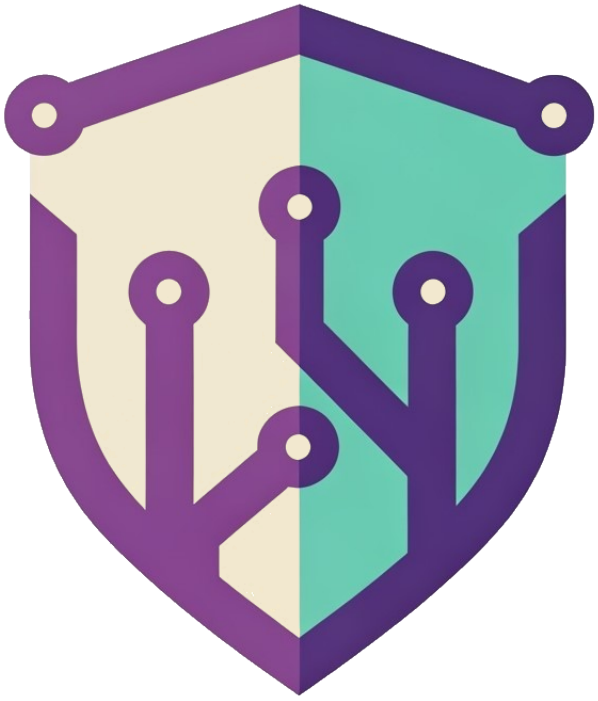

hAIcker

`https://github.com/Zimuch/hAicker`

Silvana De Martino, Simon Carbone

February 11, 2025

Contents

Introduzione

Al giorno d'oggi, l'intelligenza artificiale sta compiendo progressi significativi, diventando uno strumento sempre più accessibile nei contesti più disparati. Le aziende stanno esplorando modi innovativi per sfruttarne il potenziale, cercando soluzioni avanzate oltre la semplice implementazione.

L'introduzione di sistemi basati su AI sta trasformando numerosi settori, dall'industria alla sanità, fino al mondo del software engineering. In quest'ultimo ambito, l'AI si rivela un prezioso alleato nell'ottimizzazione dei processi, nella riduzione degli errori e nel supporto decisionale. Tuttavia, nonostante le sue potenzialità, non può ancora sostituire completamente la figura del progettista, il quale deve integrare l'intelligenza artificiale con esperienza e conoscenza tecnica.

Lo scopo di hAIcker è fornire un supporto decisionale agli ingegneri software fin dalle prime fasi della progettazione. Il progetto prevede l'impiego di hAIcker immediatamente dopo la raccolta dei requisiti, aiutando a definire problemi, costi e obiettivi del sistema. Questa fase è cruciale, poiché una valutazione errata delle risorse o delle vulnerabilità può compromettere la sicurezza e la robustezza del software finale.

L'algoritmo elabora dati preliminari quando le informazioni scambiate con il cliente sono ancora a livello astratto. Questo aspetto è fondamentale, poiché spesso chi commissiona un progetto ha una conoscenza limitata delle risorse necessarie, sia economiche che temporali. Una scarsa attenzione alla sicurezza informatica, inoltre, può avere conseguenze critiche. L'algoritmo hAIcker mira a fornire a clienti e progettisti una stima realistica della qualità finale del progetto in relazione ai costi e agli obiettivi. L'output dell'algoritmo tiene conto del rapporto tra costi e resistenza del sistema agli attacchi informatici. La sicurezza è oggi un aspetto imprescindibile nello sviluppo software, dato l'aumento delle minacce informatiche. Un'allocazione errata delle risorse potrebbe lasciare aperte vulnerabilità critiche, mettendo a rischio il sistema e i dati degli utenti.

Partendo da un numero predefinito di risorse e da un insieme di punti vulnerabili nel software, l'algoritmo suggerisce la strategia ottimale per la loro distribuzione. Questo permette agli ingegneri di prendere decisioni basate su analisi e simulazioni, anziché su mere ipotesi.

Descrizione del problema

2.1 Obiettivi

hAIcker mira a produrre codice robusto, partendo dalla suddivisione accurata delle risorse, che includono costi, tempo e personale, per ridurre in maniera significativa il rischio di incursioni hacker o altri danni che potrebbero compromettere l'integrità del sistema. Parallelamente, hAIcker si propone di ridurre i costi complessivi del progetto. Grazie a una valutazione preventiva basata su analisi qualitative e simulazioni, il sistema consente di individuare soluzioni che portino a un notevole risparmio in termini di tempo e denaro, elementi fondamentali per evitare sprechi e ottimizzare le risorse disponibili. Un ulteriore aspetto su cui il progetto pone particolare attenzione è la priorizzazione dei punti sensibili del software. Non tutti i componenti di un sistema rivestono la stessa importanza; hAIcker analizza le aree critiche per identificare quelle che richiedono un intervento prioritario, garantendo così che le risorse vengano allocate in maniera mirata e strategica. In questo modo, si assicura che le parti più vulnerabili siano adeguatamente protette, contribuendo a rendere il sistema complessivamente più sicuro e affidabile.

2.2 Analisi del problema

In un sistema software, la protezione delle componenti software è una questione critica, che può determinare interamente l'andamento della progettazione. La distribuzione delle risorse di difesa può influenzare direttamente la sicurezza e le prestazioni globali del sistema, e determinare tale distribuzione preventivamente può snellire di molto i tempi di sviluppo e progettazione.
Ogni parte del sistema ha una diversa importanza relativa e una diversa vulnerabilità che devono essere prese in considerazione per allocare le risorse in modo ottimale. Tuttavia, l'allocazione non può essere arbitraria, poiché è necessario rispettare vincoli sulle risorse disponibili e bilanciare le priorità tra sicurezza e costi.
Gli elementi principali del problema includono:

1. La necessità di rappresentare il sistema software in modo che sia possibile analizzare e ottimizzare la distribuzione delle risorse di difesa;

2. La definizione di un metodo per quantificare l'importanza relativa e la vulnerabilità delle varie parti del sistema;

3. La gestione delle risorse disponibili, vincolate tra un minimo e un massimo, per garantire flessibilità senza superare i limiti imposti;

4. La necessità di bilanciare due obiettivi principali:

    - Minimizzare i danni potenziali alle componenti più vulnerabili del sistema.
    - Minimizzare i costi complessivi legati all'uso delle risorse.

Inoltre, il problema presenta una componente di complessità aggiuntiva poiché le risorse allocabili devono tenere conto di una serie di vincoli e priorità che variano in funzione dello stato del sistema e delle sue esigenze. Questo rende necessaria una strategia di ottimizzazione in grado di adattarsi a queste dinamiche complesse, garantendo al contempo efficienza e robustezza.

2.3 Specifica PEAS

I PEAS (Performance measure, Environment, Actuators, Sensors) sono un framework utilizzato per definire in modo sistematico il comportamento di un agente intelligente, in particolare nei contesti dell'intelligenza artificiale. Ogni componente del PEAS descrive un aspetto cruciale del sistema in cui l'agente opera:

1. **Performance** : sono definite dal valore dei danni relativi inflitti. Per danni relativi si intende la somma del valore dei danni calcolato in relazione al ranking della zona;

2. **Enviroment** : l'ambiente in cui opera l'agente è lo stato iniziale di un sistema software in fase di sviluppo, quindi un campo ancora teorico. Dal punto di vista pratico l'ambiente consisterà di una array, di cui l'indice corrisponderà il ranking e il contenuto della cella alla vulnerabilità della zona in questione, e da un valore numerico variabile, il numero di risorse impiegate per quel determinato punto;

3. **Actuators**: azioni difensive del sistema, si intendono in questo caso tutti quegli elementi che permettono di effettuare una miglior difesa del sistema, come firewall, password, crittografia etc. Dal punto di vista pratico dell'algoritmo agisce assegnando ad ogni punto del sistema un numero dato di risorse;

4. **Sensors** : l'agente percepisce l'ambiente prendendo lo stato attuale e valutando il posizionamento delle risorse.

