UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Ingegneria del Software II

Un Algoritmo Genetico per la Generazione di Casi di Test per le Applicazioni Web basate su JavaScript

Anno Accademico 2013/14

relatore

Ch.mo Prof Porfirio Tramontana

candidato
Michele Palumbo
matr. M63/215

Indice

Capitolo 1 - Problematiche del Testing

Le tipologie di attacco verso le moderne applicazioni web fanno parte ormai di un insieme sempre più vasto e sfaccettato. I correnti strumenti di sicurezza automatizzati sono stati trovati carenti in molti settori, guardando l'intero ambito di valutazione di una applicazione web.

Il testing rappresenta un passo imprescindibile dello sviluppo software, tuttavia è una attività che diventa tanto complessa quanto è più articolata l'applicazione da validare. Inoltre, con la crescente popolarità di metodologie di sviluppo agile, un numero sempre maggiore di applicazioni web viene rilasciato in tempi brevissimi. Purtroppo però, ad un incremento quantitativo non è sempre corrisposto un miglioramento qualitativo. Al contrario infatti, si è avuto un forte abbassamento della qualità del software dovuto alla drastica riduzione dei tempi dedicati a ciascuna delle attività di sviluppo, ed in particolare, la fase di verifica e valutazione ne ha risentito maggiormente. Quest'ultima, infatti è spesso considerata troppo complessa, difficile da automatizzare e molto costosa, in termini di tempo. Tutto ciò comporta il rilascio di un software ricco di malfunzionamenti che, inevitabilmente vanno ad aggravare la fase di manutenzione la quale, a sua volta, a lungo andare può portare ad un ulteriore deterioramento della qualità del software causandone la diminuzione del ciclo di vita.

Per poter ottenere una maggiore qualità, bisognerebbe prestare più attenzione all'attività di valutazione finale attraverso lo sviluppo di strumenti efficaci ed efficienti ma soprattutto automatizzati, a supporto di questa fase.

Questo capitolo analizzerà le problematiche dell'automazione del testing limitandosi all'ambito delle applicazioni web moderne; il secondo, invece, illustrerà le tecniche di testing automatico e i relativi strumenti utilizzati; il terzo fornirà una proposta di soluzione al problema discusso, introducendo gli algoritmi genetici; il quarto mostrerà l'architettura dello strumento realizzato con i dovuti riferimenti alle principali classi e metodi implementati; infine, il quinto capitolo presenterà i risultati sperimentali ottenuti.

1.1 Testing Automation

Una prima macro-distinzione tra le varie tipologie di testing può essere fatta in *testing automatizzato* e *testing manuale*. Secondo E. Dustin [38] le ragioni che portano a preferire il primo tra i due sono da ricercare innanzitutto nei costi, in termini di tempo e successivamente nell'efficienza. Infatti nel caso automatizzato utilizzando una tecnica di *regression testing*, informazioni di feedback possono agire sulla logica di controllo in modo che il processo automatico evolva sempre vero la direzione migliore, al fine di ottenere il risultato più efficiente possibile, inerentemente alle metriche che si è scelto di adottare.

D'altra parte il testing manuale rappresenta una soluzione accettabile quando, per esempio, il testing automatizzato non è sufficiente, o costa troppo o non è applicabile perché non in grado simulare in modo adeguato casi d'uso reali del software sotto test.

La tecnica automatizzata, dove applicabile, può ridurre notevolmente i costi della fase di testing che, così come affermato in *The art of software testing* [39], dovrebbe occupare più del 50% dell'intero sforzo del ciclo di sviluppo del software, rappresentandone quindi l'attività più dispendiosa.

Per le motivazioni appena descritte, nel corso di questo studio si è affrontato il problema di ricercare una valida ed innovativa metodologia di testing automatizzato per le applicazioni web moderne, di cui, nel paragrafo seguente, ne saranno illustrate le caratteristiche principali.

1.2 Applicazioni web

Un'applicazione web è un qualsiasi software che viene eseguito in un browser web (client) e che permette l'accesso a funzionalità e risorse remote (server). La loro enorme popolarità sta nella praticità di utilizzo, nell'ubiquità dei browsers, la possibilità di aggiornare e manutenerle senza dover distribuire e installare il software su tutti i computer client.

La loro nascita avviene in contemporanea con lo sviluppo del web dinamico (termine usato per la prima volta nel Gennaio del 1999 da Darcy DiNucci []). Prima di allora infatti, il web era popolato esclusivamente di pagine web statiche che, tramite utilizzo di solo HTML, venivano consegnate all'utente esattamente come memorizzate sui server ed alle quali era associato univocamente un indirizzo URL.

Con la graduale introduzione di linguaggi di scripting, frameworks, e nuove tecnologie che si sono aggiunte alla classica navigazione HTML, la pagina web ha assunto un aspetto sempre più "interattivo". L'effetto della dinamicità è reso possibile in due modi differenti:

- Utilizzando *script client-side* che in risposta alle azioni del mouse o della tastiera o in occasione di eventi di temporizzazione specificati, modificano parti del Document Object Model (DOM) che racchiude tutti gli oggetti dell'intera pagina HTML caricata;
- Utilizzando *script server-side* per cambiare il sorgente della pagina, regolando ad esempio la sequenza o il reload delle pagine web o contenuti web forniti al browser. Le risposte asincrone del server possono essere scatenate da eventi come: inserimento di dati in un form HTML, parametri nell'URL, il tipo di browser in uso, il passare del tempo, o uno stato del database o del server.

Client-side o server-side script, o una combinazione di questi offrono l'esperienza del web dinamico in un browser.

1.2.1 JAVASCRIPT

JavaScript è linguaggio di scripting originariamente sviluppato da Brendan Eich della Netscape Communications con il nome di "Mocha" e successivamente di "LiveScript", ma in seguito è stato rinominato "JavaScript" ed è stato formalizzato con una sintassi più vicina a quella del linguaggio Java di Sun Microsystems (che nel 2010 è stata acquistata da Oracle) fino a diventare uno standard ISO.

JS rappresenta il terreno comune a tutte le applicazione AJAX, operando "di nascosto" tra il browser ed il Web Server. Questo motore tipicamente si occupa della comunicazione con il server e le modifiche sull'interfaccia utente. E' un linguaggio interpretato, il cui codice non viene compilato, ma solo "decifrato" (si noti che in JavaScript lato client, l'interprete è incluso nel browser). La sintassi è relativamente simile a quella del C, del C++ e ovviamente JAVA, dei quali JS ne definisce le funzionalità tipiche ad alto livello (strutture di controllo, cicli, ecc.) e consente l'utilizzo del paradigma object oriented anche se il suo vero punto di forza sta nel fatto di essere un linguaggio debolmente tipizzato e debolmente orientato agli oggetti.

Come si è accennato, in JavaScript lato client, il codice viene eseguito direttamente sul client e non sul server. Il vantaggio di questo approccio è che, anche con la presenza di script particolarmente complessi, il web server non viene sovraccaricato a causa delle richieste dei clients. Di contro, nel caso di script che presentino un codice sorgente particolarmente grande, il tempo per lo scaricamento può diventare abbastanza lungo. Un altro svantaggio è il seguente: ogni informazione che presuppone un accesso a dati memorizzati in un database remoto deve essere rimandata ad un linguaggio che effettui esplicitamente la transazione, per poi restituire i risultati ad una o più variabili JavaScript; operazioni del genere richiedono il caricamento della pagina stessa. Con l'avvento di AJAX, quindi della comunicazione asincrona, tutti questi limiti sono stati però superati.

1.2.2 AJAX e Web 2.0

Il termine AJAX - Asynchronous JavaScript and XML - coniato nel 2005 e strettamente legato al Web 2.0, è una tecnica di sviluppo software per la realizzazione di applicazioni web interattive.

La suddetta non è una tecnologia individuale, piuttosto è un gruppo di tecnologie che collaborano insieme:

- *HTML (o XHTML) e CSS* per il markup e lo stile;
- *DOM (Document Object Model)* manipolato attraverso JavaScript o JScript per mostrare le informazioni ed interagirvi;
- L'oggetto *XMLHttpRequest* per l'interscambio asincrono dei dati tra il browser dell'utente e il web server. In alcuni framework Ajax e in certe situazioni, può essere usato un oggetto *Iframe* invece del *XMLHttpRequest* per scambiare i dati con il server e, in altre implementazioni, tag <*script*> aggiunti dinamicamente (JSON).

In genere viene usato XML come formato di scambio dei dati, anche se di fatto qualunque formato può essere utilizzato, incluso testo semplice, HTML preformattato, JSON e perfino EBML. Questi file sono solitamente generati dinamicamente da script lato server.

Le applicazioni web che usano Ajax richiedono browser che supportano le tecnologie necessarie.

Oggi giorno gran parte delle nuove applicazioni web sono realizzate usando AJAX e altri frameworks del Web 2.0 per lo sviluppo interattivo. Queste tecnologie hanno il grande vantaggio di poter trasportare dati “in background” dal web server senza che l’utente ne faccia esplicita richiesta (ad esempio senza dover aggiornare la pagina html) e quindi in maniera del tutto asincrona (Figura 1). Questo aiuta a creare siti web con molte funzioni che li rendono più simili alle classiche applicazioni desktop, fornendo all’utente finale una forte percezione di interattività. Ovviamente tutto ciò comporta una maggiore complessità a cui consegue un maggior numero di scenari che se non propriamente testati potrebbero portare a spiacevoli imprevisti sia a livello di funzionalità che di performance generando un effetto negativo sulla qualità dell’esperienza finale dell’utente.

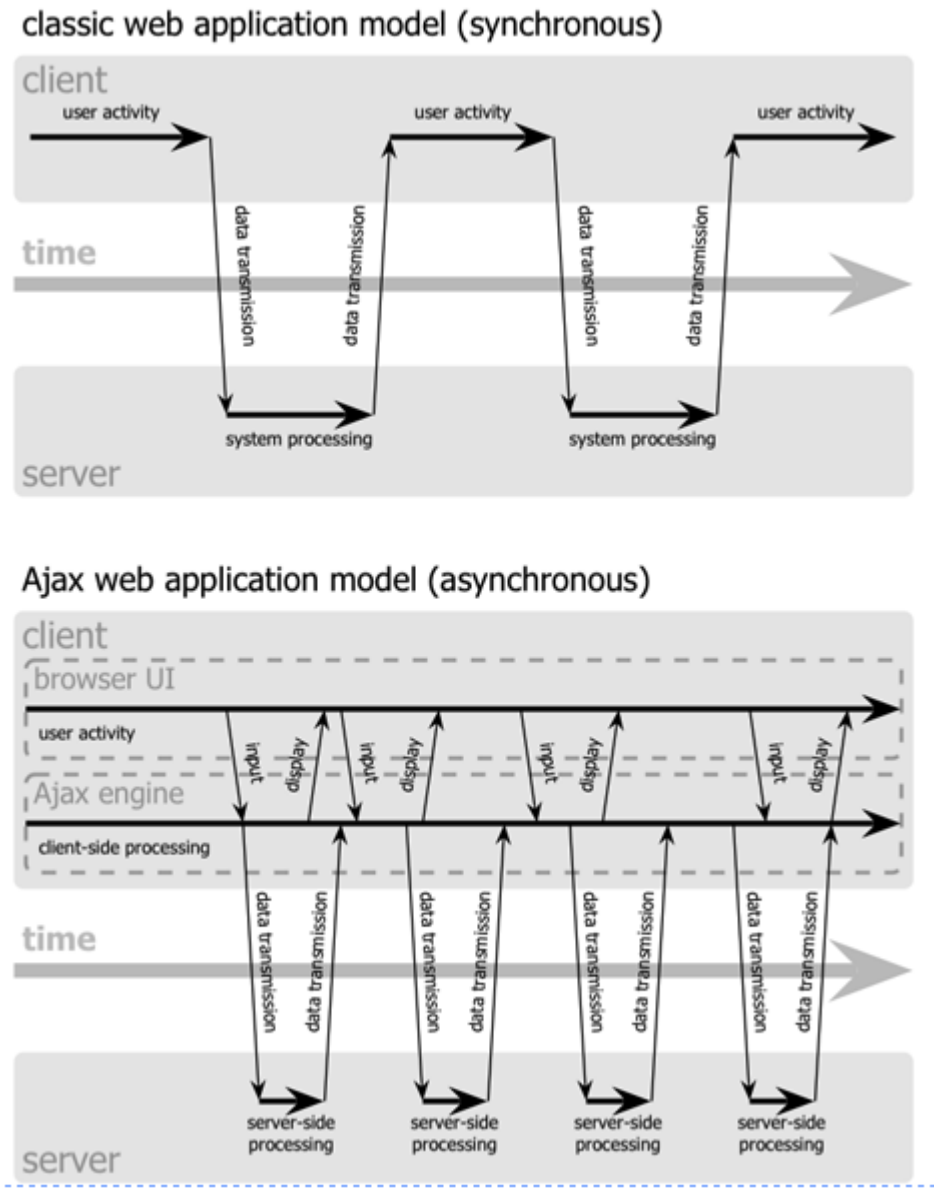


Figura 1

Nella figura sovrastante sono messi a confronto il modello classico, client-server (sincrono) e quello invece introdotto, che introduce un motore (*Ajax engine*) posto lato client che intercetta le richieste e le risposte tra quest'ultimo ed il server, introducendo quindi asincronia nella comunicazione.

1.2.3 Problema del testing per le applicazioni web moderne

La complessità delle nuove tecnologie, e la moltitudine dei frameworks emergenti rende

difficile costruire strategie di testing dotate di soluzioni automatiche che siano valide in modo universale. Inoltre la maggior parte delle tecniche tradizionali si concentrano sulla verifica del livello del protocollo, sulla presenza o meno del supporto a livello framework o sulla capacità di riconoscere accuratamente gli oggetti di scambio tra client e server.

Esplorare e quindi testare efficacemente applicazioni AJAX è quindi fondamentalmente più difficile rispetto alle classiche web applications muti-pagina. In queste ultime infatti l'identificazione dello stato in cui si trova l'applicazione è immediata data la corrispondenza univoca tra URL e pagine web. In AJAX tutto ciò non è vero e non è semplice determinare lo stato dell'interfaccia a causa delle frequenti modifiche che il DOM subisce in seguito alla generazione di eventi seguita dall'esecuzione di codice JavaScript.

Quindi se per le classiche web applications multi pagine, è sufficiente una tecnica di testing tradizionale (estrazione del link, invio della richiesta al server e analisi della risposta) ciò non è vero nel caso di AJAX. Infatti, in questo caso un tipo di analisi statica perderebbe tutto il comportamento runtime e d'altra parte, strumenti di registrazione e ri-esecuzione come Selenium (descritto nei capitoli successivi) richiederebbero un sostanziale sforzo manuale.

Nasce quindi la necessità di introdurre strumenti in grado di leggere il comportamento dinamico in modo da derivarne automaticamente dei modelli (ad esempio grafo degli stati) che in un secondo momento dovranno costituire l'oracolo indispensabile alla fase di testing.

Lo studio in atto si pone come obiettivo il miglioramento della qualità delle testsuites automaticamente generate, in seguito all'esplorazione dell'applicazione, secondo parametri di valutazione dettagliatamente esposti nel paragrafo che segue.

1.3 Valutazione della qualità dei tests

Il testing è un processo di esecuzione del software che ha l'obiettivo di scoprire il maggior numero di malfunzionamenti possibili [41]. Si definisce malfunzionamento l'incapacità

del software di comportarsi secondo le aspettative o le specifiche; la sua natura è dinamica in quante accade in un istante di tempo definito e può essere osservato solo mediante l'esecuzione del software.

Si dice che un test *t* ha successo per un programma *P* se rileva uno o più malfunzionamenti presenti in *P*, mentre un insieme di test *T* è inadeguato se esistono malfunzionamenti in *P* che il test *T* non è in grado di rilevare.

Secondo Dijkstra, il testing non può dimostrare l'assenza di difetti (la correttezza assoluta del programma) ma può solo dimostrare la presenza di essi. Da ciò ne deriva che non esiste e non esisterà mai un test, il cui fallimento (mancata rilevazione di malfunzionamenti) implica la correttezza del programma. Quest'ultimo è di conseguenza, un problema indecidibile.

Nonostante l'indecidibilità della correttezza, l'attività di testing risulta comunque indispensabile allo sviluppatore perché consente a quest'ultimo di acquistare fiducia nel software realizzato, prima di rilasciarlo e renderlo disponibile all'utilizzo.

Gli obiettivi del testing possono essere suddivisi in:

- *Localizzazione dei difetti*: il testing deve aiutare a localizzare i difetti per facilitarne il debugging.
- *Ripetibilità*: il test deve essere ripetibile. Ciò può aiutare lo sviluppatore a testare se il problema è stato risolto. Non sempre però è possibile ottenere la ripetibilità di un test (ad esempio nei casi in cui nel software siano presenti elementi indeterministici).

È possibile inoltre misurare la bontà di un insieme di test tramite due metriche:

- *Efficacia*: è una metrica che valuta in che misura la test-suite sta raggiungendo l'obiettivo del testing. Generalmente è definita come:

$$\frac{Numero\ di\ malfunzionamenti\ trovati}{Numero\ di\ malfunzionamenti\ da\ trovare}$$

Questa definizione si scontra però col problema che non è possibile sapere a priori quanti malfunzionamenti sono presenti all'interno del programma. Quindi spesso si preferisce

misurare la bontà della test-suite secondo valori più oggettivi, come la valutazione della percentuale di linee di codice coperte (LOCs) o numero di nuovi stati scoperti dalla test-suite.

- *Efficienza*: è una metrica che misura quanto la test-suite sta utilizzando bene il tempo. Una test-suite efficiente è dunque una test-suite che raggiunge i risultati preposti col minimo spreco di tempo e di sforzi. Ad esempio può essere calcolata come:

$$\frac{Numero\ di\ test\ che\ trovano\ malfunzionamenti}{Numero\ di\ test\ totali}$$

1.3.1 Problematica: generare test più efficaci e più efficienti

Per la maggior parte dei sistemi, è praticamente impossibile sapere quanto testing è essenziale e quando bisognerebbe fermarlo. Un testing esaustivo, in cui tutte le possibili sequenze di esecuzione del programma sono testate, è impossibile per programmi non banali. Il testing, quindi, deve essere basato su un sottoinsieme di possibili casi di test. Idealmente, quindi, le compagnie di software dovrebbero avere delle politiche sulla scelta di questo sottoinsieme. Queste politiche possono essere basate su politiche generali sul testing, come quella in cui tutte le condizioni devono essere verificate sia per valori veri che per valori falsi. Altrimenti, possono essere basate sull'esperienza d'uso del sistema e focalizzare il testing sulle caratteristiche operazionali del sistema [42]. Ad esempio:

1. Tutte le funzioni del sistema che sono accessibili tramite menu dovrebbero essere testate.
2. Le combinazioni di funzioni che sono accessibili dallo stesso menu dovrebbero essere testate.
3. Dove è previsto un inserimento di un input da parte dell'utente, tutte le funzioni andrebbero testate sia per input corretti che per input non corretti.

Un simile approccio però funziona bene quando le funzioni del programma sono usate in isolamento. Nel caso in cui, invece, queste funzioni sono usate in combinazione con funzioni meno comuni, possono sorgere dei problemi [43].

È quindi chiaro che la problematica della generazione di test più efficaci e più efficienti si deve risolvere in un compromesso dettato dalle esigenze di tempo e/o di qualità.

Un testing che privilegia l'efficacia dovrebbe essere utilizzato quando si vuole ottenere un software affidabile, mentre l'efficienza andrebbe privilegiata quando si vuole ottenere un test meno costoso. La tecnica del testing automatizzato porta a privilegiare l'efficacia in quanto non è richiesto sforzo umano, ma solo costo in termini di tempo.

Possibili condizioni di terminazione di un test potrebbero essere:

- *Criterio temporale*: il testing viene eseguito per un periodo di tempo predefinito.
- *Criterio di costo:* il testing utilizza un sforzo (in termini di mesi/uomo) allocato predefinito.
- *Criterio di copertura:* il testing mira a coprire almeno una volta un determinato scenario. Ad esempio coprire almeno una volta ogni linea di codice.
- *Criterio statistico:* il testing termina quando gli ultimi k test non hanno rilevato malfunzionamenti.

Ovviamente tutte le considerazioni appena fatte valgono in generale, cioè per qualsiasi tipo di software si voglia testare. In realtà però ogni tipo di applicazione richiederebbe delle considerazioni aggiuntive fatte ad-hoc e quindi ciò vale anche per le applicazioni web moderne che, utilizzando la tecnologia AJAX, richiedono un processo di testing che sia leggermente modificato e per il quale è stata proposta una soluzione innovativa che sarà analizzata in maniera dettagliata nel corso di questo studio.

Capitolo 2 – Background e lavori correlati

La maggior parte delle attività di testing del software, ed in particolare quelle relative ai test di regressione, sono troppo laboriose e troppo costose, per essere effettuate manualmente. Inoltre, un approccio manuale non sempre risulta essere efficace nel trovare particolari classi di difetti.

L'automazione offre la possibilità di svolgere efficacemente la fase di verifica e valutazione del software, permettendo di eseguire i test più velocemente e in modo facilmente ripetibile.

2.1 Tecniche di Testing automatico

L'insieme delle tipologie di testing automatico è sempre più vasto e sfaccettato. Oggi giorno sono presenti numerosi approcci differenti, una prima possibile distinzione può essere fatta tra tre macro-categorie elencate di seguito:

- *Code-driven test* : le interfacce, i moduli e le librerie sono testati attraverso una varietà di argomenti di input per convalidare che i risultati restituiti siano corretti.
- *Graphical user interface (GUI)*: un framework genera eventi di interfaccia utente, come sequenze di tasti e click del mouse, e osserva i cambiamenti che provocano su di essa, per verificare se il comportamento osservato del programma corrisponde con quello atteso.
- *API-driven test* : un framework che utilizza un'interfaccia di programmazione per l'applicazione per convalidarne il comportamento sotto test.

Nonostante la ricerca continui a dar vita a numerose metodologie differenti, uno dei modi più validi per generare automaticamente casi di test consiste nel fare riferimento ad un modello del sistema (*test model-based*). In alcuni casi, l'approccio model-based consente agli utenti, anche ai non esperti di linguaggi di programmazione, di creare casi di test in modo automatizzato.

Code – Driven Test.

