ftp:zSzzSzgarage.cps.msu.eduzSzpubzSzGAzSzlilgpzSzlilgp1.02.pdf/lil-gp-user-s.pdf.