2.4 Caratteristiche d'ambiente

Le caratteristiche d'ambiente descrivono le condizioni operative dell'agente e il suo processo decisionale.

1. **Completamente osservabile** : l'agente sa in qualunque momento quali sono i ranking delle varie parti del sistema e il numero di risorse massimo a sua disposizione;

2. **Discreto** : Il numero di percezioni dell'agente è limitato;

3. **Stocastico** : l'evoluzione è influenzata da elementi casuali generati dagli algoritmi di crossover e mutation;

4. **Singolo agente** : per quanto vengano percepiti degli attacchi, l'ambiente si focalizza unicamente su software al fine di minimizzare il danno subito;

5. **Sequenziale** :la soluzione finale è data dal raffinamento sequenziale delle fasi di definizione dell'algoritmo;

6. **Dinamico**: alcuni aspetti si raffinano in base all'evoluzione dell'algoritmo.

Altri Algoritmi

3.1 Algoritmi di ricerca non informata ed informata locale

Algoritmi di ricerca non informata, quali ad esempio la ricerca in ampiezza (Breadth-First Search) e la ricerca in profondità (Depth-First Search), sono stati scartati a causa di una loro principale caratteristica: la staticità. Il problema che stiamo andando a discutere in questo documento, infatti, è in continua evoluzione e cambiamento, il che richiede necessariamente un continuo adattamento del codice al presentarsi di nuove difficoltà e nuove priorità da parte di stakeholder o implementatori. Questo aspetto avrebbe portato ad una bassa possibilità di personalizzare il codice e quindi una bassa adattabilità delle soluzioni a contesti diversi, richiedendo tempistiche dilatate per il tuning dei parametri e progettisti specializzati per interagire con il codice stesso.
Un'altra caratteristica che ha portato a evitare la scelta dell'impiego di questa tipologia di algoritmi, è la loro complessità, che risulta essere molto alta, sia in termini di tempo che di spazio (ad esempio, il tempo di convergenza in problemi di ottimizzazione complessi o la gestione di grandi popolazioni). Nonostante si sappia che, con le giuste accortezze e utilizzando le opportune variazioni, questo problema possa essere efficacemente aggirato, si è comunque preferito evitare di incorrere in questi tipi di difficoltà. Immaginando, infatti, un contesto di utilizzo reale, la complessità di alcuni di questi algoritmi avrebbe reso il loro utilizzo poco efficace e poco funzionale, portando aziende e progettisti a scartare l'idea di impiegarlo.

Lo stesso discorso si può sostenere per la ricerca informata, che prevede l'uso di una singola funzione obiettivo, anch'essa poco flessibile in un contesto in cui gli obiettivi possono essere descritti da più funzioni. Di conseguenza, sia la ricerca non informata sia quella informata presentano limitazioni che le rendono inadatte a rispondere efficacemente alle esigenze di un sistema in continua evoluzione.

3.2 Teoria dei giochi/Ricerca con avversari

La ricerca con avversari è stata, durante il periodo di ideazione del progetto, la scelta che sembrava essere più valida. La possibilità di tradurre un simile problema da un contesto reale ad un contesto di gioco, permettendo un'astrazione che ne favorisse lo studio e l'impiego successivo, sembrava essere la scelta migliore.
L'idea principale era quella di utilizzare l'algoritmo MINIMAX e andare poi ad analizzare il suo comportamento, comparandolo con i risultati e le performance ottenute a seguito di una potatura Alpha-Beta. La teoria dei giochi trova infatti applicazione in scenari come quello proposto. Sistema ed hacker avrebbero ritrovato i loro ruoli naturali nelle figure del difensore e dell'attaccante.
Il primo dal lato del mini, il cui obiettivo principale sarebbe stato quello di andare a minimizzare il valore dell'attacco inflitto, e il secondo dal lato del max, che avrebbe avuto come scopo quello di massimizzare i danni inflitti. I due punti di vista sarebbero stati considerati con uguale attenzione e si sarebbe andati a snocciolare il problema partendo dal punto di vista dell'attaccante come spesso accade in contesti di cybersecurity.
L'impiego della teoria dei giochi risultava quindi particolarmente interessante, poiché consentiva di rappresentare la realtà in esame utilizzando un modello familiare e ben consolidato, tipico dei giochi strategici.
Cosa quindi ha portato alla decisione di non utilizzare questa tecnica? La rigidità della teoria dei giochi e la poca competenza sul campo. In particolare l'idea di valutare l'ottimizzazione di più di un obiettivo avrebbe portato ad una complessità troppo alta del progetto finale.
La teoria dei giochi si rivela infatti ideale al fine di descrivere situazioni competitive con obiettivi chiari ed un'unica soluzione. Un sistema di sicurezza è caratterizzato da obiettivi multipli e vulnerabilità distribuite in maniera non uniforme. Vanno considerati anche tutti gli aspetti all'apparenza marginali, come i costi e le priorità date dal cliente, che però nello sviluppo di un sistema possono fare la differenza. In sostanza questa metodologia di approccio non si sarebbe prestata bene al problema e potenzialmente avrebbe condotto ad un'eccessiva semplificazione delLO STESSO rispetto all'idea iniziale.
Per concludere su questo aspetto possiamo dire che la teoria dei giochi quindi, per quanto si presti bene alla considerazione di ambiti reali in cui è possibile individuare due attori in conflitto tra loro, non avrebbe permesso di focalizzarsi in maniera opportuna su tutti gli aspetti desiderati.

3.3 Soluzione del problema : Algoritmo genetico

La motivazione dietro la scelta dell'algoritmo genetico è quindi chiara: la capacità d'adattamento di questo tipo di algoritmi è la più congeniale al problema così come lo si è proposto. Si va ora definendo gli aspetti fondanti dei GA e la maniera in cui si è deciso di applicarli al contesto in essere e alla modalità con cui si è rappresentato il problema.

3.4 Codifica degli individui

Un individuo, prima di poter essere analizzato algoritmicamente, deve essere codificato, e cioè rappresentato in una struttura dati utilizzabile per le varie operazioni che saranno necessarie ai fini dello svolgimento dell'algoritmo. Per questo problema si è scelto di optare per un individuo estremamente semplice che però potesse contenere i valori necessari ad estrarre tutti gli altri con semplici calcoli matematici: e cioè un array di interi.
Tale array contiene:

- **Ranking** : il valore della cella i-esima corrisponde al ranking i-esimo del elemento del sistema. Il ranking non fa riferimento ad un elemento fisso, infatti non viene esplicitato il nome dello stesso, ma si lascia agli utilizzatori la definizione dello stesso (magari tramite apposita documentazione) in base alla realtà in cui lo si sta adoperando.