Nel code-driven test le interfacce pubbliche verso le classi, moduli e librerie sono sollecitate con differenti tipologie di input al fine di verificare se il risultato che viene restituito è corretto.

Una tendenza crescente nello sviluppo del software è l'uso di framework di test, come i frameworks xUnit (ad esempio, JUnit e NUnit) che permettono l'esecuzione di test d'unità per determinare se varie sezioni del codice agiscono come previsto in determinate circostanze. Il code-driven test automatico è una chiave fondamentale per lo sviluppo agile del software, dove è conosciuto come *test-driven development* (TDD). In questo caso infatti i test d'unità sono scritti per definire le funzionalità prima che il codice stesso venga scritto. Tuttavia, questi test evolvono e vengono estesi al progredire della stesura del codice del programma, e quando i malfunzionamenti vengono scoperti, viene effettuata l'operazione di *refactoring* [4]. E' considerato più affidabile, sia perché la copertura del codice è migliore, sia perché è gestito costantemente durante lo sviluppo, piuttosto che una sola volta al termine di un ciclo di sviluppo cascata. Lo sviluppatore scopre immediatamente difetti dopo aver effettuato una modifica, ed è quindi meno costoso da riparare. Infine, il refactoring del codice è più sicuro poiché è molto meno probabile introdurre nuovi difetti quando il codice si presenta in una forma semplice con meno duplicazioni, ma comportamento equivalente.

Graphical User Interface (GUI) Test.

Molti strumenti di automazione di test forniscono funzionalità di registrazione e

riproduzione che consentono agli utenti tester di memorizzare le azioni in modo interattivo e riprodurle nuovamente sull'interfaccia, un numero illimitato di volte, confrontando i risultati effettivi rispetto a quelli previsti. Il vantaggio di questo approccio è che richiede poco o addirittura alcun tipo di sviluppo di software. Questo approccio può essere applicato a qualsiasi applicazione che dispone di un'interfaccia utente grafica. Tuttavia, ricorrere a questa tecnica pone gravi problemi di affidabilità e manutenibilità dei test realizzati. Infatti, operazioni come ad esempio la re-assegnazione di un etichetta ad un pulsante o il suo riposizionamento all'interno della finestra potrebbero richiedere una nuova registrazione del test. Inoltre, la fase di registrazione non sempre è efficiente perché spesso può o aggiungere attività irrilevanti o comunque non memorizzare correttamente alcune sequenze.

Inoltre se una pagina web viene vista come una interfaccia utente allora questo tipo di testing è utilizzabile anche nell'ambito delle applicazioni web. Tuttavia, gli strumenti da utilizzare saranno del tutto diversi perché in questo caso per rilevare i cambiamenti di stato bisogna leggere ed interpretare una pagina HTML che è ben diverso dall'osservare un cambiamento di interfaccia di una applicazione desktop.

Un'altra variante è l'automazione dei test senza script che non utilizza registrazione e riproduzione, ma costruisce un modello della Under Test Application (AUT) e quindi consente al tester di creare casi di test, semplicemente modificando parametri e condizioni all'interno dei test stessi. Questo non richiede competenze di scripting, ma ha tutta la potenza e la flessibilità di un approccio script. Inoltre la manutenzione dei test case sembra essere facile, in quanto non esiste un codice da manutenere e quando AUT cambia gli oggetti software questi possono essere semplicemente re-adattati o aggiunti. Quanto appena detto può essere applicato a qualsiasi applicazione software basata su GUI.

API-driven Test.

Test API – Application Program Interface – è ampiamente utilizzato nell'ambito del software testing ed è una tecnica utilizzata principalmente per cui sistemi che presentano

una collezione di API che necessitano di essere testate. E' per molti aspetti molto simile al test del software a livello di interfaccia utente, solo che invece di sfruttare i principali input e output utente, si utilizza il software per inviare le chiamate alle API, ottenere l'output, e registrare la risposta del sistema. A seconda dell'ambiente di test, è possibile utilizzare una suite di applicazioni test appositamente realizzate, ma molto spesso, si arriva a scrivere codice specifico per verificare il corretto funzionamento dell'API.

.

2.2 Testing Sistematico e Random

In ingegneria le tecniche per automatizzare il testing possono essere ulteriormente suddivise in altre tre categorie principali che si distinguono per il modo in cui viene esplorata l'applicazione:

- Tecniche di *Testing Sistematico*: sono quelle tecniche che esplorano l'applicazione sulla base di un modello.
- Tecniche di *Testing Random*: tecniche che effettuano un'esplorazione casuale dell'applicazione.
- Tecniche di *Ricerca*: sono tecniche che ricercano un compromesso "ottimo" tra gli utilizzi della metodologia sistematica e quella random.

Testing sistematico.

La metodologia sistematica è spesso identificata come la tecnica che *automatizza il design dei test black-box* [32], ma a differenza di questi non è necessario scrivere manualmente i test basati sulla documentazione dei requisiti, piuttosto si crea un *modello* del comportamento del sistema sotto test, il quale cattura alcuni dei requisiti. Sono utilizzati poi strumenti per generare automaticamente i test da questo modello. Il modello deve avere essere piccolo rispetto alle dimensioni del sistema totale, così che non sia troppo costoso produrlo, e dettagliato per descrivere accuratamente le caratteristiche che si vogliono testare.

Una volta ottenuto il modello, come detto, è possibile utilizzare uno strumento per

generare automaticamente i casi di test. L'output del generatore di casi di test sarà un insieme di *casi di test astratti*, ognuno dei quali è una sequenza di operazioni con i valori di input associati e il valore di output che ci si aspetta (l'*oracolo*).

Il prossimo passo è di *trasformare* questi casi di test astratti in casi di test eseguibili. Questa trasformazione è effettuata spesso tramite strumenti automatici e scritti in linguaggi specifici di programmazione, come test JUnit in Java. Questi test possono essere così eseguiti per provare a individuare fallimenti all'interno del sistema sotto test.

I vantaggi dell'adozione di una tecnica di testing sistematica sono molteplici, tra questi:

- riduzione del tempo per la progettazione dei casi di test, che non devono essere scritti manualmente.
- maggiore diversità tra test-suite derivate dallo stesso modello ottenuta semplicemente adottando criteri di selezione diversi.

Testing Random.

Convenzionalmente il testing random è visto come un'alternativa di secondo piano rispetto a quello sistematico. Questa discrepanza tra le due tipologie è dovuta al fatto che il testing random ha bisogno di un *oracolo automatico*, per poter giudicare i risultati raggiunti e di un *profilo operazionale*, ossia una descrizione di utilizzo del sistema da parte di un possibile utente finale [35]. Siccome queste due condizioni sono spesso problematiche nella pratica, il testing random non sempre è possibile.

Ci sono alcuni casi, tuttavia, in cui il testing random è preferibile rispetto al sistematico [33], ad esempio quando i domini di input sono troppo ampi e sparsi o quando è difficile identificare gli stati in cui si trova il sistema. Nel primo caso, infatti, un testing sistematico richiede troppo sforzo, nel secondo invece, può essere vantaggioso utilizzare un testing random quando una ricerca sistematica degli stati che dovrebbero verificarsi e/o si sono verificati è un procedimento troppo lungo e complesso.

2.3 Tecniche basate sulla ricerca

Le tecniche basate sulla ricerca sono una famiglia di tecniche utilizzate per risolvere

complessi problemi di ottimizzazione. Esse sono molto utilizzate in campo ingegneristico in diverse attività, soprattutto nel testing [5, 6, 7, 8] e nella manutenzione automatizzata [9, 10], ma anche in altre attività come nella gestione del ciclo di vita del software [1], progetti di pianificazione e stima dei costi [2, 4] e molte altre ancora.

Un'ampia varietà di differenti tecniche di ottimizzazione e di ricerca sono state proposte. In generale però esse possono essere divise in due macro-categorie:

- Tecniche di ottimizzazione classiche
- Tecniche di ricerca metaeuristiche

Ottimizzazione classiche

Le tecniche di ottimizzazione classiche sono utilizzate nella programmazione lineare e cercano di trovare la soluzione globale ottima.

Gli input per un modello di programmazione lineare sono un insieme di variabili reali e non negative $\{x_1, x_2, \dots, x_n\}$, chiamate variabili decisionali. L'obiettivo è di massimizzare (o minimizzare) il valore di una funzione, chiamata *funzione obiettivo*, espressa sotto forma di espressioni lineari.

Ad esempio una funzione obiettivo è del tipo:

Massimizza $\sum_{i=1}^{n} c_i x_i$

Dove $\{c_1, \dots, c_n\}$ sono un insieme di coefficienti specifici per il problema, soggetti ad un insieme di *m* vincoli della forma:

$$\sum_{i=1}^{n} a_{1i} x_i \leq b_1$$

$$\dots$$

$$\sum_{i=1}^{n} a_{mi} x_i \leq b_m$$

Dove a_{ij} e b_i sono costanti determinate del problema.

Questa formulazione è tipicamente utilizzata nei problemi di allocazione delle risorse. Se un problema ingegneristico può essere formulato in questo modo, allora la programmazione lineare è una buona scelta poiché esistono algoritmi efficienti per la soluzione del problema che è garantita essere di ottimo globale.

Ottimizzazioni metaeuristiche.

In ingegneria ci sono problemi molto complessi dal punto di vista computazionale (NP-hard) che costringono ad orientarsi verso tecniche cosiddette metaeuristiche.

Di seguito saranno discusse le tre principali tecniche di ricerca metaeuristica, con particolare interesse verso gli algoritmi genetici, argomento principale del lavoro svolto di tesi, a cui sarà dedicato un paragrafo.

Hill Climbing: Hill Climbing inizia da una soluzione candidata scelta casualmente. Ad ogni iterazione, sono considerati solo gli elementi di un insieme di "vicini" alla soluzione corrente. I "vicini" tipicamente costituiscono una piccola mutazione della corrente soluzione. Una "mossa" è fatta verso il vicino che incrementa la funzione di fitness. La funzione di fitness è quella che permette di associare ad ogni soluzione uno o più parametri legati al modo in cui quest'ultima risolve il problema considerato.

Esistono due modi di selezionare il vicino verso il quale muoversi per migliorare la fitness:

- *Next ascent hill climbing* in cui viene selezionato il primo vicino trovato
- *Steepest ascent hill climbing* in cui tutti i vicini sono esaminati per trovare il vicino che incrementa maggiormente la fitness

Se non esiste nessun vicino con una fitness migliore, allora la ricerca termina e quindi l'ottimo (forse locale) è stato trovato.

Simulated Annealing: Simulated Annealing (letteralmente Ricottura Simulata) può essere pensata come una variazione della tecnica di Hill Climbing che evita il problema del massimo locale permettendo mosse che possono decrementare la fitness. Si ispira al processo di ricottura in ambito metallurgico, in cui si permette una fase di riduzione della temperatura del metallo per poterne aumentare la forza.

Durante l'esecuzione di un algoritmo che utilizza la tecnica di Simulated Annealing ci si potrà muovere da un punto x_1 verso un punto *"peggiore"* x'_1 con una probabilità che è in

funzione del calo di fitness e di un parametro di "temperatura". Il comportamento finale è simile a quello della tecnica di Hill Climbing, ma il Simulated Annealing consente nelle prime fasi "più calde" un'esplorazione migliore dello spazio di ricerca con la speranza che le temperature più alte consentano di scappare dai massimi locali.

2.4 Algoritmi genetici

Quelli genetici sono degli algoritmi di ottimizzazione e di apprendimento automatico totalmente ispirati ai processi dell'evoluzione biologica della specie. *John Holland* è stato il primo ad introdurre questo concetto nell'ambito del software [30] con una importante pubblicazione all'interno della quale spiegava come fosse possibile astrarre i meccanismi di selezione naturale in campo genetico e farne da ispirazione per la progettazione di sistemi software automatizzati.

L'algoritmo genetico rappresenta un efficace metodo di ricerca intelligente dell'ottimo e rivela la sua grande forza ed efficacia nella risoluzione di problemi di grosse dimensioni, fornendo una nuova metodologia nell'ambito della generazione dei dati di test [1].

Il problema dell'ottimizzazione è affrontato tramite la manipolazione di una popolazione iniziale che rappresenta l'insieme di tutti gli individui, ovvero i cromosomi, sui quali l'algoritmo lavora singolarmente. Infatti, ogni cromosoma viene valutato in base ad una funzione di fitness, che ricopre un ruolo fondamentale perché proprio ad essa è legato il successo della risoluzione di un dato problema. Data la popolazione iniziale, si procede scegliendo coppie di individui, da utilizzare nel ruolo di genitori per l'accoppiamento, e sostituendo poi alcuni membri della popolazione attuale con i nuovi cromosomi ottenuti dall'incrocio. Il processo di ricambio generazionale della popolazione va avanti fino a quando un criterio di arresto non è stato soddisfatto [32].

Così, gli algoritmi genetici sono stati impiegati con successo per automatizzare la generazione di dati di test, riducendo sensibilmente il tempo e lo sforzo del tester. Una serie di operazioni vengono ripetute in modo ciclico con l'obiettivo di ottenere al termine di ogni iterazione, una nuova generazione migliore (in base alle fitness adottate) rispetto alla popolazione di partenza.

Un algoritmo genetico, visto in modo molto generico è presentato di seguito:

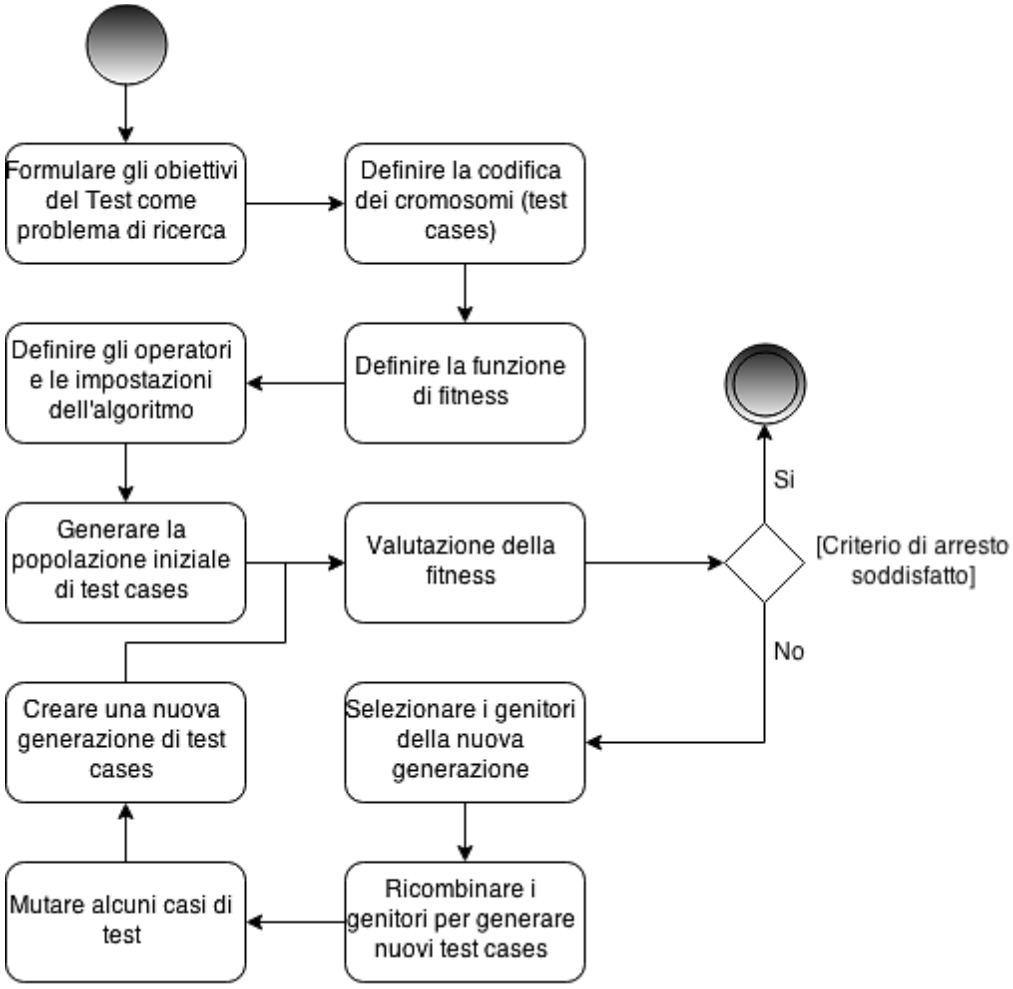


Figura 2

Questo tipo di algoritmo è stato adattato alle esigenze degli sviluppatori per essere sfruttato al meglio nei diversi ambiti di ricerca, e quindi nel corso degli anni sono state introdotte molte variazioni rispetto al processo globale, tuttavia gli elementi principali rimangono:

- La scelta della rappresentazione del problema (codifica)
- La scelta della popolazione iniziale
- Definizione della funzione di fitness
- Il modo in cui avviene la ricombinazione (crossover)
- La tecnica di mutazione

2.4.1 La codifica

Il primo passo per lo sviluppo di un algoritmo genetico è la scelta della rappresentazione del problema tramite una codifica scelta.

La funzione di codifica è definita come:

$$C: S \rightarrow X$$

Dove S è lo spazio delle soluzioni del problema e X è lo spazio dei cromosomi (spazio di ricerca).

Le due tipologie di codifica più diffuse sono:

- *Codifica binaria*: la codifica binaria è particolarmente adatta per problemi di ottimizzazione combinatoria, come nel *problema dello zaino* in cui bisogna trovare la miglior combinazione di oggetti, ad ognuno dei quali è associato un peso e un valore, da poter inserire all'interno di uno zaino che può sopportare un certo limite di peso.
- *Codifica a permutazione*: ideale per problemi a permutazione, come il *problema del commesso viaggiatore* in cui, data una rete di città, connesse tramite delle strade, bisogna trovare il percorso di minore lunghezza che un commesso viaggiatore deve seguire per visitare tutte le città una e una sola volta per poi tornare alla città di partenza.

Ogni tipo di codifica può però essere usata per meglio adattarsi al problema in esame, come la codifica a numeri reali o un'altra codifica ad-hoc.

2.4.2 La funzione di fitness

La funzione di fitness è un indicatore della salute della popolazione. Il modo in cui è definita dipende dal problema in esame. Proseguendo l'analogia tra il processo di evoluzione in natura e i problemi di ottimizzazione, la funzione di fitness rappresenta la funzione obiettivo mentre i geni ed i cromosomi sono identificabili come le variabili decisionali.

L'obiettivo di un algoritmo genetico consiste nell'ottimizzare il valore della funzione di fitness globale manipolando le variabili decisionali al fine di migliorare le fitness locali.

2.4.3 Il crossover

Il crossover è l'operatore principale di un algoritmo genetico e si occupa di diversificare la popolazione. Esso accoppia due individui (genitori) generando due nuove soluzioni (figli) che presentano un patrimonio genetico misto composto delle variabili dedotte da quelle dei genitori.

Ci sono tre passi fondamentali per l'implementazione di un crossover:

- La modalità di scelta dei genitori
- La modalità di selezione degli individui da scartare per far posto ai figli generati tramite il crossover
- La tipologia di crossover da applicare

Per quanto riguarda la modalità di scelta dei genitori, in accordo con Goldberg et al. [15], esistono quattro tipi principali di selezione:

- *Proportionate reproduction*: un individuo avrà una possibilità *p* di essere scelto per la riproduzione proporzionale al valore della sua fitness. Un esempio di proportionate reproduction è la roulette wheel selection [16] [18], mostrata in figura 3.

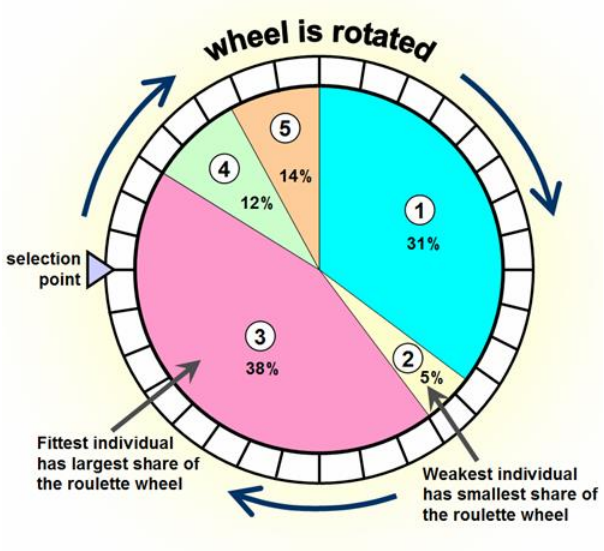


Figura 3

- *Ranking selection* [19]: i cromosomi vengono ordinati dal migliore al peggiore e ad

ogni posizione viene associato il numero di accoppiamenti prestabiliti. Agli individui migliori sarà consentito, quindi, un numero maggiore di accoppiamenti stabilito in maniera statica.

- *Tournament Selection* [20]: nella sua variante più semplice, coppie di individui sono prese a caso tra la popolazione e confrontate. All'individuo con maggiore valore di fitness sarà concesso l'accoppiamento. Questo processo si itera fin quando non si è raggiunto il numero prestabilito di accoppiamenti. È anche possibile utilizzare tornei più grandi di quello binario, dove il migliore tra gli *n* individui scelti entrerà nella fase di accoppiamento.
- *Steady-State reproduction* [17]: la Steady-State reproduction (o anche *Genitor*) è una tecnica di riproduzione che sceglie una coppia di candidati per la riproduzione in base alla loro fitness e la sostituisce al peggiore individuo della popolazione. L'idea alla base di questa tecnica differisce quindi dalle altre, in cui si sceglievano prima tutti gli individui da far accoppiare. La Steady-State reproduction invece funziona in modo individuale ed è effettuato solo un accoppiamento per volta.

Sempre in accordo con Goldberg et. al [15], la Steady-State reproduction è la tecnica che raggiunge aumenti maggiori di fitness. Solo la Tournament Selection, con tornei di ampie dimensioni o ranking non lineari, ha prestazioni comparabili mentre la Proportionate Reproduction è la tecnica che si è comportata in modo peggiore.

Esistono invece quattro modalità di scelta degli individui da scartare per dar posto alle nuove generazioni:

- Tutti i cromosomi di una generazione *n* vengono rimpiazzati. Ciò è ottenuto generando, in fase di crossover, tanti cromosomi figli quanti sono gli abitanti della popolazione della generazione precedente. Ciò comporta che ci sia la possibilità che un cromosoma con alta fitness, e quindi con alta probabilità di riprodursi, riesca a riprodursi troppe volte diventando così un *super-cromosoma* (ovvero un cromosoma con fitness molto alta) o che, in una generazione sfortunata, non riesca a riprodursi affatto facendo perdere così un importante patrimonio genetico.

- I genitori vengono scartati per far posto ai propri figli. Questa modalità dovrebbe evitare che un *super-cromosoma* possa prendere il sopravvento della popolazione scongiurando il rischio che col passare delle generazioni tutti gli abitanti inizino a somigliarsi tra loro. C'è però il rischio che un crossover tra due cromosomi con alta fitness possa generare due figli con bassa fitness, perdendo così patrimonio genetico.
- I peggiori individui della popolazione sono scartati e si lasciano sopravvivere sia i genitori che i propri figli (come nella Steady-State reproduction). In questo modo sono creati un minor numero di individui per iterazione, ma non dovrebbero presentarsi i problemi delle modalità precedenti.
- I *k* individui migliori di una popolazione vengono "salvati" e parteciperanno alla generazione successiva insieme ai cromosomi generati tramite gli operatori di crossover. Necessita di un processo di normalizzazione per evitare che un cromosoma prenda il sopravvento all'interno della popolazione.

Infine ci sono più modalità in cui due cromosomi possono scambiarsi il loro materiale genetico. Il materiale genetico viene scambiato attorno ad uno o più punti, chiamati *crosspoints* o punti di taglio secondo diverse opzioni:

- *Single-point crossover:* il materiale genetico viene scambiato attorno ad un singolo punto di taglio scelto in maniera casuale o predefinita per ottenere due teste e due code. La prima nuova soluzione avrà la testa del primo genitore e la coda del secondo, viceversa la seconda soluzione sarà combinazione della testa del secondo genitore e la coda del primo. Per una codifica binaria avremo la situazione espressa in figura 4.

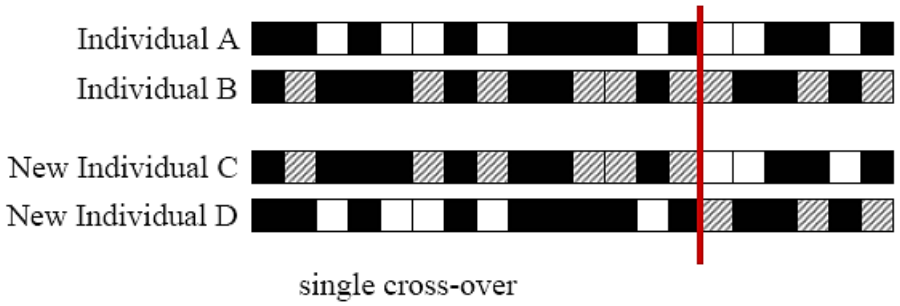


Figura 4

- *Two-point crossover*: il two-point crossover consiste nel creare due punti di taglio. Si avranno così due cromosomi aventi una testa, una parte centrale e una coda. La prima soluzione generata avrà la testa e la coda del primo genitore e la parte centrale del secondo. La seconda nuova soluzione avrà la parte centrale del primo genitore e testa e coda del secondo. Per una codifica binaria avremo la situazione espressa in figura 5.

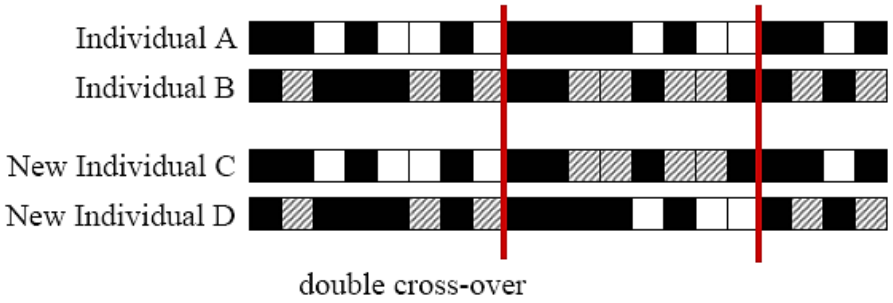


Figura 5

- *Crossover uniforme*: questa tecnica di crossover è differente dalle altre. Il cromosoma figlio sarà generato dal materiale genetico dei genitori in accordo a una "maschera di crossover", generata casualmente, che decide se copiare per ciascun gene dal primo cromosoma genitore o dal secondo. Se il k-esimo bit della maschera vale '0', il cromosoma figlio riceverà quel gene dal primo genitore, altrimenti lo riceverà dal secondo. Per il secondo figlio si avrà il ragionamento contrario.

2.4.4 La mutazione

La mutazione consiste nella modifica pseudocasuale di alcune parti dei geni in base a coefficienti definiti inizialmente. La mutazione può essere utile per riportare nella popolazione materiale genetico che era andato perduto nelle generazioni precedenti o per esplorare nuovi spazi di ricerca.

Tradizionalmente la mutazione è vista come un operatore secondario a cui è associata una probabilità minore rispetto all'operatore di crossover, tuttavia nel caso in cui quest'ultimo sia di difficile applicazione ad esempio per mancanza di individui compatibili, il ruolo della mutazione può assumere un aspetto predominante.

2.5 Testing automatizzato tramite Algoritmi Genetici

Le tecniche genetiche sono state molto utilizzate nel mondo dell'ingegneria e in special modo nel campo del testing automatizzato. Di seguito sono riportati alcuni lavori basati sul testing automatizzato del software tramite algoritmi genetici.

Un primo esempio è proposto in [22] dove B.F. Jones et al. hanno utilizzato gli algoritmi genetici per generare un insieme di test per il testing strutturale. B.F. Jones fa uso di un crossover uniforme, in quanto ha mostrato prestazioni migliori rispetto al single-point e al two-point crossover, e ha ottenuto importanti risultati anche dall'operatore di crossover. In un test mutazionale effettuato per analizzare le prestazioni della test-suite generata, si è riusciti a scovare ed uccidere il 99% di mutanti.

P. R. Srivastava e T. Kim [26] hanno utilizzato invece un algoritmo genetico per scovare i percorsi più a rischio di errori in un software. La funzione di fitness è descritta tramite un CFG (Control Flow Graph) pesato, che assegna un peso maggiore a nodi più a 'rischio'. L'algoritmo fa uso di un single-point crossover e seleziona i cromosomi da riprodurre tramite la tecnica di Proportionate Reproduction.

C. C. Michael et. al [27] cercano di risolvere il problema della generazione dei dati per il testing di un software. L'algoritmo genetico sviluppato fa uso di un single-point crossover e selezione dei candidati tramite ruota della fortuna.

2.6 Strumenti utilizzati

Nel panorama del testing automatico delle applicazioni web moderne, il continuo evolversi delle tecnologie e l'introduzione di nuovi frameworks costringe gli sviluppatori di test a cercare sempre nuove e valide soluzioni adattabili alle più moderne tecnologie. Dalla nascita di AJAX e del Web 2.0 quindi, è nata l'esigenza di introdurre diversi strumenti che messi a servizio dei tester hanno permesso la realizzazione di tool di testing più complessi.

In questo paragrafo e nei successivi saranno introdotti e discussi brevemente alcuni di essi, con particolare interesse verso *Crawljax* [28] che, come si vedrà in seguito, sarà parte integrante del lavoro svolto.

2.6.1 Selenium

Selenium [29] è un framework utilizzato nello sviluppo di testing software per applicazioni web. E' un tool che registra e riesegue i test senza imparare il linguaggio di scripting. Inoltre offre la possibilità di scrivere i test in un vasto numero di linguaggi di programmazione (come Java, C#, Groovy, Perl, Php, Python, Ruby) e l'esecuzione di essi attraverso tutti i web browser moderni. E' uno strumento open source dotato di licenza Apache 2.0.

Analizziamone brevemente i componenti principali:

- *Selenium IDE:* è un ambiente di sviluppo completamente integrato per i tests Selenium; è implementato come componente aggiuntivo di Firefox e permette di registrare, modificare e fare debugging dei tests;
- *Selenium Client Api:* come alternativa al linguaggio Selenese (linguaggi di default di Selenium) , offrono la possibilità di scrivere i tests in diversi linguaggi
- *Selenium Remote Control (RC):* è un server, scritto in Java, che accetta comandi per il browser via HTTP e rende possibile scrivere test automatici in tutti i linguaggi
- *Selenium WebDriver:* è il successore di RC, accetta i comandi (inviati tramite

Selenese o tramite Client Api) e li invia al browser. Il WebDriver è implementato attraverso uno specifico browser che invia comandi e ne riceve i risultati senza aver bisogno di un server speciale che ne esegue i tests

- *Selenium Grid:* è un server che permette ai tests di usare istanze di web browser in esecuzione su macchine remote; Grid funge da hub che viene contattato dai tests per ottenere accesso alle istanze di web browser in remoto, permettendo quindi l'esecuzione di tests in parallelo.

2.6.2 Crawljax

Crawljax [30] è stato il primo strumento disponibile in grado di riconoscere dei contenuti dinamici all'interno delle web application AJAX-based senza dover necessariamente richiedere uno specifico URL per ciascuno web-state.

Crawljax è implementato in Java e comprende un numero elevato di librerie e web tools che sono stati utilizzati per costruirlo ed eseguirlo. L'interfaccia del browser integrato supporta tre differenti tipologie in corrispondenza dei più famosi web browser in circolazione (IE, Chrome e Firefox) ed inoltre essa è stata implementata al di sopra del WebDriver di Selenium (APIs.6).

Il rilascio di Crawljax è avvenuto nel 2008 sotto licenza open source di Apache e da allora è stato usato da un numero sempre più vasto di utenti e anche sviluppatori che lo hanno impiegato in un ampio range di caso di studio differenti, di seguito alcuni riferimenti ai lavori principali nei quali è stato utilizzato:

- *automated model-based and invariant-based testing* [Mesbah and Van Deursen 2009]
- *security testing* [Bezemer et al. 2009]
- *regression testing* [Roest et al.2010]
- *cross-browser compatibility testing* [Mesbah and Prasad 2011]

Nel quarto capitolo verrà fatta un analisi più dettagliata dell'architettura dello strumento che è alla base dello studio effettuato.

2.6.3 JUnit

JUnit è un framework open source progettato per effettuare la scrittura e l'esecuzione dei test cases nel linguaggio di programmazione Java. Originariamente scritto da Erich Gamma e Kent Beck [31], è stato fondamentale per l'evoluzione di sviluppo test-driven, che fa parte di un più ampio paradigma di progettazione software noto come Extreme Programming (XP).

JUnit permette allo sviluppatore di costruire in modo incrementale test suite per misurare i progressi e rilevare i difetti del software. I test possono essere eseguiti in modo continuo ed i risultati vengono forniti immediatamente.

Il framework fornisce un meccanismi di annotazioni per identificare i vari test case all'interno della testsuite e mette a disposizione differenti tipologie di asserzioni atte a verificare se l'esito del codice eseguito all'interno del caso di test rispecchia i risultati attesi. Inoltre, i test JUnit possono essere eseguiti in modo automatico e ognuno di essi fornisce un feedback immediato, e non c'è quindi bisogno di intervenire manualmente attraverso una relazione dei risultati ottenuti. Infatti, durante l'esecuzione JUnit, mostra l'andamento di ciascun test in una barra di progressione che è verde se tutti sta andando come atteso ma diventa rossa appena un test fallisce.

Uno dei principali punti di forza di questo framework sta nella possibilità di definire ed eseguire metodi pezzi di codice, in diverse fasi dell'esecuzione della testsuite utilizzando semplici annotazioni come:

- *@BeforeClass*: una sola volta prima dell'intera testsuite
- *@Before:* prima di ciascun test
- *@After*: dopo ciascun test
- *@AfterClass*: solo una volta dopo l'intera testsuite

Queste annotazioni sono molto utili in quanto permettono di settare precondizioni prima che ciascun test case sia mandato in esecuzione (si pensi ad esempio al ripristino di un database) o magari memorizzare in un file esterno i risultati di ciascun test

immediatamente a seguito della sua esecuzione.

JUnit mette a disposizione molte altre annotazioni che permettono agli sviluppatori di test di personalizzare l'esecuzione delle proprie testsuite in modo semplice ed intuitivo.

Il fatto che non sono necessarie valutazioni soggettive umani o interpretazioni dei risultati dei test, fa si che JUnit si presti ottimamente allo sviluppo di metodologie completamente automatiche.

2.4.4 JSCover

JSCover è un tool che misura la copertura di codice per i programmi JavaScript e rappresenta una implementazione avanzata dell'eccellente strumento JSCoverage [36].

JSCover è un software open source distribuito da GNU GPLv2 (GNU General Public License versione 2) [37].

Il tool lavora instrumentando il codice JavaScript prima che esso venga eseguito all'interno del browser e fornisce numerosi modi alternativi per farlo:

- *Modalità server*: è il metodo più semplice e consiste nell'usare un semplice web server che instrumenta il codice appena è servito al client
- *Modalità file-system*: in questa modalità i file.js vengono replicati nella loro versione instrumentata prima ancora di iniziare l'attività di testing
- *Modalità proxy*: simile alla modalità server ma in questo caso il server funge da proxy e quindi tutto il codice js che viaggia attraverso di esso, verso il client, viene intercettato e instrumentato.

La modalità server (sia web che proxy) ha un duplice vantaggio:

1. È possibile memorizzare i report di copertura sul file system
2. È possibile includere codice JavaScript non caricato o non testato all'interno dei files report.

Lo strumento è in grado di misurare la copertura relativa a linee, salti ed intere funzioni attraverso il browser permettendo l'interazione con il DOM della web application.

Inoltre offre la possibilità di eseguire più test contemporaneamente e unire tutti i dati dei singoli report al termine. I report di copertura possono essere memorizzati in modalità

differenti ed in diversi formati:

- *LCOV*
- *XML summary*
- *Cobertura XML per Jenkins [40]*

JSCover lavora quindi instrumentando il codice JavaScript come nell'esempio che segue:

```
alert('Hello World!');
```

diventerà:

```
//...header code above
if (! this._$jscoverage) {
   this._$jscoverage = {};
}
if (! _$jscoverage['test.js']) {
   _$jscoverage['test.js'] = {};
   _$jscoverage['test.js'].lineData = [];
   _$jscoverage['test.js'].lineData[1] = 0;
}
_$jscoverage['test.js'].lineData[1]++;
alert('Hello World!');
```

L'esempio mostra che quando un pezzo di codice è eseguito, un contatore relativo al file e alla linea viene incrementato. Il codice header gestisce la dichiarazione della variabile _$jscoverage.

Capitolo 3 – Problema e soluzione

3.1 Definizione del problema

L'obiettivo che ci si è posto è la realizzazione di una tecnica di testing completamente automatica che, a partire da una test suite ottenuta con l'ausilio di un tool esterno (Crawljax in questo caso), sia in grado di selezionare e mutare i singoli test cases al fine di massimizzare l'efficacia e migliorare l'efficienza della testsuite finale.

I test generati esternamente costituiscono la prima "versione" della test suite che rappresenta l'input per l'algoritmo realizzato. I test quindi, devono essere rieseguibili e fornire un'adeguata informazione circa i risultati ottenuti.

Trattandosi di una tecnica di tipo automatizzato, il problema dell'efficacia è prioritario rispetto a quello dell'efficienza, che non per questo però è da trascurare come si vedrà nel seguito della trattazione.

3.2 Soluzione proposta

Come soluzione per il problema specificato è stato utilizzato un algoritmo genetico ossia una tecnica di testing basata sulla ricerca di tipo globale, che si ispira alla teoria dell'evoluzione naturale delle specie di Darwin, per far progredire una soluzione iniziale fino a raggiungere un punto di ottimo nei casi in cui l'algoritmo è ben progettato [44].

La decisione di utilizzare una tecnica di questo tipo è scaturita dal fatto che le tecniche di testing automatizzate, attualmente presenti, soffrono di problemi di efficacia. Inoltre è stato dimostrato che le tecniche genetiche applicate al testing hanno raggiunto ottimi risultati in applicazioni basate sugli eventi [46] [47].

3.3 Le componenti principali dell'algoritmo

Data la complessità dell'algoritmo proposto è doverosa una suddivisione in blocchi e una analisi separata tra le varie fasi, ecco una figura esemplificativa delle suddivisioni:

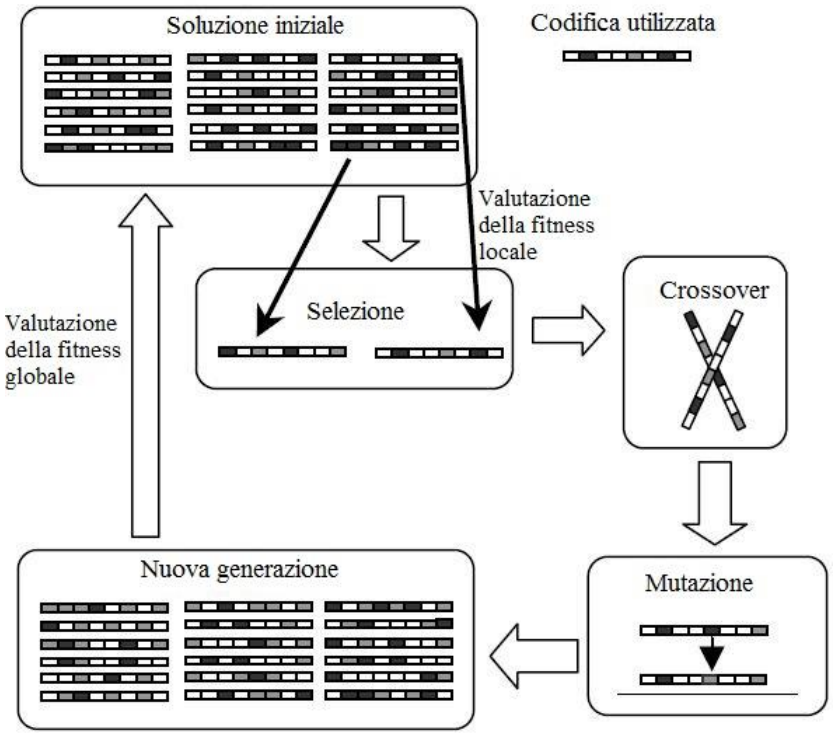


Figura 6

3.3.1 La soluzione iniziale

Ogni algoritmo di ricerca comincia la sua esecuzione a partire da una soluzione iniziale ammissibile.

È stato deciso di utilizzare l'output proveniente da uno strumento che realizzasse una profonda esplorazione dell'applicazione sotto test al fine di generare una prima testsuite che rappresentasse un'ammissibile soluzione iniziale. Come accennato nel capitolo precedente, lo strumento scelto è Crawljax, un tool altamente configurabile dotato di una architettura a plugins attraverso la quale è possibile personalizzare sia il tipo di esplorazione sia la tecnica di generazione di test cases a seconda delle proprio esigenze.

La decisione di utilizzare questo strumento per creare una soluzione iniziale presenta diversi vantaggi in quanto:

- Permette di generare una soluzione ammissibile.
- È una base di partenza con una buona efficacia e quindi permette all'algoritmo di convergere più velocemente.
- La sua struttura plugin-based permette allo sviluppatore di intervenire in ogni fase dell'esplorazione concentrandosi sugli aspetti desiderati
- Durante l'operazione di crawling, memorizza tutte le proprietà e gli attributi relativi agli stati, agli eventi ed ai percorsi (in termine di successione ordinata di stati-eventi) di cui l'algoritmo genetico necessita per poter evolvere.

Lo scopo dell'algoritmo genetico è quello di migliorare l'efficacia della soluzione iniziale manipolando i tests tramite operazioni di crossover e mutazioni singole.

3.3.2 Codifica

Un problema che va affrontato quando si progetta una tecnica di tipo genetico consiste nel rappresentare in modo adeguato i cromosomi (i tests). Nel caso in esame le informazioni da codificare sono le seguenti:

- *Gli stati:* DOM-states, in cui può trovarsi l'applicazione. Crawljax utilizza una struttura dati appositamente definita per modellare uno stato: *StateVertex.* Il nome deriva dal fatto che rappresentano i nodi (vertici) dello state flow graph finale realizzato da Crawljax. Gli *StateVertex* vengono memorizzati in memoria come delle *HashMap* in cui ad ogni id corrisponde un DOM (memorizzato come una Stringa) e sul disco in formato xml all'interno del file *States.xml* in modo da averne una versione permanente. In seguito alla generazione di un evento, sia in fase di esplorazione che di testing effettivo, l'algoritmo confronta l'albero DOM risultante con quello presente prima della generazione dell'evento in ordine di determinare gli eventuali cambiamenti di stato.
- *Gli eventi:* le azioni sul DOM che scatenano il passaggio dell'applicazione da uno stato *A* verso uno stato *B*. Rappresentano gli archi dello state-flow graph e anche per loro è stata definita una apposita struttura da crawljax: *Eventable.* Essi però racchiudono un numero maggiore di informazioni rispetto agli stati, infatti ad

ognuno di essi sono associati:

- *ID* e nome come identificativi univoci
- *XPATH*: posizione esatta all'interno dell'albero DOM (ad esempio: //DIV[1]/SPAN[4])
- *ID Source StateVertex*: id dello stato entrante
- ID Destination StateVertex: id dello stato uscente
- *RelatedFormInputs*: eventuali form input associati, con i valori usati al momento della generazione dell'evento

In memoria vengono usate delle *HashMap* e sul disco anch'essi come gli stati vengono memorizzati in un file xml.

- *Gli input associati agli eventi*. La loro codifica è pressoché simile a quella degli eventi con la differenza che non hanno bisogno di conservare gli id degli stati essendo loro legati univocamente ad un evento che conserva già di suo questo tipo di informazione. Tuttavia devono memorizzare un altro tipo di informazione legata alla tipologia di input, relativa ai tipi definiti dall'HTML (*text, textarea, checkbox, radio, select etc..).*

A questo punto si può dire che ci sono tutte le carte in tavolo per poter definire gli attori dell'algoritmo genetico realizzato, ovvero: *il gene* ed *il cromosoma.*

Il gene lo si può definire come una semplice sequenza: *stato-evento-stato*. Più geni in sequenza vanno a formare un cromosoma, ovvero il test case (i due termini saranno usati indistintamente nel prosieguo della trattazione).