- **Risorse allocate** : ogni cella conterrà un valore intero positivo, che non potra scendere al di sotto di un numero variabile deciso sulla base della definizione di un vincolo apposito. Per quanto riguarda il numero massimo di risorse allocabili per cella non si è voluto fornire un vincolo esplicito.

- **Risorse allocabili** : la cella 0 non è considerabile all'interno delle operazioni che verranno successivamente esposte ed impiegate, in quanto restituirebbe un ranking di valore 0 che, invece di impattare in modo significativo, andrebbe ad annullare il valore proprio del primo elemento, che dovrebbe essere quello più importante in termini di distribuzione delle risorse. Per questo la cella 0 non sarà intaccata dai cambiamenti evolutivi dell'algoritmo, il valore contenuto rappresenterà la soglia massima di risorse allocabilli, e non sarà eccedibile nella distribuzione.

Inoltre, nonostante non facciano parte della codifica in termini di strutture dati del sistema, sembra conveniente inserire in questa parte la definizione delle seguenti componenti fondamentali del sistema:

- **Danni potenziali**: avendo scartato la teoria dei giochi, non è stato possibile considerare un reale danno. I danni potenziali hanno costituito un'ottima soluzione, permettendo di valutare la gravità che un attacco avrebbe potuto infliggere al sistema tenendo in consideranzione risorse allocate e ranking;
- **Vulnerabilità** : La vulnerabilità è un concetto derivato dal ranking, che dà la percezione di quanto un sistema sia "a rischio" o delicato.

3.4.1 Vincoli

I vincoli definiti in funzione della soluzione sono pochi, seppur molto stringenti. Il primo riguarda la non modificabilità della prima cella dell'array. Questa infatti rappresenta le risorse messe a disposizione da coloro che si occupano di definire le specifiche del sistema, ed è quindi un valore che deve rimanere invariato nel corso di processi come mutazione e crossover.
Il secondo riguarda sempre le risorse, ma si espande a tutto l'array. Infatti, questo determina che nell'assegnazione delle risorse a tutto il sistema, la somma di queste non debba mai superare il valore definito nella cella zero.
l'ultimo vincolo ha a che fare con lo stesso argomento dei due precedenti: le risorse. In questo caso però questo definisce una regola sul numero minimo di risorse assegnabili. Immaginando di fatti un sistema reale, è ben comprensibile che l'assenza totale di risorse in una qualunque parte del sistema potrebbe compromettere l'integrità totale dello stesso. Di conseguenza, quest'ultimo vincolo ha la funzione di evitare il presentarsi di questo problema, andando a definire 20 come valore minimo delle risorse da assegnare ad ogni cella.

3.5 Funzione di fitness

Le funzioni di fitness sono fondamentali, ma prima di scendere nei dettagli c'è bisogno nel contesto in essere di fare un passo indietro. Come detto all'inizio della trattazione del problema, si è deciso di lasciare al lato computazionale la stragrande maggioranza della complessità. Prima di poter procedere alla definizione della funzione di fitness è necessario che vengano esplicitate in termini matematici alcune caratteristiche chiave del problema definite già nel paragrafo 3.4.

3.5.1 Primo obiettivo

Il primo obiettivo è caratterizzato dai valori di vulnerabilità e danno, definiti come segue.

1. **Vulnerabilità**:
$$VN_i = \lambda \cdot \frac{n}{r_i}$$
VN_i rappresenta la vulnerabilità della i-esima cella, quindi quanto la componente associata alla cella i-esima sia soggetta a causare danni d'importanza crescente, λ è una costante che normalizza il valore della funzione, r_i è il ranking della cella i, quindi il valore di i, n è il numero totale di celle nel sistema, corrisponde alla lunghezza dell'array. Il valore VN_i definisce la vulnerabilità della singola cella in assenza di risorse allocate.

2. **Variazione della vulnerabilità** - a seguito dell'impiego di risorse - :
$$V_i = \lambda \cdot \frac{n}{r_i \cdot \sqrt{a_i}}$$
Il valore $\sqrt{a_i}$ indica il decrescere dell'impatto delle risorse sulla vulnerabilità del sistema stesso. Si suppone che, oltre un determinato valore numerico, le risorse impiegate inizino ad avere un impatto di molto ridotto. Questa seconda formula è intesa come un aggiornamento della prima e permette di valutare anche come il numero di risorse impiegate possa andare a causare variazioni in un valore tanto delicato. Questa funzione è il primo momento in cui gli elementi introdotti finora iniziano a interagire tra loro assumento maggior senso nel contesto di applicazione. Risulta essere anche più completa rispetto alla precedente, di fatti unicamente questa seconda tipologia verrà impiegata nella definizione della funzione di fitness.

3. **Danni potenziali**:
$$D_i = \frac{V_i}{r_i \cdot a_i}$$
A r_i corrisponde il ranking, a_i rappresenta il numero di risorse impiegate nella cella ed V_i la vulnerabilità della cella stessa. Questa relazione evidenzia come il danno sia influenzato dall'importanza relativa della cella (espressa dal ranking) e dalla vulnerabilità calcolata, fornendo una misura bilanciata dell'impatto delle risorse assegnate. Il ragionamento si basa sul fatto che, all'aumentare del ranking, quindi al decrescere del valore di i, il danno potenziale cresce, ma una maggiore allocazione di risorse a_i riduce V_i, riducendo il danno complessivo.

Da tutti questi valori calcolati si ottiene un unico valore, quello dei danni totali, che corrisponde anche al valore che si vuole minimizzare, quindi al primo obiettivo da raggiungere.

$$D_t = \sum_{i=1}^{n} D_i$$

3.5.2 Secondo obiettivo:

Andando avanti nell'analisi del problema, cercando di ragionare in relazione ad un contesto quanto più realistico possibile, non si è potuto non considerare come obiettivo quello della minimizzazione dei costi:

1. **Costo relativo**: Si può osservare che questo punto è facilmente ottenibile a seguito della sottrazione tra il valore delle risorse utilizzate e il numero totale di risorse disponibili, che ricordiamo essere definito nella cella 0 dell'array. Tutto questo viene poi moltiplicato per un valore λ_2 che normalizza il risultato della funzione rispetto agli altri obiettivi. La funzione di costo presenta la seguente forma:

   $$C = \lambda_2 \cdot (A - \sum_{i=1}^{n} a_i)$$

   Dove valori di C più alti indicano un utilizzo efficiente delle risorse.

3.5.3 Terzo obiettivo:

Strettamente legato al precedente, il terzo obiettivo ha una complessità più alta in termini di calcolo, e riguarda la corretta distribuzione delle risorse allocate. Tenta di sfruttare nel migliore dei modi una quantità di risorse anche inferiore a quelle effettivamente rese disponibili.

Deriva dalla necessità di bilanciare l'allocazione delle risorse in base alla rilevanza delle celle e alla criticità delle aree da proteggere. La distribuzione pesata, che rappresenta questo equilibrio, considera sia il contributo proporzionale delle risorse assegnate sia il ranking associato alle zone sensibili. Prima di definire la distribuzione in termini più tecnici però è necessario introdurre un ulteriore elemento:

1. **Peso della risorsa**:

   $$P_i = \frac{a_i}{\sum_{k=1}^{n} a_k}$$

   Con questa funzione si va a definire un rapporto tra le risorse effettivamente usate per una singola cella rispetto al totale impiegato. Come visto sopra, questa funzione è d'importanza fondamentale per calcolare il terzo obiettivo del nostro sviluppo.