Ogni *cromosoma* è strutturato nel seguente modo (figura 7):

- *Identificativo univoco*
- *Id dello stato iniziale* nel quale si trova l'applicazione in un determinato istante. Per riconoscere uno stato all'interno dell'algoritmo genetico non è necessario conoscere tutto l'albero DOM, ma è sufficiente conoscere l'identificativo assegnatogli da Crawljax durante l'esplorazione.
- *Id degli stati intermedi* che sono attraversati al alla generazione degli eventi

- *Id degli eventi* che si andrà a scatenare. Al cromosoma non interessa il tipo di evento, perché utilizzerà una classe di supporto per generarli ma solo gli eventuali valori ad esso associati.
- *Valori di input* da inserire in eventuali campi di testo presenti nello stato esplorato e associati all'evento da scatenare in quello stato
- *Id dello stato finale* a cui si è arrivati a seguito della generazione di tutti glie eventi della sequenza.

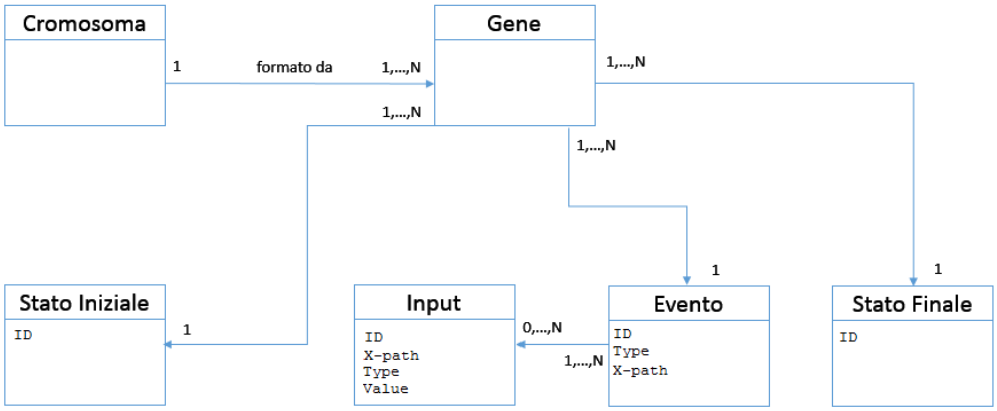


Figura 7

Più formalmente diremo che un gene *G* è definito come:

$$G = \{S, E(I), F\}$$

Dove:

- *S* è lo stato iniziale definito dal suo identificativo unico
- *E* è l'evento che si andrà a scatenare definito come $E = \{p, I\}$ dove p rappresenta l'XPATH dell'elemento da cliccare e I è l'eventuale lista di valore da inserire negli input box del DOM di S
- I è l'insieme degli input, dove ogni input I_j è definito come $I_j = \{p_j, v_j\}$, con p_j XPATH dell'elemento j-esimo del DOM all'interno del quale sarà inserito l'input di valore v_j
- *F* stato finale nel quale l'applicazione si troverà dopo che l'evento E sarà stato scatenato

Definiremo invece un cromosoma C come un insieme di geni:

$$C = \{G_1, \dots, G_n\}$$

Per meglio comprendere come viene codificata la struttura di un cromosoma vediamo un esempio pratico. Consideriamo la sequenza di stati per l'applicazione *TuduList* in figura 8.

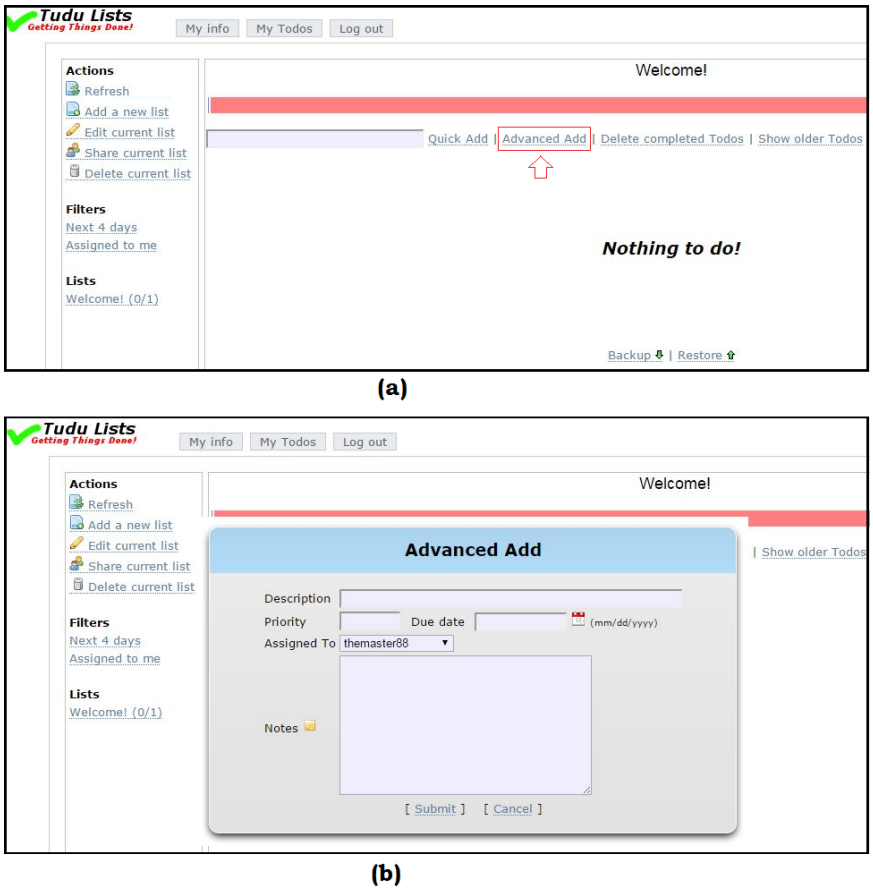
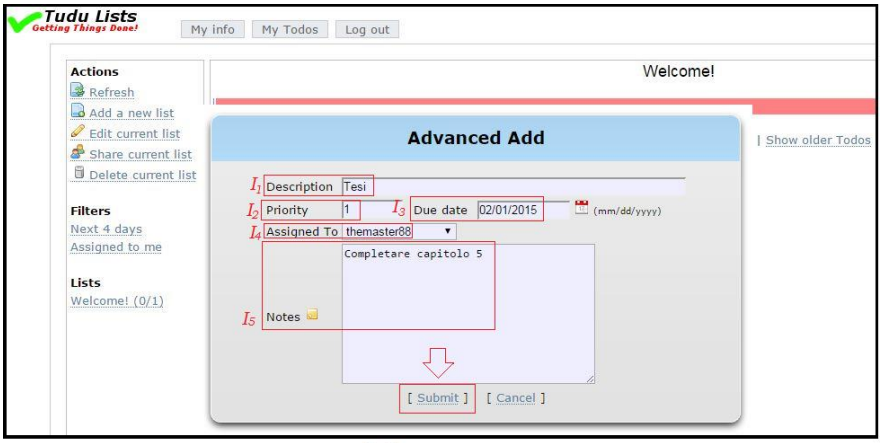


Figura 8

In *fig.7a* è mostrato lo stato iniziale che si suppone essere codificato da *Crawljax* con un nome univoco *'state1'*. Quindi viene scatenato un evento di tipo '*click'* sul DOM con nome '*Advanced Add'* e id '*1*' e l'applicazione si sposta nello stato, codificato con *'state2'*, in *fig.7b*. Le informazioni saranno codificate quindi nel seguente modo:

- Start DOM: *state1*
- Event: click "Advanced Add" , id 1
    - Event input*: null*
- Final DOM: *state2*



(c)

(d)

Figura 9

Nel prosieguo della navigazione, da *fig.7b* inserendo i valori negli apposito campo e cliccando sul pulsante *"Submit"* (*fig.8c*) l'applicazione si sposta nello stato in *fig.8d*. Supponendo che lo stato finale abbia id: *'state4'*, otterremo la seguente codifica:

- Start DOM: *state2*
- Event: click "Submit", id: 2
    - Event input:

{ I_1 : ("description", "Tesi")

I_2 : ("priority", "1")

I_3 : ("date", "2/2/2015")

I_4 : ("assignedTo", "themaster88")

I_5 : ("notes", "completare capitolo 5") }

- Final DOM: *state3*

L'insieme di queste due transizioni *'stato→evento→stato'* forma il cromosoma che

attraversando gli stati: *"state2"* e *"state3"* e generando su di essi rispettivamente gli eventi *"Advanced Add"* e *"Submit"* passa dallo *"stato1"* allo *"state3"*.

Possiamo rappresentare graficamente il tutto nel seguente modo come in Fig.9 :

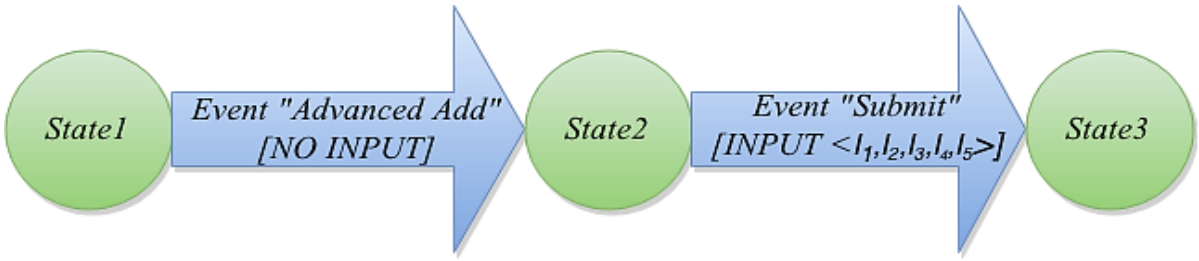


Figura 10

3.3.3 La funzione di fitness globale

Quando si comparano due algoritmi, il valore di efficacia proposto è un indicatore di quanto il processo ha lavorato bene. Ad ogni modo, per assicurare una comparazione equa, è importante definire anche la quantità di sforzo, comunemente misurato tenendo traccia del numero di test che sono stati eseguiti. Per esempio, un algoritmo A potrebbe trovare una soluzione migliore di un algoritmo B, ma allo stesso tempo può capitare che l'algoritmo A abbia impiegato molto più tempo a trovare la sua soluzione migliore rispetto a quanto ne abbia impiegato B. In questo caso si può dire che l'algoritmo A è più efficace di B, mentre B è più efficiente di A.

In letteratura ci sono molti modi di misurare l'efficacia. Solitamente, però, l'efficacia è definita come:

$$\frac{Numero\ di\ malfunzionamenti\ trovati}{Numero\ di\ malfunzionamenti\ da\ trovare}$$

Mentre l'efficienza è definita come:

$$\frac{Numero\ di\ test\ in\ grado\ di\ scoprire\ malfunzionamenti}{Numero\ di\ test\ totali}$$

Nel nostro caso però, non potendo conoscere a priori il numero di malfunzionamenti

presenti nell'applicazione, un'alternativa consiste nel valutare la porzione di codice eseguito per ipotizzare che ad un numero maggiore di linee di codice coperte corrisponda la possibilità di scovare il maggior numero di malfunzionamenti. Diremo quindi che l'algoritmo è tanto più efficace quante più linee di codice riesce a coprire. Possiamo quindi dire che:

$$Efficacia = \frac{Numero\ di\ linee\ coperte}{Numero\ totale\ di\ linee\ valide}$$

Nell'algoritmo genetico sviluppato definiamo due funzioni di fitness: una globale e una locale.

Come detto in precedenza, l'obiettivo principale che ci si è posto durante lo sviluppo dell'algoritmo è di massimizzare l'efficacia della test-suite ma senza trascurare l'efficienza. La misura scelta per calcolare la *fitness* globale della popolazione è di conseguenza rappresentata dalla somma di linee di codice coperte dall'insieme di test generato dall'algoritmo genetico.

3.3.4 La funzione di fitness locale

La funzione di fitness locale calcola la fitness dei singoli cromosomi e, stilando una classifica dei cromosomi in base a questo valore, ci suggerisce quali saranno i cromosomi più idonei ad essere scartati durante il prossimo ciclo di riproduzione.

Per fare ciò, innanzitutto l'algoritmo calcola tre metriche differenti:

1. *Metrica di copertura locale*: numero di linee coperte dal test (cromosoma) rispetto al numero totale di linee valide
2. *Metrica di diversità*: matrice delle diversità per identificare test identici o test interamente inclusi in altri.
3. *Rank*: punteggio calcolato in base al peso assegnato alle righe di codice

La prima metrica viene calcolata come già accennato, grazie a *JSCover* che fornisce informazioni tanto dettagliate da permettere la definizione di un matrice di copertura per ciascun test. Più precisamente per ogni test viene calcolata una mappa che alla posizione

k-esima associa la lista di copertura relativa al k-esimo file JSCover o 0 se nessuna linea di quel file è coperta. La liste di copertura semplicemente associano 1 se la linea è coperta, 0 viceversa. Le mappe di copertura sono indispensabili per il calcolo delle altre due metriche fondamentali nella fase di selezione dell'algoritmo.

Per quanto riguarda la *diversità* è un indice che misura quanto due test differiscono ed è calcolata facendo un confronto fra tutte le matrici di copertura. Più precisamente, detta *D* la *matrice delle diversità*, nella posizione *D[i][j]* è memorizzato il numero di linee coperte dal test *i-esimo* ma non dal test *j-esimo*. Quindi dopo aver calcolato le diversità per ciascuna possibile coppia, analizzando la matrice *D* si evince che:

- se $D[i][j]>0$ e $D[j][i]>0$ allora i test sono diversi (nel senso che coprono linee di codice differenti)
- se $D[i][j]>0$ e $D[j][i]=0$ allora il test j-esimo è incluso (nel senso delle linee di codice coperte) nel test i-esimo
- se $D[i][j]=0$ e $D[j][i]>0$ allora il test i-esimo è incluso nel test j-esimo
- se $D[i][j]=0$ e $D[j][i]=0$ allora i test sono identici (sempre nel senso delle linee di codice coperte)
- se la riga i-esima contiene tutti 0 allora il test i-esimo è incluso in tutti gli altri (equivalentemente la colonna i-esima avrà tutti valori maggiori di 0)
- se la colonna j-esima contiene tutti 0 allora il test j-esimo non è incluso in nessun altro (equivalentemente la riga j-esima avrà tutti valori maggiori di 0)

Le informazioni della matrice vengono utilizzate nella scelta dei test che potenzialmente possono essere scartati perché non contengono informazioni "diverse" rispetto agli altri, portando invece avanti i test migliori, ovvero i "più diversi". La metrica di diversità, essendo chiaramente legata alla copertura locale, come quest'ultima, deve essere ricalcolata ad ogni passo dell'algoritmo genetico.

Per quanto riguarda invece la metrica *rank* bisogna prima di tutto introdurre il concetto di copertura pesata. Questa si ottiene dopo avere assegnato a ciascuna riga un peso *w* inversamente proporzionale al numero di volte che la stessa è stata coperta dai test.

Dato L={l_1, $l_{2, \dots,}$ l_n} l'insieme delle *n* linee di codice di un'applicazione generica e W = {w_1, w_2, … , w_n} l'insieme dei pesi associati alle linee L, allora la fitness locale *rank* di un generico cromosoma sarà definita come:

$$rank = \sum_{1 \le j \le n} l_j w_j$$

Dove:

- con l_j che vale 1 se la i-esima linea è coperta, 0 viceversa.
- $0 \le w_j \le 1/2 \ con \ w_j = 1/\sum_{k=1}^{T} \boldsymbol{m}_k$
  con *T* dimensione della test-suite e $\boldsymbol{m}_k$ che vale 1 se la i-esima linea è coperta dal k-esimo test di T e 0 viceversa.

E' chiaro quindi che un test che copre in modo unico almeno una linea di codice (ovvero una linea non coperta da nessun altro test nella test-suite) avrà un *rank* molto alto, viceversa, un test che copre linee coperte da molti altri test avrà *rank* molto basso.

Ad ogni ciclo *k-esimo* l'algoritmo ricalcola le fitness locali perché ovviamente i crossover e le mutazioni del ciclo *(k-1)-esimo* modificano il comportamento dei test e di conseguenza le loro coperture.

Inoltre, se la diversità guida la scelta dei test da eliminare perché portatori di informazioni ridondanti, la fitness *rank* aiuta a selezionare i testi migliori da sottoporre all'operazione di incrocio.

3.3.5 Tecnica di crossover

La tecnica di *crossover* implementata è di tipo *single-point*, o anche a taglio singolo, e consiste nel determinare un punto esatto attorno al quale viene scambiato il materiale genetico tra due cromosomi, detti *genitori*. In questo modo si genereranno due cromosomi *figli* con corredo genetico funzione dei loro genitori. Più precisamente, il primo cromosoma figlio sarà formato dalla testa del primo genitore e dalla coda del secondo; viceversa, l'altro figlio sarà formato dalla testa del secondo genitore e dalla coda del primo.

Si è già discusso di come un cromosoma, per l'algoritmo genetico proposto, sia codificato tramite un insieme di sequenze '*stato → evento →stato'*.

Il punto di taglio è definito solo sugli stati equivalenti, ovvero sugli stati navigati da entrambi i test.

Definiti due cromosomi X e Y, con:

$$X = \left\{(X_i)^+, (A_n,\ E_1(I_1^*), F_1), \left(X_j\right)^+\right\}$$

$$Y = \{(Y_h)^+, (A_n, E_2(\ I_2^*), F_2), (Y_k)^+\}$$

Dove:

- $(X_i)^+$ e $\left(X_j\right)^+$ sono sequenze di uno o più geni di X
- $(Y_h)^+$ e $(Y_k)^+$ sono sequenze di uno o più geni di Y
- A_n è uno stato navigato da entrambi i cromosomi
- $E_1(\ I_1^*)$ e $E_2(\ I_2^*)$ sono eventi qualsiasi che portano l'applicazione rispettivamente nello stato F_1 e F_2

Avremo che il crossover tra i cromosomi X e Y sarà definito come:

$$XY = \{(X_i)^+, (A_n, E_2(I_2^*), F_2), (Y_k)^+\}$$

$$YX = \{(Y_h)^+, (A_n, E_1(I_1^*), F_1), \left(X_j\right)^+\}$$

Dove:

- XY è il cromosoma figlio generato dalla testa del cromosoma X e la coda del cromosoma Y.
- YX è il cromosoma figlio generato dalla testa del cromosoma Y e la coda del cromosoma X.

Per meglio comprendere le motivazioni del vincolo sul punto di taglio definito solo sugli stati equivalenti, a titolo esemplificativo si supponga di avere due cromosomi genitori X1 e X2 e un taglio sullo stato *"state3"* come in Figura 11.

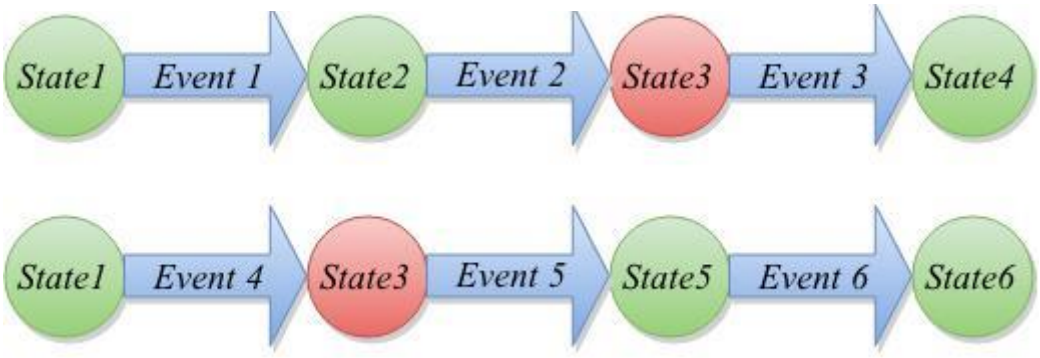


Figura 11

Quindi, sapendo che il punto d'incrocio è rappresentato dallo stato *"state3"*, selezioniamo le teste e le code dei due cromosomi:

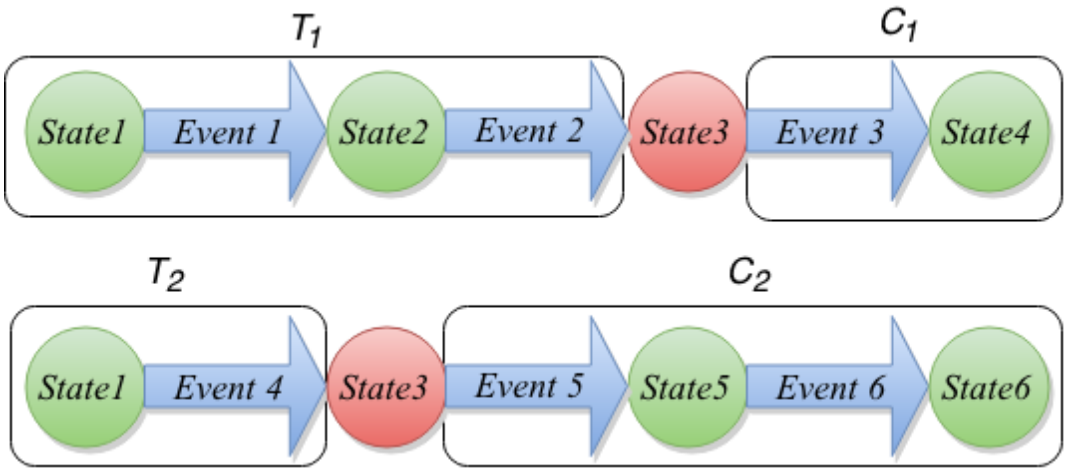


Figura 12

Il *crossover* genera come risultato due cromosomi figli come in Fig.13. Questi due nuovi test generati saranno quindi le due combinazioni possibili di testa e coda, valer a dire: $\{T_1,C_2\}$ e $\{T_2,C_1\}$, così come mostrato in figura:

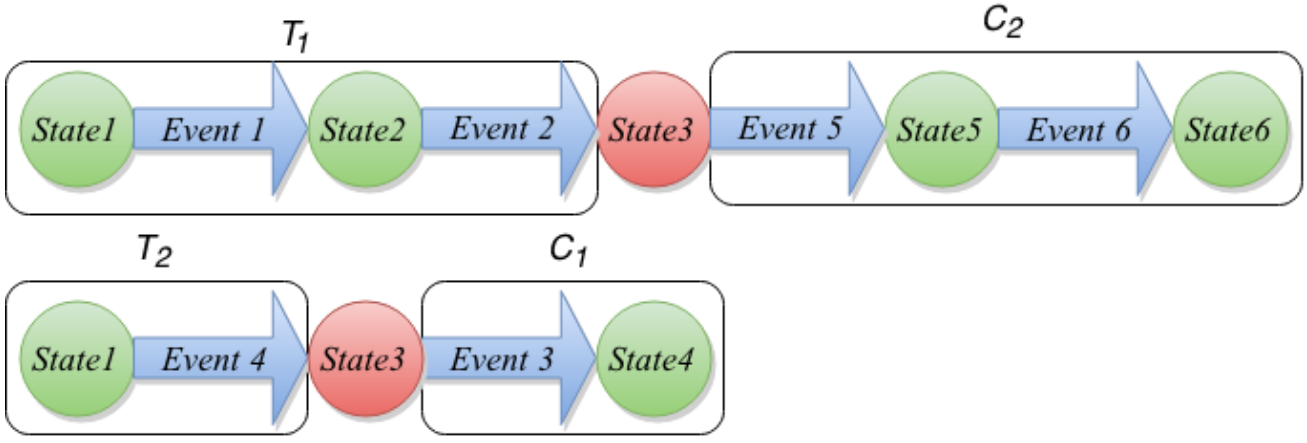


Figura 14

Lo stato di crossover non è scelto in maniera banale, ossia non è sufficiente che due test *i* e *j* abbiano uno stato in comune affinché si possano definire compatibili per l'operazione di incrocio. Le condizioni che devono essere soddisfatte sono le seguenti:

- I test devono avere almeno uno stato *S* in comune
- *S* deve essere diverso sia dallo stato *index* (che rappresenta lo stato iniziale di ogni test) sia dallo stato finale
- La testa T_i deve essere diversa dalla testa T_j
- La coda C_i deve essere diversa dalla coda C_j

Inoltre, diciamo che una testa T_i è diversa da T_j se:

- Contiene almeno un gene diverso (ovvero uno stato o un evento diverso)

Per migliorare l'efficienza del *crossover*, è necessario che tutte le condizioni sopra esposte siano verificate altrimenti si corre il rischio di generare figli che non sono geneticamente diversi dai loro genitori, rendendo vano lo sforzo dell'operatore di crossing.

3.3.6 Tecnica di mutazione

La tecnica di mutazione consiste nel cambiare valore ad un evento di gene di un cromosoma selezionato in maniera pseudocasuale. La mutazione può aiutare l'algoritmo a per esplorare nuovi spazi di ricerca o magari riportare nella popolazione materiale genetico che era andato perduto nelle generazioni precedenti.

Anche la tecnica di mutazione, così come quella di *crossover*, segue le più tradizionali tecniche presenti in letteratura, distinguendosi però in alcuni punti per meglio adattarsi al caso in esame.

Definito un cromosoma X, con:

$$X = \{(X_i)^*, (S, E(I), F), (X_j)^*\}$$

Dove:

- $(X_i)^*$ e $(X_j)^*$ sono una sequenza di geni
- I è la sequenza di uno o più input, associata all'evento E, con:
  - $I = \{I_1, I_2 \dots I_n\} = \{(w_1, v_1), (w_2, v_2), \dots (w_n, v_n)\}$ dove con v_i si indica il valore associato al campo di testo w_i

- $S\ e\ F$ sono rispettivamente stato iniziale, un evento e uno stato finale qualsiasi

Si definisce mutazione, l'operazione che muta un cromosoma X in un cromosoma X_m così composto:

$$X_m = \{(X_i)^*, (S, E(I_m), F), \left(X_j\right)^*\}$$

$$dove\ I_m = \{(I_1)^*, (I_2)^*, \dots, (I_n)^*\} = \{(w_1, m_1)^*, (w_2, m_2)^*, \dots, (w_n, m_n)^*\}$$

In altre parole, la mutazione sugli input consiste nel modificare i valori di input dei campo di testo legati ad un evento di un gene di un cromosoma scelto in modo casuale tra quelli che hanno degli input associati.

Nel caso in esame, infatti, non ha senso mutare un qualsiasi gene di un cromosoma poiché, come nel caso del *crossover*, ci sarebbe un alto rischio di creare un test non eseguibile. Infatti, se la mutazione non fosse applicata agli input bensì agli eventi, con ampia probabilità si genererebbero test non eseguibili in quanto l'evento potrebbe non appartenere a quello stato.

Quindi, i test che possono essere modificati devono avere almeno un evento a cui è associato almeno un input.

Nella fase di mutazione viene scelto in modo casuale un test tra quelli presenti nella *test-suite* che inseriscono almeno un input, e se ne modifica il testo da inserire. Nel caso in cui il test inserisca più input, vengono mutati tutti, e questa scelta è stata ponderata e scaturita dalla fase di sperimentazione che ha mostrato che le singole mutazioni molto raramente riuscivano a portare informazioni diverse da quelle del test originale.

I possibili valori di input da inserire sono scelti in modo casuale da una lista popolata secondo il criterio delle classi di equivalenza. Non potendo però conoscere a priori il tipo di input richiesto dal campo di testo, sono state coperte le classi più probabili da trovare all'interno di un'applicazione web, come:

- Indirizzi: indirizzi validi, non validi o coordinate geografiche
- Nomi o cognomi validi e/o di personaggi di interesse

- Numeri: zero, interi, reali, decimali, negativi, esponenziali o molto grandi
- Percorsi dove memorizzare ad esempio file: percorsi Windows, Linux, Crawljaaax
- Numeri di telefono: con o senza prefisso, con e senza spaziature tra i numeri
- Informazioni utili per eventuali registrazioni: email valide o non valide, username, password numeriche, alfanumeriche, corte o molto lunghe
- Stringhe con caratteri speciali
- Date: secondo le convenzioni italiane, americane, date molto vecchie, date future, date in formato esteso o numeriche.
- Orari validi o meno, ore divise dai punti o dai due punti

3.4 L'algoritmo genetico proposto

Nei paragrafi precedenti sono stati descritti i componenti principali della tecnica genetica. Di seguito è presentata una descrizione ad alto livello dell'algoritmo che mostra i principali passi dell'algoritmo genetico implementato.

Algoritmo 1 – Pseudo-codice dell'algoritmo genetico proposto

```
procedure
TestSuite T(0) = getCrawljaxTestsuite();
Execute(T(0));
setConfigParameters(crosoverRate, mutationRate, Random, impulseFrequence, turnOverRate, numMaxCycles,  numMinTestCases, numMaxTestCases, desiredRate);
indexCycle=0;
while(!satisfiedCondition()) {
    evaluateGlobalFitness(T(i)); //i=indexCycle
    evaluateLocalFitness(T(i));
    numCross=0;
    while(numCross<crossoverRate){
```

```
        crossover();
        numCross++;
}
numMut=0;
        while(numMut<mutationRate){
        mutation(Random);
        numMut++;
}
T(i+1)= (T(i)); //al ciclo successivo
Execute(T(i+1));
for(test t : allMutatedTestCases(T(i+1))){
improvements=evaluateImprovements(t);
if(improvements<0)
        t.goBack();
          totalImprovements=+improvements;
}
if(totalImprovements  < turnOverThreshold && indexCycle%impulseFrequece==0){
                turnover();
        }
        indexCycle++;
}
globalFitness ← evaluateGF (T(i));
```

Di seguito sono riportate brevemente le definizioni dei termini utilizzati all'interno dello pseudo-codice:

- *crossoverRate*: la percentuale di crossover. Indica la percentuale di turnover da destinare all'operatore di crossover.
- *mutationRate*: percentuale di mutazione .

- *T(i)*: rappresenta la testsuite e contiene gli abitanti della popolazione all'iterazione *i*-esima.
- *improvements:* è legato ad un singolo test mutato. Viene calcolato solo se il test è stato mutato al ed indica la differenza di lines rate tra prima e dopo la mutazione.
- *totalImprovements:* è la sommatoria di tutti gli *improvements* di tutti i cicli
- *turnOverRate*: rappresenta la percentuale di turnover, ovvero la porzione di popolazione che sarà sostituita ad ogni iterazione.
- *turnOverThereshold:* soglia minima di miglioramento da raggiungere per evitare un nuovo impulso di turn over.
- *impulseFrequence*: scandisce frequenza di impulso di turnover, che avviene però solo se l'algoritmo non ha raggiunto la soglia minima di miglioramento (*turnOverThereshold*)
- *numMinTestCases:* dimensione minima della popolazione.
- *numMaxTestCases*: dimensione massima della popolazione. Rappresenta la dimensione massima della test-suite finale
- *localFitness*: contiene la fitness locale dei cromosomi.
- *globalFitness*: contiene la fitness globale dei cromosomi.
- *numMaxCycles*: numero massimo di iterazioni eseguibili dall'algoritmo genetico.
- *random:* è un flag booleano che decide la scelta dei test da mutare in modo sistematico o random, la logica di controllo dell'algoritmo setta questo parametro ad ogni iterazione in base al numero di test mutabili presenti.
- *desiredRate:* percentuale di lines rate totale che si desidera raggiungere

Prima che l'algoritmo inizi la sua evoluzione, la TestSuite *T(0)*, generata con l'ausilio di Crawljax viene eseguita per ottenere la copertura di partenza. Inoltre al tester viene esplicitamente chiesto se effettuare un settaggio manuale dei parametri che guidano l'evoluzione genetica.

Nel blocco iterativo vengono dapprima calcolate le funzioni di *fitness* locali e globali.

Dopo di che intervengono le fasi di *crossover* e *mutazione* seguite dall'esecuzione della *test-suite* formata dai nuovi individui della popolazione. Al termine dell'esecuzione, vengono valutati gli eventuali miglioramenti, e nel caso di peggioramenti per ogni singolo cromosoma è previsto un ripristino allo stato precedente la mutazione. Se la percentuale di miglioramenti si mantiene troppo bassa viene dato un impulso che aumenta il valore di *turn over.*

3.4.1 Tecnica di selezione

La tecnica di selezione si occupa di selezionare i cromosomi da far riprodurre e, allo stesso tempo, scegliere quali cromosomi dovranno invece essere eliminati per far posto ai nuovi alle nuove generazioni.

Durante la fase di crossover, la tecnica di selezione si occupa di selezionare i cromosomi, *genitori*, da incrociare per dar vita così a due *figli*. In questa selezione sono preferiti i test che hanno un *rank* maggiore, anche se in molti casi, la difficoltà nel trovare cromosomi compatibili costringe l'algoritmo a fare una ricerca sistematica.

Inoltre vengono selezionati una coppia di cromosomi da *eliminare*, scelti tra quelli con la *diversity* più bassa, per fare posto ai cromosomi figli generati, così come avviene nella *Steady-State reproduction*. L'alternativa, invece, è di eliminare i cromosomi *genitori*, in modo da non avere nella test-suite sia i cromosomi *genitori* che i loro *figli*. Questa seconda modalità è utilizzata nelle tecniche genetiche più tradizionali.

Per quanto riguarda la selezione dei test da mutare questa viene effettuata in due modalità: random o sistematica. La prima è quella utilizzata di default e va a selezionare, in maniera casuale, individui che non siano stati già mutati al ciclo corrente e che presentano almeno un input associato ad un evento. Se la modalità random fallisce troppo spesso, ossia il numero di tentativi falliti supera una soglia prestabilita, perché ad esempio il numero dei test mutabili è diventato troppo basso, la logica di controllo passa ad una modalità sistematica che velocizza l'operazione di selezione.

I procedimenti di ricombinazione e mutazioni singole sono entrambi iterati fino al raggiungimento delle rispettive percentuali di *crossoverRate* e *mutationRate* che possono

cambiare nel corso di ogni iterazione dell'algoritmo.

3.4.2 L'impulso di turnover

Il problema principale di un algoritmo di ricerca euristico, come quello genetico, è di evitare di arenarsi in un punto di massimo locale da cui è difficile uscire.

Per le tecniche genetiche invece si cerca di sfuggire ai punti di massimo locale alzando, per una iterazione, la percentuale di turnover [49]. Aumentando la percentuale di turnover, infatti, si ha la possibilità di esplorare uno spazio di ricerca più ampio. Tale percentuale, però, deve ritornare ad un valore più basso altrimenti le prestazioni dell'algoritmo peggiorerebbero in quanto la tecnica diventerebbe più simile a una tecnica di ricerca casuale [48].

Nella tecnica genetica proposta si è deciso di implementare una strategia, chiamata "*impulso di turnover*", che permette di aumentare il valore di turnover in maniera dinamica. Quando la *fitness* globale rimane stabile per un numero determinato di iterazioni si ipotizza che l'algoritmo è bloccato su un punto di massimo che può essere locale. Si ha allora una nuova fase di ottimizzazione della test-suite, come quella vista nel paragrafo 3.4.2, in cui si eliminano tutti i test che hanno una copertura, in termini di linee di codice, inclusa in almeno un altro test. Ogni test eliminato in questa fase sarà poi rimpiazzato da un altro test, generato tramite gli operatori di crossover o mutazione.

3.4.3 Nuovi stati

Durante le operazioni di crossover o mutazione è possibile che l'applicazione testata giunga in uno stato nuovo rispetto a quelli già esplorati da *Crawljax* o dalle altre operazioni di crossover e mutazione effettuate in precedenza durante l'algoritmo di ricerca.

Per stabilire se uno stato è nuovo rispetto a quelli già incontrati in precedenza si confronta lo stato attuale del test con quello degli stati già navigati. Se l'insieme non combacia con nessuno degli insiemi degli stati navigati, allora lo stato attuale è considerato nuovo.

3.4.4 Condizioni di terminazione

L'algoritmo genetico, una volta iniziata la sua evoluzione, prosegue autonomamente senza ausilio di intervento esterno. E' chiaro quindi che sono state previste delle condizioni di terminazione, ossia:

- l'algoritmo ha eseguito tutte le iterazioni previste (*indexCycle=numMaxCycles*)
- l'algoritmo ha raggiunto la soglia minima di *linesRate* desiderata
- impossibilità di effettuare operazioni di crossing e mutazioni

Quest'ultima condizione si verifica qualora l'algoritmo non è in grado né di trovare nuovi tests compatibili per il crossover né test mutabili, ad esempio perché non ci sono test con input associati o tutti i test sono stati già mutati con successo.

3.4.5 I parametri

È possibile lavorare su diversi parametri per modificare le prestazioni dell'algoritmo genetico. I parametri possono essere impostati dal tester o automaticamente secondo una configurazione di default che dipende dalla popolazione della test suite iniziale.

I principali parametri sono i seguenti:

- *Percentuale di turnover.* Indica la percentuale di popolazione che in ogni iterazione viene sostituita per lasciar posto a nuovi cromosomi generati tramite crossover e mutazioni. Un valore eccessivamente alto di questo parametro rischia di muoversi troppo lungo lo spazio di ricerca senza però mai trovare gli ottimi. Al contrario, con un valore basso si rischia di rimanere bloccati più facilmente su un ottimo locale.
- *Percentuale di crossover*. Indica, tra tutti i test che saranno generati durante un ricambio generazionale, quale percentuale sarà destinata a essere generata tramite crossover.
- *Percentuale di mutazione*. Così come per la percentuale di mutazione di crossover, questo parametro indica la percentuale di turnover da destinare alla mutazione degli input.

- *Tecnica di mutazione*. La tecnica di mutazione implementata consente di scegliere se il cromosoma mutante andrà a sostituire il cromosoma mutato o se il cromosoma mutante sostituirà un cromosoma con bassa *fitness* locale. Inoltre l'algoritmo può cambiare la modalità di selezione dei test da mutare da random a sistematica.
- *Numero massimo di iterazioni*. Una prima condizione di terminazione della tecnica genetica prevede che l'algoritmo abbia un limite massimo di iterazioni. Una volta raggiunto questo limite, si ipotizza di non voler investire altre risorse di tempo nella ricerca dell'ottimo, quindi si termina l'algoritmo.
- *Numero di impulsi di turnover massimi*. La tecnica genetica può prevedere come altra condizione di terminazione il verificarsi di un numero precisato di occorrenze di impulsi di turnover. Gli impulsi sono utilizzati per cercare di smuovere l'algoritmo da un punto stazionario di ottimo locale verso un ottimo "migliore".
- *Frequenza di impulso*. Questo parametro rappresenta il numero di iterazioni da attendere prima di dare un nuovo impulso di turnover se la fitness globale si mantiene al di sotto di una soglia prestabilita. Un valore troppo basso di questo parametro rischia di eliminare troppo presto dalla test-suite i test generati tramite crossover che, pur non portando contributo immediato alla fitness globale della popolazione, se incrociati con un altro test potrebbero portare nuovo contributo genetico. Viceversa, un valore troppo alto rischia di lasciare fermo l'algoritmo per troppo tempo su un punto stazionario e allungare il tempo di esecuzione dello stesso prima di trovare l'ottimo.
- *Dimensione minima della test-suite*. Questo parametro determina la dimensione minima, in termini di test, della test-suite. Questo valore dovrebbe essere scelto in base ai precedenti valori dei parametri di percentuali di turnover, crossover e mutazione per garantire che, per ogni iterazione, sia possibile avere almeno un'occorrenza di crossover e mutazione.

Il problema dell'assegnazione dei valori a questi parametri sarà ripreso nel cap.5 in modo euristico.

Capitolo 4 - L'architettura dello strumento

Per descrivere l'architettura dello strumento di testing genetico sviluppato è importante innanzitutto conoscere l'architettura di *Crawljax* che si trova alla sua base, per analizzare poi come si interfacciano i sistemi.

4.1 Architettura di Crawljax

Crawljax, come già accennato in precedenza, propone un tipo di analisi dinamica, la quale si pone come obiettivo la costruzione di un grafo di flusso degli stati che sia in grado di descrivere il comportamento runtime del software. Per raggiungere lo scopo vengono svolte le seguenti azioni: l'applicazione AJAX viene eseguita in un browser, l'albero DOM viene esaminato per trovare gli elementi capaci di cambiare lo stato, si generano gli eventi legati agli elementi selezionati e si analizzano gli effetti sul DOM.
La struttura interna della singola pagina (DOM) rappresenta uno stato, da ciò ne deriva il fatto che ad uno specifico URL possono corrispondere più stati, supponendo appunto che il DOM stesso assuma una forma diversa per ognuno di essi. Si evince che parlare di cambio di stato significa parlare di modifica del DOM, e quest'ultima può essere causata da:

1. eventi scatenati dal motore AJAX lato client
2. cambiamenti lato server propagati al client

Sul browser ogni volta che si scatena l'*evento click* il tool controlla che si sia verificato un cambiamento sul DOM, vale a dire una transazione tra due stati [30].
Il grafo di flusso finale quindi si ottiene registrando tutti gli stati e tutti gli eventi che hanno generato le transazioni tra essi. In questo contesto è doveroso dare una definizione

più attenta di state-flow graph:

data *A* una generica applicazione AJAX composta di una sola pagina html allora

"*Uno state-flow graph G per un applicazione AJAX A è un grafo diretto definito da una quadrupla <r,V,E,L>"* dove:

- *r* è il nodo radice (Index) rappresentante lo stato iniziale, quando A è completamente caricata nel browser
- *V* è il set di vertici, stati. Ogni v ∈ V rappresenta uno stato-DOM a tempo d'esecuzione di A
- *E* è il set di archi tra i vertici. Ogni arco $(v_i, v_j) \in E$ rappresenta un elemento cliccabile c che connette due stati se e solo se lo stato v_j è raggiungibile tramite l'esecuzione di c in v_i
- *L* è la funzione che assegna un etichetta da un set di eventi tipo e le proprietà del DOM per ogni arco
- *G* può essere muti-arco e ciclico

Un esempio di grafo è mostrato nella figura sottostante che rappresenta il flusso di stato di un semplice sito AJAX. Da notare l'etichettatura degli archi che sono identificati o dagli attributi o dagli XPath degli elementi cliccabili.

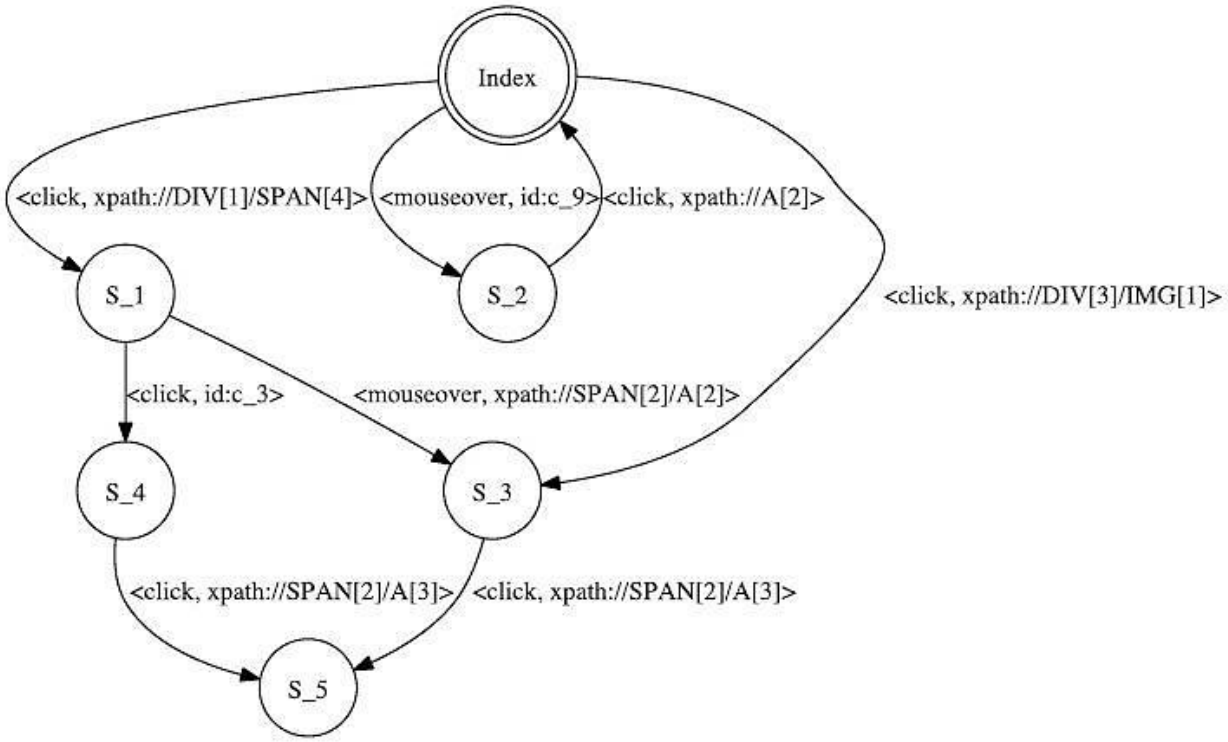


Figura 14

Lo state-flow è generato in maniera incrementale: inizialmente esso contiene solo il nodo radice (stato: Index), successivamente, durante l'esplorazione e l'analisi dei cambiamenti dell'interfaccia, i nuovi stati sono creati e aggiunti al grafo.