2. **Distribuzione pesata**: Adesso possiamo definire la distribuzione pesata:

$$W = \lambda_3 \cdot \frac{\sum_{i=1}^{n} P_i \cdot \frac{a_i}{r_i}}{\sum_{j=1}^{n} V_j}$$

Maggiore sarà il valore ottenuto da questa funzione, migliore sarà il rapporto tra la vulnerabilità e la distribuzione delle risorse effettivamente impiegate. Un valore più alto determinerà sempre infatti un basso valore di V ed un più alto valore di P. La moltiplicazione tra P_i e $\frac{a_i}{r_i}$ serve per attribuire al valore finale della funzione un peso che sarà determinato principalmente dal valore di r. Il tutto viene poi moltiplicato per una costante λ_3 che normalizza il valore della funzione rispetto agli altri obiettivi. Questo permetterà di ottenere un valore in definitiva valido per la considerazione della corretta distribuzione delle risorse definito un certo valore di ranking.

3.5.4 Fitness

Avendo definito il problema come multi obiettivo, ora bisogna determinare le metodologie sulla base delle quali andare a definire un'unica soluzione che le consideri tutte. La metodologia che si è considerata essere la più appropriata è l'impiego di una funzione combinata.

$$Fitness = \omega_1 \cdot C + \omega_2 \cdot W - \omega_3 \cdot D_t$$

In questo calcolo ω_1 e ω_2 sono parametri che bilanciano l'importanza tra massimizzare il valore di costo C, quindi le risorse rimanenti, e massimizzare la distribuzione pesata W, mentre il valore ω_3 regola l'importanza di minimizzare i danni rispetto agli altri obiettivi. Questi valori devono essere definiti dall'utente in base agli obiettivi del sistema. La funzione di fitness ha quindi il chiaro obiettivo di venir massimizzata, ottenendo un buon numero di risorse rimanenti e valore di distribuzione a parità con il decremento dei danni possibili.

3.6 Selezione degli individui - Algoritmo K-Way Tournament

Il primo punto da indicare è chiaramente la dimensione della generazione. Per quanto il valore ottimale sia un argomento da tralasciare fino alla parte del documento dedicato alle performance, un valore troppo piccolo, procedendo per ragionamento empirico, determinerebbe una diversità troppo bassa all'interno delle soluzioni pro-

poste, quindi il numero di individui della popolazione è stato posto inizialmente ad un valore di **50** individui.

La selezione della prima generazione dovrà tenere conto di due aspetti: la necessità di essere scalabile, permettendo la facile generazione anche di un alto numero di individui, e il rispetto dei vincoli stringenti definiti.

Terminata questa breve trattazione di aspetti però fondamentali, è bene passare a discutere dell'algoritmo che si impiegherà per la selezione degli individui e alle motivazioni che si celano dietro questa decisione.

A seguito di un'attenta valutazione di vantaggi e svantaggi si è deciso di adottare un algoritmo di selezione di tipo **K-Way Tournament**. L'algoritmo, a seguito della scelta di un numero K, che verrà posto per iniziare a 10 ma il cui valore reale potrà essere soggetto a cambiamenti in fase di osservazione delle performance, ridurrà il rischio di introdurre soluzioni non ammissibili. Il K-Way Tournament è infatti un algoritmo che seleziona le generazioni successive sulla base di tornei, di cui unicamente i vincitori vanno avanti nel processo evolutivo partecipando alla formazione della seguente generazione. Tale vittoria è determinata dal valore della funzione di fitness.

Lo svantaggio principale che però questo algoritmo porta con sé è quello legato alla mancanza di diversificazione degli individui. Questo tipo di svantaggio, seppur possa sembrare essere banale, porta in realtà a problemi di subottimalità dell'algoritmo, che potrebbe finire per non considerare soluzioni molto valide al fine di risolvere i problemi proposti. Un individuo meno promettente a primo impatto potrebbe celare un'ottimizzazione di uno degli obiettivi che un individuo apparentemente più valido invece non comporta. A questo limite è chiaramente sensibile la soluzione proposta, in quanto vengono valutati tre obiettivi di cui, l'ottimizzazione di uno non è direttamente correlata all'ottimizzazione dell'altro, Nonostante ciò si è deciso comunque di adoperare questa scelta, andando a mitigare questa problematicità lavorando con gli algoritmi di crossover e mutation, che quindi andranno a diminuire l'impatto della limitazione del K-Way Tournament.

Il campo di possibile applicazione reale dell'algoritmo necessita di quanta più affidabilità possibile. I vincoli che già sono presenti e quelli che potrebbero essere nel tempo richiesti sono estremamente stringenti e il mantenimento di una base solida

di individui è cruciale per il successo dell'algoritmo a lungo termine. L'affidabilità delle soluzioni è pressocchè il punto focale dell'algoritmo stesso, anche se questo significa sacrificare la diversità in parte. Il controllo sulla popolazione degli individui che partecipa alla generazione della popolazione successiva è quindi di fondamentale importanza e permette di avere un controllo stretto sulla qualità delle soluzioni che si andranno potenzialmente a generare. Si è quindi fatto null'altro che un trade-off tra i vantaggi e gli svantaggi dell'algoritmo in essere.

3.7 Crossover - Algoritmo K-point Crossover

L'algoritmo scelto per il crossover è il ***K-point Crossover***. In questo approccio, vengono individuati K punti casuali all'interno del cromosoma e, successivamente, i segmenti compresi tra questi punti vengono alternati tra i due genitori per creare i figli.

Inizialmente, si era considerato l'adozione di un *Two-point Crossover*, ma questo approccio non riusciva a soddisfare adeguatamente la necessità di introdurre diversità nel processo evolutivo, che risultava fortemente limitata dall'algoritmo di selezione scelto. Il K-point Crossover, al contrario, offre una maggiore flessibilità: il numero di tagli può essere adattato alle specifiche necessità del problema. Un numero maggiore di punti di crossover è indicato quando è necessario introdurre una maggiore diversità nelle soluzioni, mentre un numero moderato di punti è più adatto per problemi che richiedono di mantenere il rispetto dei vincoli, pur introducendo un minimo livello di diversità.

Nel caso in questione, un numero moderato di K è stato scelto per bilanciare la qualità delle soluzioni con la necessità di esplorare nuove combinazioni. Questo approccio consente di garantire la qualità senza appesantire il processo con calcoli complessi per la validazione delle soluzioni. Ad ogni passaggio, il crossover introduce un fattore di diversità che permette di esplorare soluzioni alternative, potenzialmente più ottimali, rispetto a quelle inizialmente identificate come le migliori.

Il valore di K per l'algoritmo in questione è stato fissato inizialmente a 4, ma si prevede di testare anche un valore di $K = 3$ per valutare l'effetto di una leggera variazione sul comportamento dell'algoritmo.

In conclusione, l'adozione del *K-point Crossover* si è rivelata la scelta migliore, poiché, con un numero adeguato di punti di taglio, l'interazione tra i segmenti dei genitori è sufficientemente alta da introdurre variabilità nelle soluzioni, ma non così elevata da compromettere l'integrità dei blocchi genetici ottimali. Questo approccio permette di ottenere un buon compromesso tra la diversità e la stabilità delle soluzioni, favorendo l'esplorazione senza compromettere i vincoli critici. Si conclude

questo paragrafo indicando il valore del crossover rate che è di 0.85, valore scelto sulla base della letteratura studiata, che sottolinea come valori vicini a quello definito portino generalmente ad un miglioramento delle performance, ma non si esclude la possibilità di effettuare un tuning più fine in fase di analisi delle performances.

3.8 Mutation - Algoritmo di Mutation Adaptive e Scramble

La mutazione è un'operazione, in questo contesto, che serve per introdurre varietà nella popolazione. Nel contesto in essere è quindi di particolare importanza, sia nelle parti iniziali per garantire una maggiore esplorazione delle varie possibili soluzioni, ma anche in fasi più avanzate del progetto per evitare che le scelte prese in precedenza, in particolare quella legata all'algoritmo di selezione, portino la soluzione a convergere troppo velocemente. La convergenza prematura è di fatto un problema che può portare a diverse conseguenze tra cui l'ottenimento di una soluzione che, seppur buona, non è quella ottimale, oppure la sospensione dei miglioramenti. Per andare a sopperire quindi a questa problematicità l'algoritmo di mutazione che si è deciso di adottare è l'algoritmo ***Adaptive***.

L'algoritmo di mutazione Adaptive ha la caratteristica di modificare nel corso del processo la probabilità di mutazione. Questo permette di adattare dinamicamente il livello di esplorazione dello spazio delle soluzioni possibili, evitando blocchi prematuri e convergenza troppo rapida dell'algoritmo. La realtà d'interesse dell'algoritmo è notoriamente in continuo sviluppo ed evoluzione, in un sistema potrebbero esserci delle modifiche che potrebbero portare all'aggiunta di nuove componenti del sistema ed ad un eventuale aumento delle celle dell'array. Questo cambiamento non costituirebbe un problema in quanto l'algoritmo di mutazione sarebbe in grado di adattarsi senza necessitare di ulteriori verifiche, o per lo meno che non siano eccessivamente time-consuming. Parametri fondamentali per il corretto funzionamento di questo tipo di mutazione sono:

1. Soglia di fitness: va definita una soglia di fitness che indichi di quanto, tra una generazione e l'altra, deve migliorare il valore. Nel caso di questo progetto la soglia di miglioramento minima che di generazione in generazione ci deve essere per non definire un aumento della mutation rate è del 2 %

2. Tasso di mutazione adattivo: ogni volta che la condizione di sopra si presenterà, la percentuale di mutation rate aumenterà di 2. Questo stesso valore verrà sottratto quando ci sarà un miglioramento più netto tra i valori della funzione

di fitness precedente rispetto a quelli della successiva.

Questi due parametri che si è deciso di inserire servono a migliorare la definizione della funzione e l'andamento dell'algoritmo, pur non essendo standard si è preferito considerarli ai fini di ottimizzare i risultati ottenuti.

Per quanto riguarda la mutation **Scramble**, si è deciso di adoperarla congiunta all'algoritmo Adaptive perchè il funzionamento dello scramble prevede un mischiaggio dei valori già presenti nell'individuo, senza andare ad inserire valori nuovi o da altri array che potrebbero portare alla generazione di soluzioni inammissibili. Questo permette di evitare di inserire controlli relativi al vincolo legato al numero massimo e minimo di risorse impiegabili.

Per quanto riguarda infine il rate iniziale del mutation, questo sarà di 0.02, che è un valore indicativo generalmente utilizzato come valore iniziale negli algoritmi genetici. Questo consentirà una leggera mutazione iniziale che poi si andrà ad adattare nel corso del tempo.

3.9 Stopping condition

L'ultimo punto di questo capitolo del documento riguarda i criteri di stop. Nel caso in essere si sono voluti utilizzare in maniera combinata due criteri d'interruzione della generazione:

1. **Valore target**: per evitare dispendi inutili di tempo e risorse, si è deciso di determinare un valore target per la fitness, che una volta raggiunto determinerà l'interruzione dell'evoluzione e quindi la terminazione del processo.

2. **Numero massimo di generazioni**: il numero delle generazioni dopo il quale si deve interrompere l'evoluzione. Questa soluzione è stata definita come complementare alla precedente, garantendo comunque un'interruzione dell'evoluzione anche qual'ora non si dovesse individuare una soluzione con valore di fitness pari a quello target.

Entrambi i valori saranno affinabili al fine di ottenere le migliori performance possibili in fase di implementazione e valutazione dell'algoritmo.

Implementazione

4.1 Strumenti e Librerie

Per strumenti si vuole intendere linguaggi utilizzati, IDE ed eventuali elementi di supporto per la definizione del codice. Si sono usati i seguenti:

1. **Python** : si è scelto di usare Python per il grande numero di librerie estremamente utili per lo sviluppo di algortimi finalizzati all'intelligenza artificiale

2. **GitHub**: è stato utilizzato per garantire collaborazione e tracciabilità tra le modifiche apportate da uno dei due componenti del team.

3. **Visual Studio Code**: L'IDE che meglio si prestava alle necessità di sviluppo. Inoltre Visual Studio Code è l'IDE meglio conosciuto e più utilizzato dai componenti del team, questo ha consentito fluidità nell'impiego dei mezzi messi a disposizione dallo stesso. L'ultimo aspetto che ha portato alla scelta di impiegare questo IDE è l'integrazione con i sistemi di pull e push dei commit di GitHub, rendendo facile la sincornizzazione oltre che permettendo di evitare così errori di sincronizzazione delle versioni.

4. **Copilot** : sempre a seguito dell'impiego di Visual Studio Code si è voluto utilizzare come supporto alla scrittura del codice Copilot, che è stato ritenuto come meglio performante rispetto al ChatGpt per la possibilità di iniettare il contesto di utilizzo oltre che di richiedere l'intervento diretto sul codice.

Le motivazioni che stanno dietro la scelta di ognuno degli elementi sopra definiti ha ragioni tra il personale e il pratico. Verranno approfondite solo le seconde.

L'impiego di librerie Python è una scelta che difficilmente è omettibile in un contesto come quello degli algoritmi genetici, in quanto permette di semplificare di molto il lavoro concentrandosi sulle performances e sull'analisi delle singole variabili che si impiegano nello sviluppo. Le librerie in questione sono numerose ma si vuole sottolineare l'impiego di tre principali, e cioè **NumPy**, **random** e **math**.
NumPy è estremamente utile in questo contesto per la possibilità che dà di generare ed interagire facilmente con le strutture dati più disparate, oltre a permettere di eseguire calcoli di diversa difficoltà tramite chiamate a funzioni semplici. Nel contesto

trattato l'ausilio fornito è stato chiaramente più relativo all'ambito matematico, in quanto la semplicità della rappresentazione dell'individuo come array avrebbe consentito la sua rappresentazione anche senza l'impiego della suddetta libreria. Come detto però i calcoli necessari alla trattazione del problema hanno richiesto un supporto sostanziale. Lo stesso discorso legato ai calcoli riguarda la libreria math. Infine la libreria random è stata utile in molti casi, quali la generazione e la scelta della popolazione, la scelta degli individui per il K-Way Tournament e deipunti di Taglio per il K-Point Crossover e per determinare se un individuo dovesse o meno subire una mutazione.