I seguenti componenti, come anche mostrato in figura 15, collaborano alla costruzione del grafo degli stati:

- *Embedded Browser.* Il browser integrato fornisce un interfaccia comune per l'accesso ai motori sottostanti e agli oggetti dinamici, come DOM e JAVASCRIPT
- *Robot.* Il robot è usato per simulare le azione dell'utente (e.g., click, passaggio del mouse, inserimento testo nei form input) nel browser.
- *Controller.* Il controller ha l'accesso al DOM del browser, controlla anche le azioni del Robor, ed è anche responsabile dell'aggiornamento dello stato della macchina quando ci sono cambiamenti rilevanti del DOM
- *DOM Analyzer.* L'analizzatore è usato per controllare ogni volta che l'albero DOM ha subito cambiamenti dopo che un evento è stato generato dal robot. Inoltre, è usato per confrontare gli alberi DOM ed evitare il problema della duplicazione degli stati.
- *Finite State Machine.* La macchina a stati finiti è il componente che si occupa della generazione dello state-flow graph mantenendo un puntatore allo stato corrente (cioè quello sotto esplorazione).

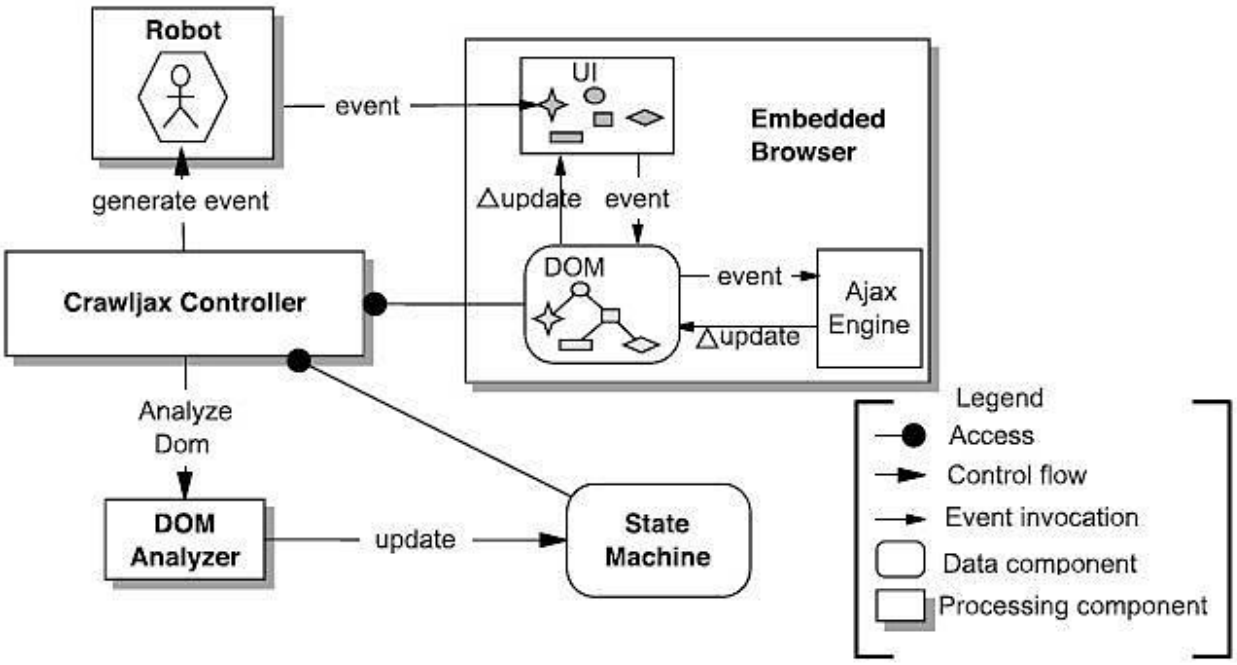


Figura 15

E' chiaro quindi che, il processo di Crawljax, essendo del tutto automatico e avendo come output finale il grafo degli stati, rappresenta un valido supporto alle attività di testing automation. Crawljax è stato progettato con un design modulare che lo rende facilmente modificabile ed evolvibile [50]. Inoltre la sua architettura plugins-based, che garantisce una ampia personalizzazione dello strumento, ha permesso, in modo semplice, il passaggio dalla procedura di crawling a quella di testing, realizzata per l'appunto nel corso di questo studio.

4.1.1 Crawljax Plugins

Affermare che Crawljax ha un struttura plugin-based vuol dire che nell'arco dell'intera fase di esplorazione (crawling) ci sono vari punti di estensione. L'interfaccia principale è *Plugin* la quale è estesa da vari tipi di plugins disponibili ognuno dei quali può essere richiamato in una delle differente fasi dell'esecuzione dell'algoritmo di crawling. Il settaggio dei plugins che si intende utilizzare va fatto nella fase di configurazione dello strumento, contemporaneamente al settaggio delle altre opzioni di base che Crawljax mette a disposizione del tester come ad es. numero massimo di stati che si vuole esplorare,

o di eventi che si vuole generare, tempo minimo e/o massimo etc. Ovviamente lo sviluppatore deve aver delle conoscenze di programmazione avanzata per poter implementare e personalizzare i plugins a seconda delle proprio preferenze.

Siccome i plugins di crawljax hanno rappresentato un punto di partenza importante per questo studio, nella tabella che segue ne sono mostrati i principali e per ognuno di essi è indicata la fase esatta nella quale vengono richiamati e un possibile esempio d'utilizzo:

TIPO	**EVENTO**	**ESEMPI**
PreCrawlingPlugin:	Prima di caricare l'URL della web-app da testare	Fare il log in. Configurare un proxy per intercettare le richieste e le risposte
OnUrlLoadPlugin:	Dopo che l'URL iniziale è stato (ri) caricato	Resettare lo stato di back-end
OnFireEventFailedPlugin:	Dopo aver generato un evento	Analizzare quando l' evento generato è fallito
DomChangeNotifierPlugin:	Dopo aver generato un evento	Notificare al crawler che l'evento generato ha portato in un nuovo stato
OnInvariantViolationPlugin:	Quando una violazione di un invariante viene rilevata	Analizzare l'origine della causa. Riportare l'invariante fallito.
OnNewStatePlugin:	Quando un nuovo stato è rilevato durante l'esplorazione	Fare uno screenshots o validare il DOM
OnRevisitStatePlugin:	Quando uno stato è rivisitato	Benchmarking, analisi

PreStateCrawlingPlugin:	Prima che un nuovo stato è esplorato	Settare gli elementi candidati (i possibili eventi da generare)
PostCrawlingPlugin:	Alla fine dell'esplorazione	Generare dei test case dal flow-graph

E' proprio quest'ultimo, come anche chiaramente indicato dal suggerimento d'utilizzo, ad aver fatto da collante tra crawling e testing.

4.2 Il processo di testing

Nel corso di questo paragrafo si vuole dare una vista globale dell'intero processo che mette insieme la fase di esplorazione e quella di testing effettivo (figura 16).

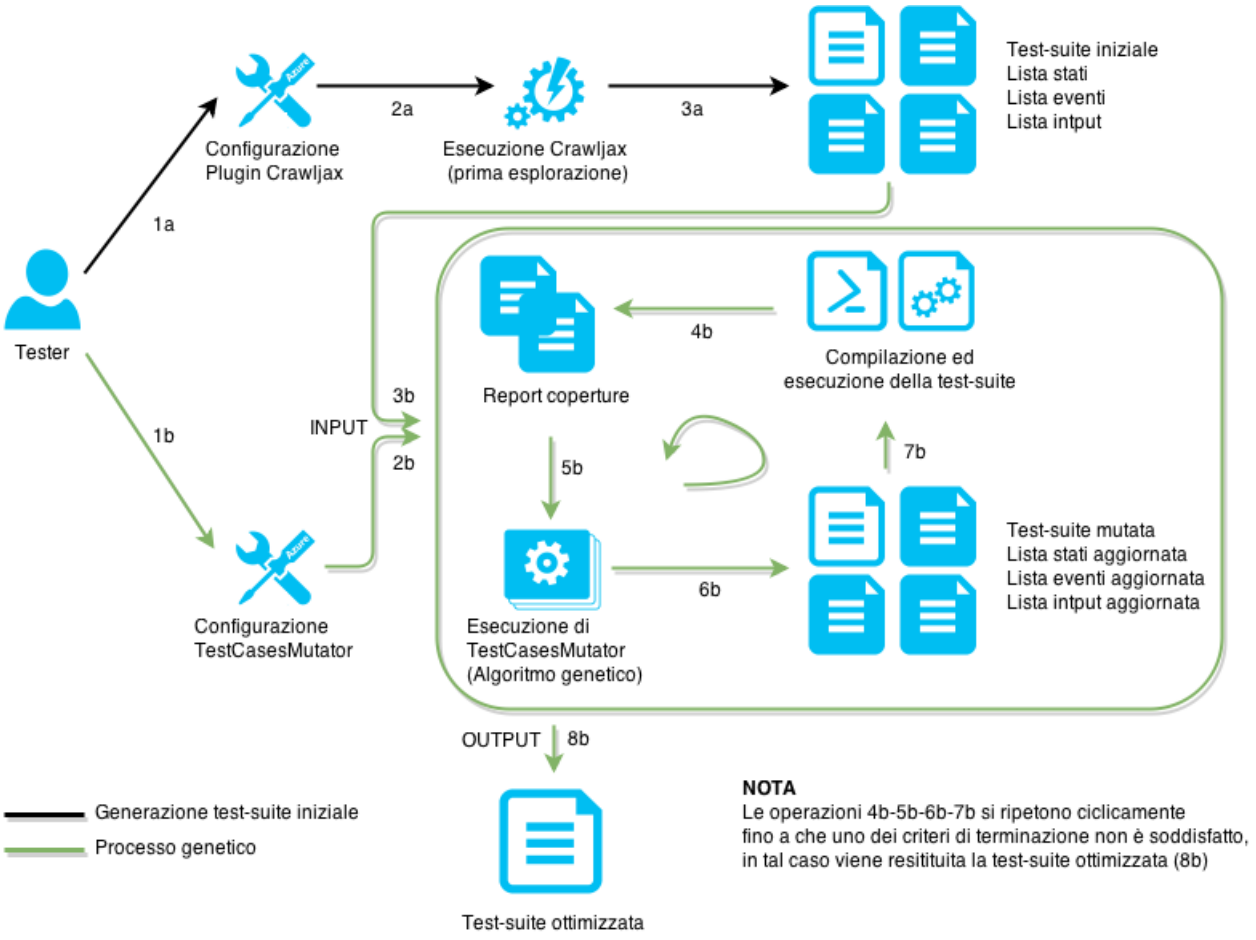


Figura 16

Il tester dapprima setta i parametri di configurazioni di Crawljax che danno comunque una misura approssimativa sulla quantità e sulla complessità (in termini di eventi) dei test cases che si intende produrre come testsuite iniziale.

Gli output del tool automatico sono quattro:

1. *GeneratedTestCases.java:* test cases in formato JUnit
2. *States.xml:* elenco degli alberi DOMs relativi agli stati esplorati e salvati in formato xml
3. *Eventables.xml:* elenco degli eventi generati durante la fase di esplorazione
4. *Input.xml:* valori associati ai soli eventi dotati di form input

Il passo successivo consiste nella compilazione ed esecuzione delle test-suite ed è reso possibile grazie a due file batch: *compile.bat* e *execute.bat.* Il primo, richiamando compilatore *javac* (che fa parte del JDK [51]) trasforma il codice sorgente *.java* in Java Virtual Machine byte code (files *.class*). Il secondo invece, per lanciare i files *.class* appena compilati utilizza la classe *JUnitCore* di JUnit (utilizzando la dichiarazione *org.junit.runner.JUnitCore*) e tutte le librerie appositamente esportate dall'applicazione e ubicate in una cartella della quale viene specificato il path nello script. I due script realizzati sono stati definiti in modo da essere riutilizzabili, vale a dire che è sufficiente cambiare i valori della stringa di comando al momento in cui vengono richiamati, per compilare ed eseguire files diversi in locazioni differenti. Questa si è verificata essere una proprietà di primaria importanza essendo che i files contenenti i tests subiscono modifiche ad ogni passo del ciclo dell'algoritmo e quindi devono essere ogni volta ricompilati e rieseguiti automaticamente.

Al termine di ogni esecuzione, vengono generati in modo ordinato, tanti files di copertura quanti sono i test. In seguito alla configurazione del tool implementato, automatica o da parte del tester, il tool genetico ha tutte le informazioni necessarie per poter iniziare la sua evoluzione. Ad ogni passo quindi viene generato un nuovo file *MutatedTestCases.java* contenente i test case mutati ed eventualmente i nuovi creati, seguono la ricompilazione e la riesecuzione, nuove coperture e così via finché non vengono soddisfatte le condizioni di

terminazione.

Nel corso del paragrafo successivo viene fatta un analisi più dettagliata delle classi che collaborano alla realizzazione dell'intero processo in questo ambito diviso in: generazione, esecuzione e mutazione

4.3 Descrizione dello strumento implementato

Il tool di testing genetico sviluppato è stato progettato per soddisfare i seguenti requisiti:

- Deve poter ricevere generare, mutare e rieseguire i test cases, che all'interno del tool sono sia dati di output che di input.
- Deve essere altamente personalizzabile, il tester può così decidere quali strategie di testing genetico utilizzare (ad esempio se i cromosomi da scartare saranno i cromosomi genitori o i cromosomi con più bassa fitness locale), il criterio di terminazione dell'algoritmo, i valori delle percentuali di turnover, crossover e mutazione, ecc…
- Deve poter restituire in output delle test-suite JUnit per poterle rieseguire

Per soddisfare questi requisiti, è stato sviluppato uno strumento di testing genetico composto da quattro packages principali come è possibile vedere in figura 17.

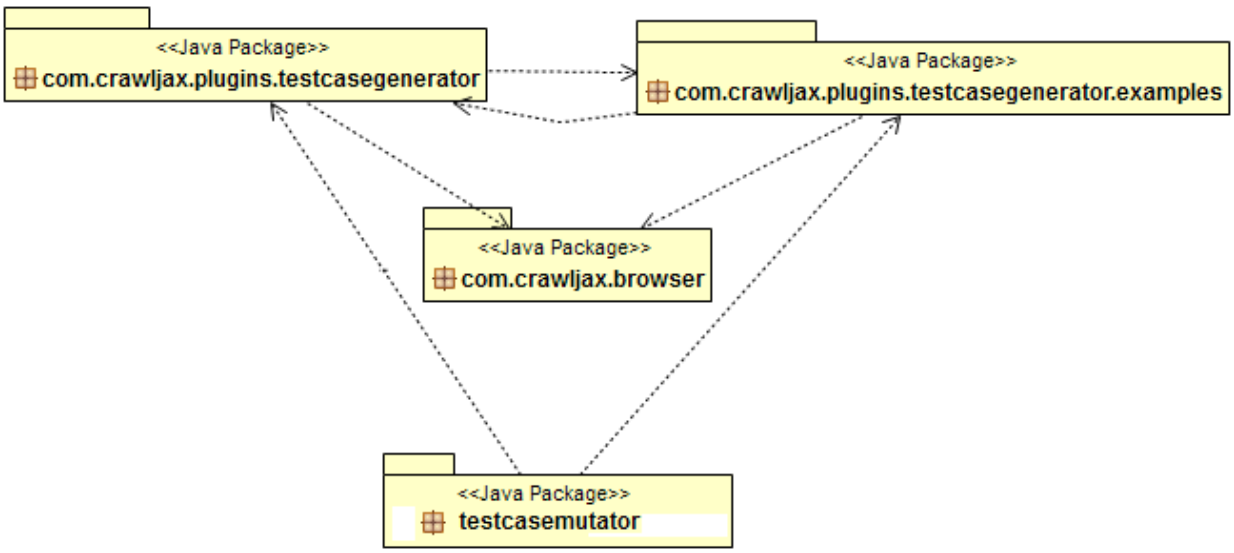


Figura 17

Il primo, "*testcasegenerator*" contiene tutte le classi realizzate allo scopo di richiamare e utilizzare le librerie fornite da crawljax per eseguire la prima esplorazione. Inoltre, sempre all'interno di questo package, sono state definite tutte le classi che collaborano alla generazione della testsuite iniziale e la scrittura della stessa all'interno di un file java.

Il package *"examples"* racchiude tutte le classi che implementano alcuni dei principali plugin messi a disposizione da crawljax che vengono richiamati sia in fase di esplorazione che di testing. Il pacchetto *"browser"* è stato interamente importato dal progetto principale di crawljax, ma le classi in esso contenute sono state estese allo scopo di permettere al tool realizzato di accedere al Web Driver in fase di esplorazione.

Infine, il pacchetto *"testcasemutator"* comprende tutte le classi più complesse dell'intero progetto che contribuiscono all'implementazione dell'algoritmo genetico realizzato.

Nei prossimi paragrafi saranno discussi i vari passaggi del processo implementato, presentando le classi protagoniste con alcuni riferimenti ai metodi principali ed alle interconnessioni fra i vari moduli.

4.3.1 Generazione della testsuite di partenza

La figura sottostante mostra le classi che intervengono e collaborano alla realizzazione della prima testsuite.

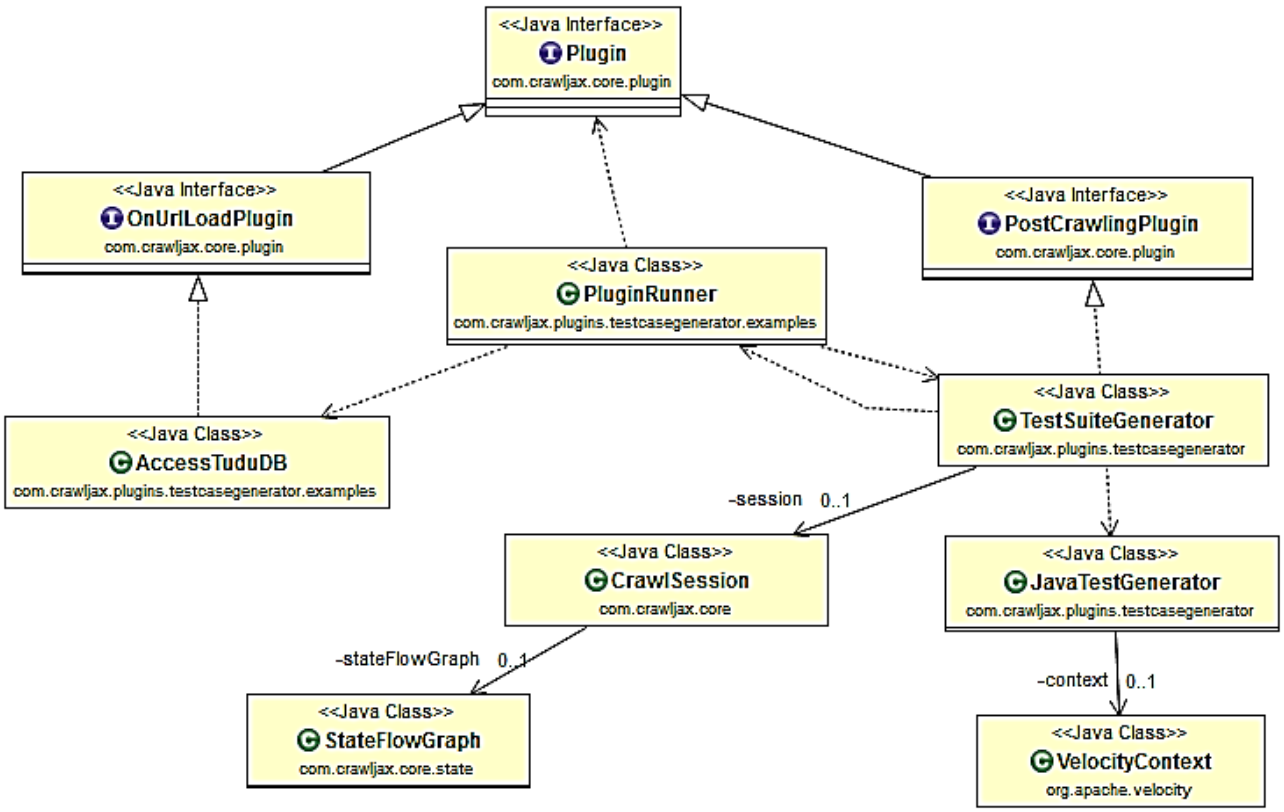


Figura 18

La classe *PuginRunner* (al centro della figura 18) rappresenta il punto di partenza dell'algoritmo, e con la quale si interfaccia il tester per i settaggi iniziali, tra cui:

- inserimento dell'URL dell' applicazione web da testare
- creazione dei sottopercorsi per il salvataggio sul disco dei files java contenenti i test cases e dei files xml contenenti informazioni sugli stati e gli eventi
- parametri di configurazione di crawljax (numero massimo di stati da visitare, numero massimo di eventi da generare, tempo massimo di esplorazione, tipologia di eventi etc…)
- eventuali configurazioni del browser (tipo Chrome, Firefox, … , eventuale proxy e porto)
- aggiunta dei plugins da utilizzare durante la fase di crawling

In seguito alla fase di configurazione, Crawljax viene avviato e i plugin, precedentemente implementati e settati, vengono richiamati automaticamente allo scatenarsi dei corrispondenti eventi trigger elencati in tabella. A tal proposito è doveroso menzionare *OnUrlLoadPlugin* (implementato dalla classe *AccessTudu* come mostrato in figura) che

viene richiamato per l'accesso all'applicazione (*Tudu Lists* in questo esempio) altrimenti l'esplorazione non potrebbe andare oltre la pagina di login. Inoltre questo stesso plugin è utilizzato anche per il ripristino delle precondizioni, (ripristino del database) per far si che ogni esplorazione (che successivamente sarà rappresentata da un test case) parta sempre con le stesse condizioni iniziali.

L'altra interfaccia di Plugin mostrata nella figura 18 è *PostCrawlingPlugin* che rappresenta il vero punto di partenza per la procedura di testing ed è implementata dalla classe *TestSuiteGenerator*. Quest'ultima ha accesso alla sessione di crawl e tutti i parametri che la caratterizzano, tra cui lo *StateFlowGraph,* che viene letto quindi al termine della procedura di crawling. La classe *TestSuiteGenerator* estrapola tutte le informazioni da grafo, memorizza una parte di esse sul disco, da vita ai primi test cases e invia tutto alla classe che *JavaTestGenerator* che effettua la scrittura su di un file java. Per compiere tutte queste operazioni, di cui alcune lunghe complesse, utilizza una classe "di supporto", *TestSuiteGeneratorHelper,* come mostrato in figura 19:

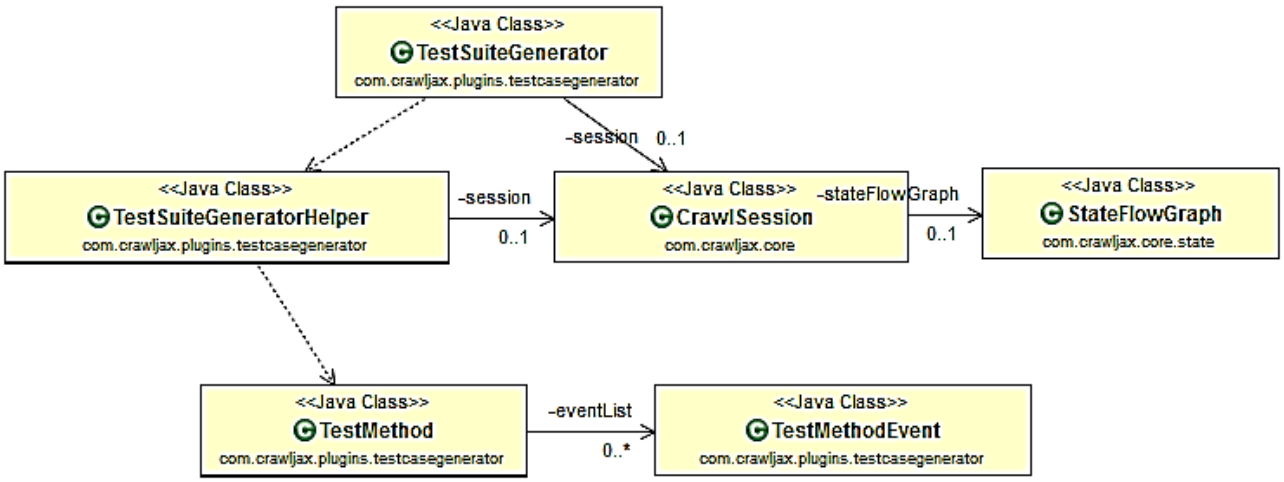


Figura 19

Alla classe Helper viene passata l'istanza di sessione e quindi anch'essa ha accesso al grafo degli stati. La classe di supporto ha i seguenti compiti:

- salvataggio sul disco degli stati (nodi del grafo), degli eventi (archi del grafo) e degli eventuali attributi (ad es. campi dei form input) legati ad essi

- lettura dei percorsi e creazione di un *TestMethod* per ogni percorso del grafo che unisce la radice (nodo Index) ad una foglia
- creazione di un identificativo univoco e di un nome per ciascun *TestMethod* che rappresenterà un test case della testsuite.

Le operazione di scrittura e di lettura di stati ed eventi, avvengono tramite i metodi: *XMLObject.objectToXML* e quello inverso *XMLObject.xmlToObject*, che effettuano operazione di write e read su file xml di oggetti *Map* (ad esempio*:*

Map<Long, StateVertex> memorizza tutti gli stati, nodi del grafo, accessibili tramite un id univoco tipo long*).* Invece, per quanto riguarda la scrittura dei *TestMehod* è una operazione più articolata e delicata perché questi devono diventare dei test cases e rispettare le regole di JUnit. Ecco perché questo compito viene affidato alla classe *JavaTestGenerator* con l'ausilio della libreria *VelocityContext* permette di creare uno streaming di dati che potrà essere scritto come un normale file. Per effettuare questa operazione è indispensabile la presenza un modello in formato "*.vm*" che sia stato precedentemente definito. La classe *JavaTestGenerator* istanzia una variabile *context* di tipo *VelocityContext* nella quale vengono inserite tutte le informazioni dinamiche in formato nome-valore, come mostrato in esempio:

```
context.put("date", new Date().toString());
context.put("classname", className);
context.put("url", url);
context.put("nameWebApp", PluginRunner.getNameWebApp());
context.put("plugins", pluginClassNames);
context.put("methodList", testMethods);
```

il context viene quindi passato al *VelocityEngine* che si occupa dell'operazione di merging, ossia della sostituzione delle variabili del template con i valori corrispondenti, di seguito è mostrato il metodo che effettua la scrittura finale:

```
public String generate(String outputFolder, String fileNameTemplate)
            Template template = engine.getTemplate(fileNameTemplate);
            Helper.directoryCheck(outputFolder);
            File f = new File(outputFolder + className + ".java");
            FileWriter writer = new FileWriter(f);
            template.merge(context, writer);
            writer.flush();
            writer.close();
      return f.getAbsolutePath();
```

A questo punto il file java contenente i test cases è pronto ad essere compilato ed eseguito, nel paragrafo che segue vengono descritte minuziosamente queste fasi.

4.3.2 Esecuzione delle testsuites

L'esecuzione dei files generati dinamicamente prevede anch'essa la collaborazioni di più classi (figura 20) tra cui spicca *TestSuiteHelper* che contiene i principali metodi a supporto di questa fase.

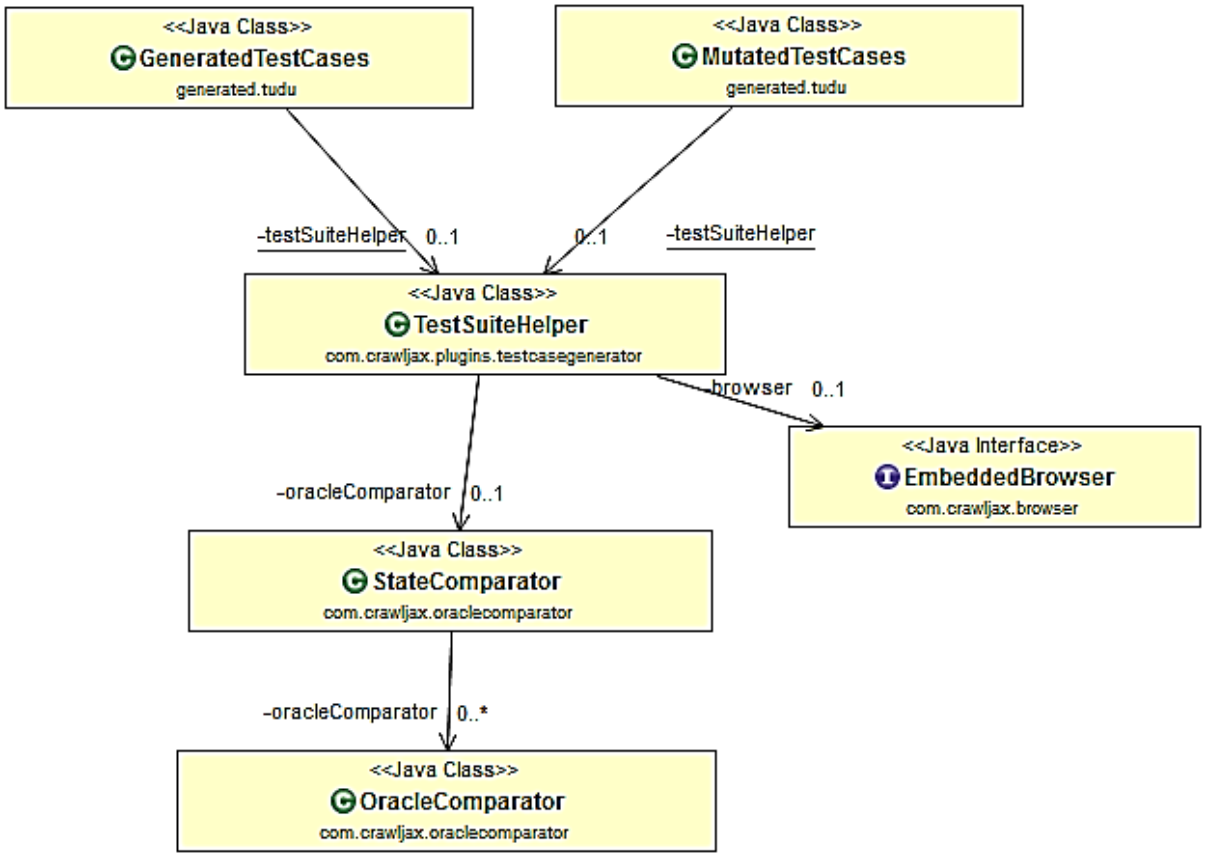


Figura 20

Di seguito vengono elencate solo le principali tra le numerose funzioni messe a disposizione dalla classe Helper:

- caricamento dell'applicazione nell'embedded browser, effettuato solo la prima volta
- inserimento dell'URL dello stato index, richiamato prima di eseguire ogni singolo test case

- lettura degli stati o gli eventi dai rispettivi files xml dati i loro id
- generazione dell'evento dato il suo id
- riempimento dei form relativi agli eventi a cui sono associati valori di input
- confronto tra stato corrente (DOM caricato all'interno del browser al momento del confronto) e stato atteso (DOM letto da file dato il suo id)
- controllo proprietà invarianti
- controllo presenza "alert" o "modal dialog" in seguito alla generazione di ogni evento
- controllo completo del *WebDriver (Selenium)* sul quale è stato costruito l'EmbeddedBrowser utilizzato durante l'esecuzione (riguardo a questa funzionalità è bene rimandare il lettore al paragrafo 4.4 per maggiori delucidazioni)

Di conseguenza, come anche chiaramente mostrato dal diagramma (in figura 19), data l'importanza delle funzionalità implementate, entrambe le classi *GeneratedTestCases* e *MutatetdTestCases* durante la loro esecuzione fanno riferimento alla stessa classe di supporto.

E' inoltre doveroso menzionare la tecnica di confronto tra stati utilizzata dalla classi *StateComparator* e *OracleComparator* implementate in Crawljax ed utilizzate non solo nella fase di esplorazione ma anche in questa di esecuzione con un duplice obiettivo:

- verificare il corretto comportamento dell'applicazione sotto test
- tentativo di scoprire nuovi stati in seguito alle mutazioni

In seguito alla generazione di un evento l'algoritmo confronta l'albero DOM con l'albero esistente esattamente prima che l'evento fosse generato (fase di esplorazione) o con quello atteso, letto da file (fase di testing). Per rilevare le eventuali differenze, *OracleComparator* calcola una distanza utilizzando il metodo Levenshtein (1996) [52], che prevede l'utilizzo di una soglia τ (0.0 – 1.0) al di sotto della quale i DOMs confrontati possono essere considerati cloni.

Inoltre durante l'esecuzione di una testsuite viene avviato JSCover in modalità proxy, cioè operando come server proxy http che instrumenta i file Javascript serviti attraverso di esso.