4.2 Introduzione al codice

Una parte importante del codice, sia per la sua corretta funzionalità che per la possibilità di effettuare una facile manutenzione e testing delle performances, sono state le costanti. Di seguito si è ritenuto necessario inserire la definizione di alcune di esse per rendere chiaro l'impatto che le singole modifiche avrebbero sull'interezza del progetto e sulla resa dell'algoritmo là dove non fosse già stato precedentemente chiarito:

- ***POP SIZE***: Indica il numero della popolazione che si ha all'inizio e che ci si aspetta di ottenere ad ogni nuova generazione;
- ***NUM CELLS***: Indica la dimensione degli individui. Riportato in termini del contesto reale di applicazione, ognuna delle celle degli array andrà a rappresentare una componente reale del sistema che si sta sviluppando;
- ***TOTAL RESOURCES***: Indica il valore totale delle risorse. Questo valore è soggettivo delle singole realtà d'interesse e si considera la possibilità che ogni utilizzatore indichi in questa variabile il valore massimo di risorse che ritiene di impiegare al fine di realizzare il proprio sistema;
- ***MIN RESOURCES***: Indica il valore di risorse minimo relativo al criterio definito in fase di analisi del problema nella documentazione, per i vincoli definiti in precedenza tale valore, nel contesto in essere, è pari a 20.
- ***RANDOM RESOURCES***: Indica un parametro ottenuto dal rapporto tra le due costanti viste precedentemente, TOTAL RESOURCES E NUM CELLS, moltiplicate per il valore fisso di 1.25.
- ***TOURNAMENT SIZE***: Indica un valore di K dell'algoritmo di selezione, tale valore non può essere inserito manualmente, ma è ricavato dalla divisione

della dimensione della popolazione per 10.

- ***NUM WINNERS***: Indica i numeri di vincitori che si intende prelevare in totale dall'algoritmo di selezione del tournament size, anche questo valore, così come il precedente, è determinato in maniera che si adatti senza l'intervento esterno, a diversi ambiti d'impiego;
- ***CROSSOVER POINTS***: Indica il numero dei punti di crossover per l'algoritmo K-Points Crossover.
- ***MAX GENERATIONS***: Indica il valore massimo di generazioni per la stopping condition.
- ***TARGET FITNESS***: Indica il secondo valore relativo alle stopping condition.

```
# Parametri iniziali
POP_SIZE = 32 # Dimensione della popolazione
NUM_CELLS = 25  # Numero di celle per individuo
TOTAL_RESOURCES = 2000 # Risorse totali disponibili
MIN_RESOURCES = 20 # Risorse minime per cella (esclusa la cella 0)
RANDOM_RESOURCES = int(TOTAL_RESOURCES/NUM_CELLS) * 1.25 # Risorse casuali per la distribuzione [NON MODIFICARE]
TOURNAMENT_SIZE = int(POP_SIZE/10) # Dimensione del torneo [NON MODIFICARE]
NUM_WINNERS = int(POP_SIZE) # Numero di vincitori [NON MODIFICARE]
CROSSOVER_POINTS = 4 # Numero di punti di crossover
MAX_GENERATIONS = 5 # Numero massimo di generazioni
TARGET_FITNESS = 7 # Soglia di fitness target
LAMBDA_VALUE1 = 80  # Valore di lambda per danni [NON MODIFICARE]
LAMBDA_VALUE2 = 0.02 # Valore di lambda per costo [NON MODIFICARE]
LAMBDA_VALUE3 = 6.7 # Valore di lambda per distribuzione risorse [NON MODIFICARE]
OMEGA1 = 0.33  # Peso per l'obiettivo danni
OMEGA2 = 0.33  # Peso per l'obiettivo costi
OMEGA3 = 0.33  # Peso per l'obiettivo distribuzione pesata
```

Figure 1: Costanti

Tali valori sono stati definiti come costanti per permettere una migliore manutenzione del codice e, in fase di analisi delle prestazioni, questa scelta implementativa ha permesso di modellare il codice a piacimento con il minimo sforzo.

4.3 Algoritmi

L'implementazione degli algoritmi di crossover e mutation ha richiesto la definizione di costanti importanti come mutation rate e crossover rate, dove questi due valori indicano la probabilità con cui crossover e mutation si andranno a verificare in relazione alla popolazione.
Per il mutation rate si è posta l'attenzione solo ulteriormente sul valore di fitness threshold, che è quello che determinerà la variazione del mutation rate nell'algoritmo

adaptive. Nell'algoritmo di crossover risiedono i controlli per assicurare il rispetto dei vincoli.
Non ci si dilungherà a spiegare l'algoritmo di selezione, in quanto la pratica non differisce dalla teoria precedentemente illustrata, l'unica cosa degna di nota che stata inserita in fase di implementazione è il controllo sul numero della popolazione, se gli individui che partecipano all'algoritmo sono in numero dispari allora si riporta il valore ad un numero pari.
Un algoritmo che richiede maggiore attenzione è però quello che definisce la popolazione iniziale e che si occupa di generare una prima popolazione controllata, assicurandosi che questa rispetti i vincoli definiti.

Popolazione.py > ...

```python
def generate_population(pop_size, num_cells, total_resources, min_resources, RANDOM_RESOURCES):
    population = []

    base_resource = 20  # Risorsa fissa da assegnare inizialmente a tutte le celle
    remaining_resources = total_resources - base_resource * (num_cells - 1)  # Risorse rimanenti per la distribuzione casuale

    if remaining_resources < 0:
        raise ValueError("Non ci sono abbastanza risorse per rispettare il minimo per ogni cella!")

    for _ in range(pop_size):
        individual = np.zeros(num_cells, dtype=int)
        individual[0] = total_resources  # La cella 0 contiene il totale delle risorse
        risorse_totali = individual[0]

        # Distribuire 20 risorse iniziali a tutte le celle tranne la cella 0
        for i in range(1, num_cells):
            individual[i] = base_resource  # Assegniamo 20 a tutte le celle (eccetto la cella 0)

        # Ricalcoliamo le risorse da distribuire
        remaining_resources = risorse_totali - base_resource * (num_cells-1)  # Risorse rimanenti dopo l'assegnazione iniziale

        # Distribuire le risorse rimanenti in modo casuale
        for i in range(1, num_cells):
            if remaining_resources > 0:
                max_possible = remaining_resources
                resources = random.randint(0, int(RANDOM_RESOURCES))
                if resources > max_possible:
                    resources = max_possible
                individual[i] += resources  # Assegniamo le risorse alla cella
                remaining_resources -= resources  # Decrementiamo le risorse rimanenti


        population.append(individual)

    return population
```

Figure 2: Population.

4.4 Codice della Fitness

La funzione di fitness combinata e i relativi obiettivi non hanno subito cambiamenti sostanziali nell'implementazione. Di seguito viene riportato il codice e le costanti relative all'obiettivo numero 3 (Fig.3) e alla funzione combinata (Fig.4) per rendere chiara l'applicazione dei valori lambda e le strategie adottate al fine di rendere la teoria realizzabile con metodologie semplici. Nella parte di performance si appro-

fondirà in che modo si è provato il valido funzionamento di questi aspetti e valori critici.

```python
def obiettivo3_distribuzione(individual, lambda_value):
    # Risorse totali disponibili (prima cella dell'individuo)
    total_resources = individual[0]

    # Risorse allocate per ogni cella, eccetto la prima
    resources_allocated = individual[1:]

    # Ranking: indice + 1 (la cella 1 ha ranking più alto, la cella n ha ranking più basso)
    rankings = list(range(1, len(resources_allocated) + 1))

    # Risorse totali allocate
    total_allocated = sum(resources_allocated)

    # Somma Vulnerabilità
    total_vulnerability = sum((len(rankings) / (rankings[j] * (resources_allocated[j] ** 0.5)))
    for j in range(len(rankings))
    if rankings[j] > 0 and resources_allocated[j] > 0
    )

    # Obiettivo 3: Distribuzione pesata
    distribution_weight = 0
    for i in range(len(rankings)):
        if rankings[i] > 0 and resources_allocated[i] > 0:
            weight = resources_allocated[i] / total_allocated if total_allocated > 0 else 0
            distribution_weight += (weight * (resources_allocated[i] / rankings[i])) / total_vulnerability


    # Funzione di fitness: Distribuzione pesata
    fitness = lambda_value * distribution_weight

    return fitness
```

Figure 3: Obiettivo 3.

```python
def fitness_combinata(individual, lambda_value1, lambda_value2, lambda_value3, omega1, omega2, omega3):
    # Calcolo delle fitness per gli obiettivi
    fitness1 = obiettivo1_danni(individual, lambda_value1)
    fitness2 = obiettivo2_costo(individual,lambda_value2)
    fitness3 = obiettivo3_distribuzione(individual, lambda_value3)

    # Calcolo della fitness combinata
    fitness = (omega2 * fitness2 + omega3 * fitness3) - omega1 * fitness1

    return fitness
```

Figure 4: Funzione di fitness.

Chiaramente sono presenti anche le costanti di Lambda e Omega, rispettivamente per normalizzare i valori degli obiettivi rispetto agli altri e per assegnare un peso specifico.

Analisi delle performances

L'analisi delle performance è stata tra tutte la parte più interessante e lunga del progetto.

5.1 Valutazione delle costanti

Il punto che ha richiesto più di tutti attenzione e definizione attenta è stata la parte relativa al tuning delle variabili costanti(Fig.1). Le variabili di cui maggiormente ci si è occupati e la cui modifica ha portato a significativi miglioramenti sono state **POP SIZE, TOURNAMENT SIZE, CROSSOVER POINT** e **RANDOM RESOURCES.**

Per quanto riguarda POP SIZE, inizialmente è stato impostato a 10, ma la popolazione generata non era sufficientemente diversificata. Aumentando il valore a 25, si è notato un miglioramento nella varietà degli individui, mentre con un ulteriore incremento a 50 si è raggiunto un equilibrio ottimale tra diversità e qualità della popolazione iniziale. Andando oltre questo valore, l'incremento della popolazione non ha portato a miglioramenti significativi nella qualità della soluzione, nonostante un maggiore consumo di memoria. Questo suggerisce che, oltre una certa soglia, l'aumento della dimensione della popolazione non comporta benefici rilevanti in termini di fitness, ma introduce costi computazionali più elevati.

Il valore di TOURNAMENT SIZE, inizialmente pari a 10, non scalava correttamente con la modifica di POP SIZE. Per risolvere questo problema, si è deciso di rendere questo valore proporzionato al numero di individui presenti, in modo da ottenere un certo equilibrio nel risultato atteso. Inoltre, si è osservato che valori troppo grandi di TOURNAMENT SIZE aumentavano il rischio di ritrovarsi sempre con gli stessi individui tra i vincitori, riducendo così la diversità nella selezione. Al contrario, un valore troppo basso diminuiva le possibilità di integrare individui con fitness ottimali nei tornei, limitando l'efficacia del processo di selezione.

Il valore di CROSSOVER POINT non è cambiato troppo rispetto a quello che si era previsto in fase di implementazione, che si pensava sarebbe stato 3. Un valore più alto di Crossover points ha dimostrato abbassare il valore ottenuto dal calcolo della funzione di fitness senza alterare nessun altro valore disponibile.

Per comprendere il ruolo di RANDOM RESOURCES, è necessario fare un passo in-

dietro e analizzare il processo di generazione della popolazione. A causa dei vincoli specifici del problema, l'algoritmo per la creazione della popolazione è stato sviluppato da zero. Inizialmente, il processo di assegnazione delle risorse prevedeva che ogni cella ricevesse un numero di risorse casuale compreso tra 0 e il massimo disponibile. Questo approccio, però, portava rapidamente all'esaurimento delle risorse, generando quasi sempre individui molto performanti a causa della distribuzione delle risorse. Tuttavia, ciò rappresentava un problema, in quanto riduceva drasticamente la diversità all'interno della popolazione iniziale.
Per ovviare a questo problema, si è modificato il processo di assegnazione introducendo RANDOM RESOURCES, un valore calcolato come una proporzione tra le celle presenti e le risorse totali. In questo modo, ogni cella riceveva un numero di risorse compreso tra 0 e RANDOM RESOURCES, garantendo un'allocazione più bilanciata. Questo approccio ha permesso di generare popolazioni più diversificate, evitando che le risorse si concentrassero nelle prime celle e mantenendo comunque la possibilità che alcuni individui fossero migliori di altri.
Meritano una piccola menzione anche i tre valori LAMBDA VALUE, introdotti per bilanciare i relativi obiettivi. Durante i test, si è osservato che i valori generati da questi obiettivi differivano significativamente tra loro. Per risolvere questo problema, dopo numerosi tentativi e confrontando diverse soluzioni, si è deciso di normalizzare manualmente tutti i valori. Questa normalizzazione ha permesso di rendere i valori comparabili tra loro, evitando che il risultato di un obiettivo potesse sbilanciare eccessivamente il risultato complessivo, introducendo incongruenze nella valutazione degli individui.

5.2 Test pratici

Passando ai test pratici, il programma è stato testato con varie configurazioni per capire come il programma rispondeva in base all'hardware. Nel dettaglio sono state provate 4 configurazioni:

1. Intel i5 11400 con 16 GB DDR4 di RAM
2. Intel N100 con 16GB DDR5 di RAM
3. Ryzen 5 2600 con 16 GB DDR4 di RAM
4. Ryzen 5 5500U con 8 GB DDR4 di RAM

A seguito dei test, si può affermare che, quanto più potente è la CPU, tanto più velocemente l'algoritmo procede attraverso le generazioni. Per quanto riguarda

la RAM, invece, il suo impatto è trascurabile, poiché l'incremento del consumo è molto lento rispetto all'aumento delle generazioni. Ad esempio, dopo circa 5000 generazioni, la RAM utilizzata era di circa 800-900 MB.
Con una configurazione iniziale di 25 celle, 2000 risorse disponibili e 50 individui generati, il punteggio di fitness massimo ottenuto è stato di circa 11. Tuttavia, questo punteggio è raggiungibile quasi esclusivamente con un numero elevato di generazioni e rappresenta uno dei valori massimi teoricamente ottenibili con questa configurazione di input.

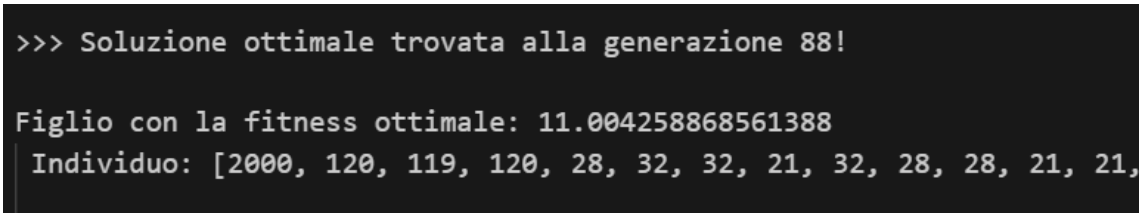

Figure 5: Fitness 11 raggiunta dopo 88 generazioni.

Se invece consideriamo un miglior compromesso tra numero di generazioni e rapporto costi/performance, si è osservato che il punteggio fitness migliore ottenuto sia di circa 7. Questo punteggio viene tipicamente raggiunto entro 5-7 generazioni, offrendo un buon bilanciamento tra efficienza e qualità della soluzione.

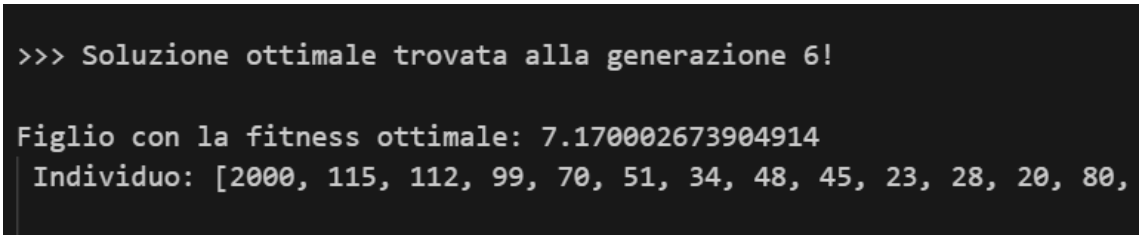

Figure 6: Fitness 7 raggiunta dopo 6 generazioni.

Da notare che questo punteggio scala in base al valore selezionato di NUM CELLS e TOTAL RESOURCES. Infatti andando ad aumentare del 100% questi valori si è notato che uno dei punteggi massimi teoricamente raggiungibili passa da circa 11 a circa 15.

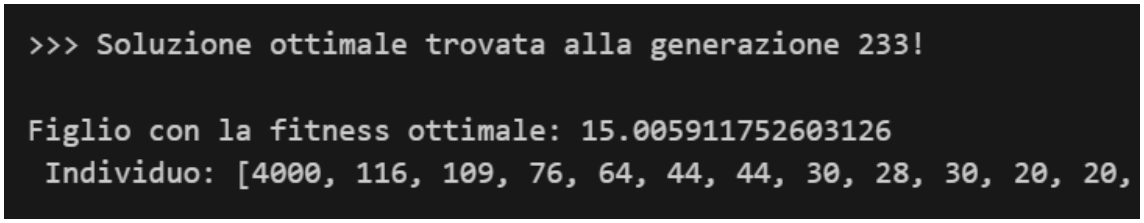

Figure 7: Fitness 15 raggiunta dopo 233 generazioni.

Ovviamente, per raggiungere questo punteggio, non ci si è posti il problema delle performance o dei costi computazionali. Se invece si volesse confrontare il risultato ottenuto con la configurazione precedente, ovvero con circa 100 generazioni, il miglior punteggio raggiunto, dopo numerosi tentativi, è stato di 14.84.

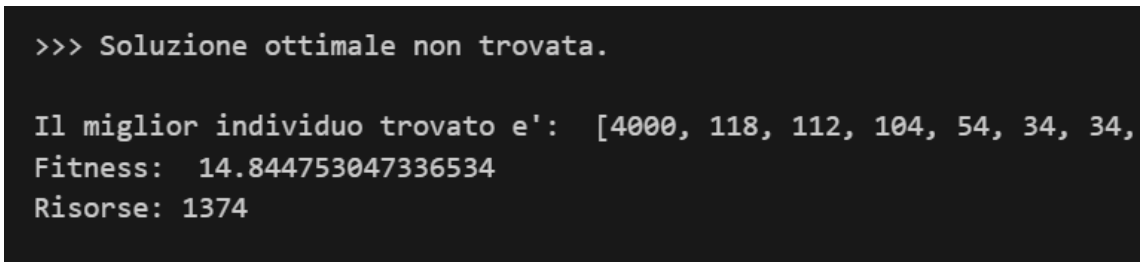

Figure 8: Fitness raggiunta dopo 100 generazioni.

Conclusioni

In questo lavoro è stato sviluppato un approccio basato su algoritmi genetici per ottimizzare la distribuzione delle risorse in un sistema software, con particolare attenzione alla sicurezza informatica. L'obiettivo principale era individuare un equilibrio tra la minimizzazione del danno potenziale e il contenimento dei costi, garantendo al contempo un'allocazione efficiente delle risorse disponibili.
L'implementazione dell'algoritmo ha previsto la definizione di una rappresentazione strutturata del problema, con vincoli ben definiti e una funzione di fitness multi-obiettivo. Attraverso la selezione K-Way Tournament, il crossover K-Point e un meccanismo di mutazione adattiva, il sistema ha dimostrato la capacità di evolversi verso soluzioni ottimali, adattandosi dinamicamente alle necessità del problema.
L'analisi delle performance ha evidenziato l'importanza di un tuning accurato delle costanti e dei parametri dell'algoritmo. Il valore della fitness ha mostrato un miglioramento significativo con l'aumento delle generazioni, raggiungendo un massimo teorico a costo di elevate risorse computazionali. Tuttavia, è stato possibile individuare un compromesso ottimale tra numero di generazioni e qualità della soluzione, ottenendo risultati soddisfacenti con un numero contenuto di iterazioni.
Nonostante i risultati promettenti, alcune limitazioni restano da affrontare. L'algoritmo, pur adattandosi bene ai vincoli e alle priorità definite, potrebbe beneficiare di ulteriori ottimizzazioni per gestire scenari più complessi e dinamici. Un'area di miglioramento riguarda la capacità del modello di adattarsi in tempo reale a condizioni variabili, riducendo i tempi di convergenza senza compromettere la qualità delle soluzioni trovate. Inoltre, l'integrazione di tecniche ibride, come l'apprendimento automatico supervisionato o euristiche avanzate, potrebbe contribuire a una maggiore efficienza e scalabilità.
In futuro, sarà interessante esplorare l'applicazione di questi metodi in contesti reali con requisiti in continua evoluzione. Ad esempio, l'adattamento dell'algoritmo a sistemi distribuiti o a infrastrutture cloud potrebbe aprire nuove prospettive per la gestione dinamica delle risorse in ambienti altamente volatili. Un'altra direzione di ricerca potrebbe riguardare l'ottimizzazione delle funzioni di fitness attraverso modelli predittivi, combinando tecniche di intelligenza artificiale con l'ottimizzazione evolutiva per migliorare ulteriormente le prestazioni.

In conclusione, questo studio rappresenta un passo significativo verso l'impiego degli algoritmi genetici per la gestione delle risorse nei sistemi software, offrendo un metodo flessibile ed efficace per supportare decisioni strategiche in contesti di elevata complessità. L'estensione del modello a scenari sempre più diversificati e la sua integrazione con altre metodologie di ottimizzazione rappresentano direzioni promettenti per future ricerche.

References

Crossover Operators in Genetic Algorithms: A Review
`https://tinyurl.com/crossover-point`
Adapting Crossover and Mutation Rates in Genetic Algorithms
`https://tinyurl.com/cross-mutation`
A Review of Tournament Selection in Genetic Programming
`https://tinyurl.com/k-way-tournament`
Introduzione a Python
`https://tinyurl.com/python-ga`
Slide del corso di Fondamenti di Intelligenza Artificiale