Al termine di ogni test viene richiamata la funzione *jscoverage_report* che genera una copertura in formato json e xml nel sottopercorso che ha lo stesso nome del test a cui fa riferimento. In questo modo si ottiene oltre che una copertura totale del codice JS anche un resoconto puntuale, di fondamentale importanza per le operazione di mutazione che si svolgeranno nella fase successiva. In figura è mostrato una generica copertura in formato xml, *cobertura-coverage.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE coverage SYSTEM "http://cobertura.sourceforge.net/xml/coverage-04.dtd">
<coverage branch-rate="0.06536420646406174" branches-covered="271" branches-valid="4146"
    complexity="0" line-rate="0.18067081604426002" lines-covered="1045" lines-valid="5784"
    timestamp="1422016196843" version="1.0.13">
  <sources>
    <source>C:/apache-tomcat-7.0.32/webapps/tudu-dwr</source>
  </sources>
  <packages>
    <package branch-rate="0.0472972972972973" complexity="0"
          line-rate="0.1359885181198421" name="/tudu-dwr/scripts/scriptaculous">
      <classes>
        <class branch-rate="0.09567901234567901" complexity="0"
                filename="tudu-dwr/scripts/scriptaculous/effects.js"
                    line-rate="0.244" name="/tudu-dwr/scripts/scriptaculous/effects.js">
          <methods/>
          <lines>
            <line branch="false" hits="1" number="14"/>
            <line branch="false" hits="0" number="15"/>
            <line branch="true" condition-coverage="0% (0/2)" hits="0" number="16">
              <conditions>
                <condition coverage="0%" number="1" type="jump"/>
              </conditions>
            </line>
            <line branch="false" hits="0" number="17"/>
            <line branch="true" condition-coverage="0% (0/2)" hits="0" number="18">
              <conditions>
                <condition coverage="0%" number="1" type="jump"/>
              </conditions>
            </line>
```

Figura 21

Le informazioni sulla copertura sono alquanto dettagliate, infatti, in ogni file xml sono presenti i seguenti nodi:

- *coverage:* rappresenta nodo root e contiene una descrizione generale in termini di lines-rate (rapporto tra il numero di linee coperte e numero di linee valide)
- *packages:* contiene una lista di nodi package che indicano il lines-rate per tutti i file js appartenenti allo stesso sottopercorso, per ognuno di essi c'è un nodo *classes*

- *classes:* contiene una lista di nodi *class* ognuno dei quali fa riferimento ad un unico file js con relativo lines-rate
- *lines:* contiene una lista di nodi *line* ciascuno inerente una singola linea del file js di riferimento e ha gli attributi *branch* (=true se è una linea di salto), *hits* (=1 se la linea è coperta) e *number* (id linea).

Un semplice parsing di questi files .xml da un resoconto sia totale che puntuale e rappresenta un ottimo punto di partenza per l'algoritmo genetico di cui al paragrafo seguente.

4.3.3 Mutazioni

Il package *testcasemutator* contiene il cuore dell'algoritmo genetico realizzato. Esso è composto dalle cinque classi che si interfacciano tra di loro e con il resto del sistema come mostrato in figura 22:

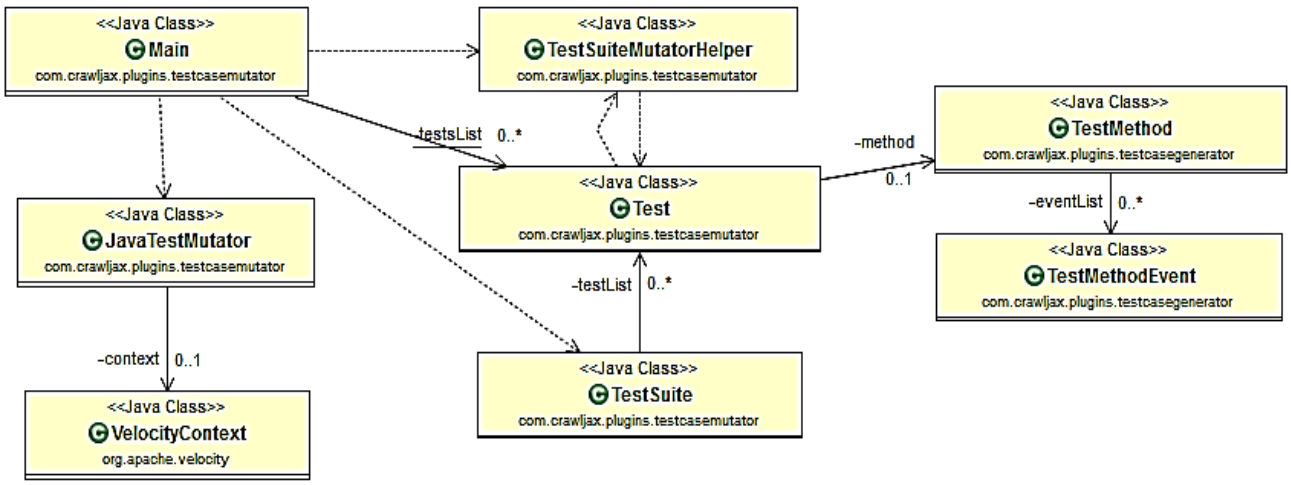


Figura 22

La classe *Main* è la classe di controllo in cui è implementata l'intelligenza dell'algoritmo genetico realizzato. Prima che questa venga avviata nessuna testsuite è stata ancora eseguita, quindi la prima operazione consiste nella compilazione ed esecuzione della test suite generata con l'ausilio di crawljax. Compilazione ed esecuzione vengono eseguite da due files batch appositamente realizzati, i cui comandi però cambiano dinamicamente con l'avanzare dell'algoritmo in modo da non sovrascrivere lo stesso file ma avere memoria di tutta la storia delle mutazioni.

Al termine dell'esecuzione, come spiegato nel paragrafo precedente, si hanno tutte le coperture locali quindi la classe *Main* legge uno ad uno tutti i file e per ognuno di essi crea un istanza della classe *Test* e li inserisce in una lista che sarà usata per costruire la *TestSuite.* Ogni istanza di *Test* contiene un *TestMethod* (descritto precedentemente) e altri parametri di valutazione quali:

- *lines-rate:* percentuali linee coperte
- *old-rate:* se un test viene mutato il *lines-rate* viene salvato in questo parametro e agli step successivi verrà ricalcolato il *lines-rate*
- *mapCoverage<Integer,List<Integer>>:* una mappa di copertura totale che associa ad un intero (*id* di un file js) una lista di valori dove :

(mapCoverage.get(i)).get(j)=1

se il test copre la j-esima riga del file i-esimo

- *diversity:* metrica di diversità fra i tests, descritta attentamente nel capitolo successivo
- *rank:* metrica di confronto, punteggio assegnato in base al numero e al "tipo" di righe coperte, anch'essa verrà discussa nel capitolo seguente
- *mutated:* semplice valore booleano, indica se un test è stato mutato o meno, il suo valore però viene resettato se la mutazione non ha portato a nessun cambiamento (miglioramento o peggioramento)
- *failed:* è un valore intero che se negativo indica che il test non è mai fallito altrimenti assume il valore positivo dell'indice del ciclo in cui è fallito

Tutte queste informazioni sono indispensabili alla fase di valutazione dell'algoritmo che ad ogni step decide quali test selezionare per sottoporli a eventuali crossover, mutazioni o eliminazione. I metodi per il calcolo delle metriche di fitness sono affidati alla classe *TestSuite* che viene costruita a partire da una lista di *Test* e ha informazioni globali, quali la copertura totale della test suite ad ogni ciclo dell'algoritmo e la matrice delle diversità (vedi capitolo 5). Infine la classe *TestSuiteMutatorHelper* contiene invece tutti i metodi per effettuare le operazioni di crossover sulle coppie di test scelte e le mutazioni sui

singoli test.

E' inoltre possibile personalizzare l'esecuzione dell'algoritmo con il settaggio dei parametri fondamentali:

- Percentuale di mutazione
- Percentuale di crossover
- Numero massimo di iterazioni
- Percentuale di copertura totale sufficiente
- Numero massimo di test cases della test suite finale (opzionale)

Le condizioni di terminazione dell'algoritmo sono tre, ed è sufficiente che se ne verifichi una affinché non vengano eseguite ulteriori iterazioni:

1. Numero massimo di cicli raggiunto
2. Operazioni di crossover e mutazione non ulteriormente applicabili
3. Efficienza desiderata raggiunta

Capitolo 5 – Casi di studio

In questo capitolo saranno presentati due casi di studio con lo scopo di verificare le prestazioni della tecnica genetica presentata nei capitoli precedenti. Le prestazioni saranno valutate in base all'abilità della tecnica genetica di generare test-suite più efficaci rispetto a quelle di partenza ricevute in input. I casi di studio coinvolgeranno il testing di 2 applicazioni web open-source reali. Essendo la tecnica genetica soggetta a variazioni aleatorie, per ogni applicazione sarà comparata la qualità delle test-suite generate tramite 5 esecuzioni differenti.

5.1 Research Questions

I casi di studio sono guidati dalla seguente research question:

- RQ1: La tecnica genetica genera test-suite più efficaci della tecnica di navigazione sistematica utilizzata come soluzione iniziale?

5.2 Variabili e metriche

Le variabili sono state divise nelle seguenti categorie:

1. *Variabili indipendenti*
2. *Variabili dipendenti*
3. *Variabili controllate*
4. *Variabili randomizzate*

Nei paragrafi seguenti verranno illustrate in modo dettagliato con particolare riferimento alle motivazioni che hanno portato alla scelta dei valori di tali parametri.

5.2.1 Variabili indipendenti

La variabile indipendente considerata in questi studi è la tecnica, che può assumere due valori: genetica o sistematica. La prima tecnica si ottiene fissando i parametri variabili, discussi al paragrafo 3.4.5, al fine di valutare la bontà della tecnica nel mondo reale.

I parametri presi in considerazione sono i seguenti:

- *Percentuale di turnover = 20%*. Indica la percentuale di popolazione che in ogni iterazione viene sostituita per lasciar posto a nuovi cromosomi generati tramite crossover e mutazioni. Il valore pari al 20% è stato scelto come compromesso in quanto con un valore eccessivamente alto di questo parametro c'è il rischio di muoversi troppo lungo lo spazio di ricerca senza però mai trovare gli ottimi. Al contrario, con un valore basso si rischia di rimanere bloccati più facilmente su un ottimo locale.
- *Percentuale di crossover = 80%*. All'operatore di crossover è stata destinata la percentuale maggiore di turnover in accordo con la tesi di Mitchell et al. [53] secondo la quale è dall'operatore di crossover che un algoritmo genetico è destinato a ricavare la maggior parte delle sue capacità.
- *Tecnica di crossover*. La tecnica di crossover scelta per l'esecuzione dell'algoritmo genetico prevede che i test generati tramite crossover non vadano a sostituire i loro genitori, bensì i test con più basso rank. Questa scelta è stata preferita per due motivi. Il primo risiede nel fatto che un crossover tra due cromosomi con importante valore genetico potrebbe generare due cromosomi figli con patrimonio genetico scarso, andando quindi a peggiorare la qualità della popolazione. Il secondo motivo è dettato dalla possibilità che un crossover tra due test con un alto rank potrebbe generare dei test non eseguibili con la conseguenza di aver perso un alto quantitativo genetico. Invece, andando a sostituire test con rank basso, la perdita di patrimonio genetico sarà nella maggior parte dei casi trascurabile o nulla.
- *Percentuale di mutazione = 20%*. All'operatore di mutazione è stata destinata una percentuale minore rispetto a quella di crossover, tuttavia il valore assegnato può essere considerato un buon compromesso tra un numero eccessivamente alto, che può

far perdere informazioni importanti sul contenuto genetico iniziale, ed un valore troppo piccolo tale da non garantire un valido ricambio generazionale.

- *Tecnica di mutazione*. La tecnica di mutazione, così come quella di crossover, prevede che i test mutati non vadano a sostituire i test mutanti, ma andranno invece a sostituire i test con più basso rank presenti all'interno della test-suite.
- *Iterazioni massime = 20*. Una delle condizioni di terminazione della tecnica genetica prevede che l'algoritmo effettui un numero di 20 iterazioni prima di fermarsi. Questo numero è stato scelto come compromesso tra un tempo di esecuzione totale accettabile e una buona probabilità che l'algoritmo riesca in questo numero di iterazioni a convergere verso l'ottimo.
- *Numero di impulsi di turnover massimi = non limitato*. La tecnica genetica può prevedere come altra condizione di terminazione il verificarsi di un numero precisato di occorrenze di impulsi di turnover. Gli impulsi sono utilizzati per cercare di smuovere l'algoritmo da un punto stazionario di ottimo locale verso un ottimo "migliore".
- *Frequenza di impulso = 3*. Il numero di iterazioni in cui la fitness globale deve essere costante prima di avere un impulso di turnover è stato impostato a 3. Un valore troppo basso di questo parametro rischia di eliminare troppo presto dalla test-suite i test generati tramite crossover che, pur non portando contributo immediato alla fitness globale della popolazione, se incrociati con un altro test potrebbero portare nuovo contributo genetico. Viceversa, un valore troppo alto rischia di lasciare fermo l'algoritmo per troppo tempo su un punto stazionario e allungare il tempo di esecuzione dello stesso prima di trovare l'ottimo.
- *Dimensione minima della test-suite = 50.* Questo parametro determina la dimensione minima, in termini di test, della test-suite. Questo valore è stato scelto in base ai precedenti valori dei parametri di percentuali di turnover, crossover e mutazione per garantire che, per ogni iterazione, sia possibile avere almeno un'occorrenza di crossover e mutazione. Infatti la dimensione minima della test-suite, per verificare che

ad esempio ci sia almeno un'occorrenza di mutazione degli input, deve rispettare la seguente condizione:

$$dim\_min \geq nMutazioniRichieste * \left(\frac{1}{\%mutazione_{input}}\right) * \left(\frac{1}{\%turnover}\right) = 2 * \left(\frac{100}{20}\right) * \left(\frac{100}{20}\right) = 50$$

Nel caso in cui la dimensione della test-suite iniziale moltiplicata per il fattore di crescita risulti minore della dimensione minima, alla test-suite saranno aggiunti ulteriori test generati tramite crossover e mutazione fino ad arrivare alla dimensione minima stabilita.

Ovviamente non è possibile affermare che i valori dei parametri scelti siano i migliori possibili e quindi non è escluso che una variazione di tali parametri non possa portare ad un miglioramento delle prestazioni della tecnica genetica. Per tale motivo all'utente finale, ovvero il tester, viene offerta la possibilità di personalizzare la configurazione iniziale di questi parametri.

La tabella 1 riporta un riassunto delle variabili discusse in questo paragrafo per la tecnica genetica.

La seconda tecnica invece si ottiene fissando i valori discussi al paragrafo 2.3.4:

- *Scheduling strategy:* Breadth First. Tramite questo parametro si specifica che il prossimo task ad essere eseguito sarà il task aggiunto meno recentemente alla task list.
- *Filtering strategy:* Questo parametro specifica quali tipologie di widget presenti su un'interfaccia sono considerati rilevanti. La tabella 2 schematizza quali widget sono stati scelti.
- *Exploration criterion:* Tramite questo parametro si stabilisce il criterio adottato secondo cui due stati saranno considerati equivalenti. Per la tecnica utilizzata, due interfacce saranno equivalenti se hanno lo stesso insieme di widget e relativi eventi.
- *History criterion:* History criterion è un predicato logico che stabilisce se la task list deve essere azzerata tra un'iterazione e l'altra.
- *Termination criterion:* l'algoritmo si ferma quando la task list è vuota.

Tabella 1 - Variabili indipendenti

Parametro	Valore
Percentuale di Turnover	20%
Percentuale di Crossover	80%
Tecnica di Crossover	Sostituzione test con basso rank
Tecnica di Mutazione	Sostituzione test con basso rank
Iterazioni massime	30
Numero impulsi di Turnover massimi	Non limitato
Frequenza di impulso	3
Fattore di crescita della popolazione iniziale	2.0
Dimensione minima della test-suite	50

Tabella 2 - Eventi considerati dalla tecnica di Filtering

Event	Clickable Elements
Click	Button, TextView, ImageView, LinearLayout, RelativeLayout, CheckBox, ToggleButton, NumberPicker, ListView, SingleChoiceList, MultiChoiceList, ListPreference, Spinner, RadioGroup, TabHost, SeekBar, RatingBar
Write Text	EditText, AutoCompleteTextView, SearchBar

5.2.2 Variabili dipendenti

La variabile dipendente dello studio è l'efficacia. L'efficacia è misurata in termini di LoC (Lines of Code) coperte durante le esecuzioni dei test case prodotti da una data tecnica. L'efficacia di una test-suite generata dalla tecnica genetica G è misurata come:

$$\eta(G) = \frac{LOC\ coperte\ da\ G}{Numero\ totale\ di\ LOC} * 100$$

5.2.3 Variabili controllate

Ci sono altre due variabili che sono in grado di influenzare le variabili dipendenti.

La prima variabile controllata è il numero di iterazioni. Per la tecnica genetica in esame, infatti, è stato previsto un numero di iterazioni prefissato prima della terminazione dell'algoritmo. Se così non fosse, difatti, ci sarebbe il rischio di confrontare due esecuzioni E1 e E2 della stessa applicazione A1 nelle condizioni in cui a E1 sono state concesse *x* iterazioni, mentre a E2 sono state concesse *y* iterazioni, con x>y. Anche se la qualità della test-suite generata da E1 fosse migliore, non si potrebbe affermare che ciò non sia causato dalle iterazioni in meno concesse a E2.

La seconda variabile riguarda il tempo che intercorre tra lo scatenamento di due eventi. Test eseguiti precedentemente sul tool di navigazione sistematica [54] hanno infatti dimostrato che un ritardo troppo piccolo tra due eventi scatenati può causare fallimenti nell'applicazione per il fatto che l'evoluzione dell'interfaccia dell'applicazione in risposta all'esecuzione di un evento potrebbe non essere stata completata prima che il prossimo evento sia scatenato. Si è deciso così di controllare tale variabile imponendo un valore costante pari a 3 secondi che dovrebbe essere sempre sufficiente ad evitare questi comportamenti dipendenti dal tempo.

5.2.4 Variabili randomizzate

Durante l'esecuzione della tecnica genetica, in diversi punti dell'algoritmo si ricorre

all'utilizzo di una variabile pseudo-casuale con distribuzione uniforme.

Crossover: durante la fase di crossover sono scelti tramite la variabile pseudo-casuale i test destinati ad incrociarsi tramite turnover. Inoltre anche il punto di taglio tra gli stati equivalenti (se presenti più di uno) è scelto in maniera pseudo-casuale.

Mutazione: durante la fase di mutazione sono scelti tramite variabile pseudo-casuale i test destinati ad essere mutati tra quelli che possiedono un input o una lista. Inoltre anche il valore di input è scelto in maniera pseudo-casuale da una lista di valori predefiniti.

5.2.5 Oggetti sperimentali

Lo studio si è concentrato sulla valutazione della tecnica genetica su 2 applicazioni web reali:

- *TuduList*: per la gestione di promemoria e note.
- *TaskFreak*: per la gestione condivisa di progetti.

Le applicazioni testate, simili per utilizzo, contano entrambe di 500 linee di codice javascript, e come tali, possono essere considerate un buon campione di valutazione della tecnica. Inoltre a ciascuna applicazione sono state imposte le stesse precondizioni iniziali.

5.3 Progetto sperimentale, Procedura e Materiale

L'architettura del tool comprende un unico componente principale, realizzato come un eseguibile Java sviluppato su una macchina Windows reale. L'eseguibile presenta due modalità di esecuzione:

1. *Generazione test-suite iniziale*
2. *Algoritmo genetico*

La prima, utilizzando le librerie messe a disposizione da Crawljax e JSCover, si occupa della generazione dei casi di test e del calcolo delle coperture puntuali ad essi associati. Al termine dell'esecuzione di questa modalità, il file .java, contenente l'intera testsuite, ed i files .xml contenenti le informazioni sugli stati e gli eventi vengono generati all'interno di un package il cui percorso è settabile in fase di configurazione. Inoltre il package è organizzato in sotto-packages in modo che tutte le informazioni (stati, eventi ed input)

relative a ciascun test siano facilmente reperibili durante l'esecuzione della seconda modalità. Infatti, tutto ciò che è stato generato in questa prima fase rappresenta l'input per la seconda modalità d'esecuzione del tool che si occupa dell'evoluzione dell'algoritmo genetico. Nel dettaglio, in questa fase vengono ciclicamente richiamati due files batch, rispettivamente per la compilazione e l'esecuzione dei files java contenenti le test-suite mutate. Ad ogni ciclo l'algoritmo calcola la fitness della popolazione e genera i nuovi test tramite le operazioni di crossover e mutazione per poi andare a creare la nuova test-suite sempre sotto forma di file .java che a sua volta dovrà essere compilato e rieseguito con l'ausilio dei batch files descritti precedentemente e così via fino a che una delle condizioni di terminazione, descritte nei capitoli precedenti, non è soddisfatta.

Inoltre, dato che il componente realizzato deve:

- effettuare il primo accesso alla web application sotto test,
- effettuare un login a seguito di ogni logout generato casualmente durante l'esplorazione o la ri-esecuzione dei test,
- salvare e ripristinare lo stato e le precondizioni a inizio di ogni nuova esplorazione e prima di ogni esecuzione di test,

sono stati definiti ad-hoc alcuni script in sql per permettere al tool di eseguire operazioni all'interno dei singoli database. L'esecuzione di questi script è possibile ovviamente solo a seguito dell'instaurazione della connessione con il database che viene realizzata con l'ausilio del driver *jdbc* [54].

Il progetto implementa quindi un'architettura software che è in grado sia di eseguire i test generati dal componente principale, sia di effettuare una navigazione sistematica dell'applicazione a partire da uno stato specificato. Quest'ultima operazione è utile nei casi in cui l'esecuzione di test generati tramite crossover o mutazione abbia portato alla scoperta di un nuovo stato rendendo così necessaria una fase di navigazione dell'applicazione a partire dalla nuova condizione.

5.3.1 Misurare la copertura – JSCover

Le metriche di copertura sono valutate automaticamente con l'ausilio delle librerie offerte da JSCover, capace di strumentare il codice Javascript dell'applicazione web sotto test, misurare la copertura di codice, generare report testuali e ipertestuali di cui verrà effettuato il parsing dal componente principale per calcolare le fitness locali e globali relative ad ogni test-suite.

JSCover è un software open source distribuito da GNU GPLv2 (GNU General Public License versione 2) [37], e rappresenta una implementazione avanzata dell'eccellente strumento JSCoverage [36].

Il tool lavora instrumentando il codice JavaScript prima che esso venga eseguito all'interno del browser e fornisce tre modi alternativi per farlo:

- *Modalità server*: è il metodo più semplice e consiste nell'usare un semplice web server che instrumenta il codice appena è servito al client
- *Modalità file-system*: in questa modalità i file.js vengono replicati nella loro versione instrumentata prima ancora di iniziare l'attività di testing
- *Modalità proxy*: simile alla modalità server ma in questo caso il server funge da proxy e quindi tutto il codice js che viaggia attraverso di esso, verso il client, viene intercettato e instrumentato. Questo tipo di modalità però supporta solo http e non HTTPS.

La modalità server (sia web che proxy) ha un duplice vantaggio:

3. È possibile memorizzare i report di copertura sul file system
4. È possibile includere codice JavaScript non caricato o non testato all'interno dei files report.

Lo strumento è in grado di misurare la copertura relativa a linee, salti ed intere funzioni attraverso il browser permettendo l'interazione con il DOM della web application.

Nell'esempio che segue, è mostrato chiaramente come JSCover instrumenta le linee di codice javascript:

```
alert('Hello World!');
```

diventerà:

```
//...header code above
if (! this._$jscoverage) {
   this._$jscoverage = {};
}
if (! _$jscoverage['test.js']) {
   _$jscoverage['test.js'] = {};
   _$jscoverage['test.js'].lineData = [];
   _$jscoverage['test.js'].lineData[1] = 0;
}
_$jscoverage['test.js'].lineData[1]++;
alert('Hello World!');
```

L'esempio mostra che quando un pezzo di codice è eseguito allora un contatore (***'_$jscoverage'***) relativo al file (***'test.js'***) e alla linea (***'1'*** in questo caso) viene incrementato. Il codice header in alto gestisce la dichiarazione della variabile ***'_$jscoverage'***. Procedure simili sono utilizzate nel caso di *branch* e *function.* La strumentazione del codice comporta un inevitabile rallentamento della normale esecuzione dell'applicazione sotto test, che quindi impiegherebbe almeno il doppio del tempo necessario all'esecuzione standard. Al momento il tool è in grado di strumentare solo file .js e non il codice presente all'interno degli elementi <script> dei file HTML. Un'altra annotazione importante è che questo strumento è in grado di instrumentare solo le linee "fisiche" dei file e quindi è in grado di lavorare al massimo uno statement per ogni linea. D'altra parte, più statement in una stessa linea potrebbero dare risultati inaspettati.

Nei casi di studi mostrati nei paragrafi successivi si è ovviamente tenuto conto di tutte le potenzialità e dei limiti dello strumento affinchè i risultati potessero essere quanto più veritieri possibili.

5.3.2 Procedura

La procedura utilizzata per eseguire il testing delle applicazioni tramite la tecnica proposta si articola quindi in due macro fasi (esplorazione e algoritmo genetico) a loro volte divise nelle varie fasi mostrate di seguito.

Fase 1: Configurazione. La prima fase consiste nell'installazione e configurazione in locale della web application da testare. Per tali operazioni, in entrambi i casi proposti

come esempio è stato utilizzato il tool XAMPP [55] che offre la possibilità di avviare un server Apache in locale sul quale eseguire le applicazioni e un dbms MySql per la gestione dei database ad esse associate. Quindi, in seguito l'avvio del server, è possibile eseguire la modalità "esplorazione" del tool. All'avvio vengono richiesti i seguenti parametri:

- *URL*: link corrispondente alla pagina iniziale dell'applicazione
- *AbsolutePath*: è il percorso contenente il tool che si sta eseguendo
- *TestAbsolutePath*: rappresenta la cartella in cui saranno generati gli output relativi ai singoli test cases organizzati in sotto cartelle
- *ClassesPath*: è la cartella dove vengono generati i files .java da compilare e eseguire
- *WebAppPath*: è il percorso contenente i sorgenti dell'applicazione sotto test
- *AllLibPath*: è il percorso relativo a tutte le librerie di cui necessitano i files batch per essere eseguiti, essendo essi esterni al tool.

Prima che l'esplorazione abbia inizio vengono automaticamente eseguiti dei controlli sulle cartelle di output affinché i test generati in una sessione non vengano confusi con quelli di un'altra precedentemente effettuata.

Fase 2: Prima Esplorazione. La seconda fase consiste nella prima esplorazione dell'applicazione che, sfruttando le librerie messe a disposizione da Crawljax, genererà la testsuite, in un file java, e le informazioni inerenti stati, eventi ed input ad essi relativi sotto forma di files xml all'interno di cartelle e sotto cartelle organizzate in una struttura gerarchica. Crawljax è stato configurato in modo da effettuare singole esplorazioni che prevedono la generazione di una sequenza di al massimo dieci eventi alla volta, al termine delle quali le precondizioni dell'applicazione vengono ripristinate. Crawljax termina l'esplorazione quando o non è più in grado di generare nuovi eventi significativi o al raggiungimento del numero massimo di stati precedentemente prefissato (per gli esempi presi in considerazione è stato dimostrato che 90 stati visitati è un buon compromesso).

Fase 3: Configurazione Algoritmo Genetico. Da questo momento in poi parte la seconda macro fase, ovvero l'algoritmo genetico vero e proprio. All'utente tester viene offerta la possibilità di inserire un certo numero di parametri di configurazione quali:

- *NumMaxCycles*: numero massimo di cicli che si desidera eseguire, tra 10 e 100, di default è 10
- *MutationRate*: percentuale di mutazione iniziale che verrà applicata al numero totale di cromosomi presente all'inizio dell'algoritmo e che potrà cambiare automaticamente nel corso dell'evoluzione dell'algoritmo, di default è 20%.
- *CrossoverRate*: percentuale di crossover iniziale che verrà applicata al numero totale di cromosomi presente all'inizio dell'algoritmo e che potrà cambiare automaticamente nel corso dell'evoluzione dell'algoritmo, di default è 20%.
- *NumTestCases* (opzionale): numero totale di test cases che si desidera avere nella test-suite finale, il suo valore dovrà essere al meno pari alla metà dei test presenti all'inizio e al massimo il doppio.

Gli altri parametri quali frequenza di impulsi, percentuale di variazioni dei tassi di mutazione e crossover non sono configurabili bensì preimpostati.
Prima che l'algoritmo inizi la sua evoluzione vengono effettuati controlli per verificare sia l'effettiva presenza dei files batch all'interno del path predefinito sia la presenza del file java relativo alla testsuite e dei files xml relativi gli stati, gli eventi e gli input.

Fase 4: Prima copertura. Se tutti i controlli appena descritti vanno a buon fine allora il file inerente la test-suite precedentemente generata viene compilato e solo nel caso in cui anche la compilazione va a buon fine, si avvia la prima esecuzione il cui obiettivo è il calcolo delle coperture iniziali. Quest'ultime vengono calcolate con JSCover che mette a disposizione delle apposite librerie per la gestione dei file di report. Per ogni test eseguito verrà creato un file xml contenente tutte le informazioni che saranno utilizzate nelle fasi successive dall'algoritmo genetico.

Fase 5: Creazione della Test List. In questa fase, l'ultima antecedente l'evoluzione dell'algoritmo genetico vero e proprio, viene creata una lista di test (test-suite di partenza) ognuno dei quali al suo interno contiene le seguenti informazioni:

- *Sequenze ordinate di stati ed eventi* con relativi input e se il test sarà mutato in futuro verrà tenuto memoria anche dell'ultima sequenza precedente la mutazione
- *Valori di copertura*: numero linee coperte, rate in base al numero totali di linee valide, mappa di copertura e se il test sarà mutato verrà tenuto memoria anche di tutti i valori di copertura antecedenti la mutazione
- *Diversità* e *rank*: metriche discusse ampiamente in precedenza
- *isFailed, isMutated, isImproved*: viene tenuto traccia se il test è stato mutato (al ciclo precedente), o se è fallito (id del ciclo in cui è fallito), o se ha subito miglioramenti (ossia se i suoi valori di copertura sono migliorati a seguito di mutazioni).

Fase 6: Iterazione 0. L'algoritmo prende in input i file precedentemente generati ed opera un'*ottimizzazione* della test-suite andando ad eliminare tutti i test la cui copertura è assicurata interamente da almeno un altro test, portando così la test-suite ad una dimensione di *x* test. Così facendo la test-suite contiene solo test con patrimonio genetico rilevante. La dimensione della test-suite viene poi aumentata allo scopo di ospitare i nuovi cromosomi derivanti dalle operazioni di crossover e mutazione tra gli abitanti della popolazione senza dover eliminare forzatamente qualcuno degli *x* test importanti. Definito come *fc* il fattore di crescita della test-suite e *dim_min* la dimensione minima che possiamo accettare per una test-suite, la popolazione allora crescerà fino al valore *y*, calcolato come:

$$y = \max[(x * fc), \dim_{min}\ ]$$

Gli *(y-x)* test che saranno aggiunti saranno distribuiti tra crossover, mutazione input e mutazione liste a seconda dei parametri imposti per la tecnica.

Fase 6: iterazioni>0. Per tutte le iterazioni diverse dall'iterazione 0, l'algoritmo genetico prevede di prendere in input i test generati dall'iterazione precedente e generare dei report ipertestuali dei file di copertura che saranno analizzati per calcolare la fitness locale dei test.

Ogni test generato andrà a sostituire il test con fitness locale più bassa della popolazione. Il numero di test generati in questo passo sarà dipendente dal valore di turnover impostato.

Quando, dopo un numero definito di iterazioni (per la tecnica genetica in esame sono 3), la fitness globale della test-suite rimane costante, per cercare di smuovere l'algoritmo da un punto di ottimo stazionario avviene una nuova fase di ottimizzazione della test-suite, in cui vengono eliminati tutti i test la cui copertura è inclusa in almeno un altro test. Tutti i test eliminati saranno poi rimpiazzati da nuovi test generati tramite crossover e mutazione (secondo le proporzioni dettate dai parametri impostati) tra i test sopravvissuti all'interno della popolazione.

Fase 7: Terminazione dell'algoritmo. L'algoritmo termina quando:

- l'algoritmo ha eseguito tutte le iterazioni previste (*indexCycle=numMaxCycles*)
- l'algoritmo ha raggiunto la soglia minima di *linesRate* desiderata
- non è possibile più effettuare operazioni di crossing e mutazione

Quest'ultima condizione si verifica qualora l'algoritmo non è in grado né di trovare nuovi tests compatibili per il crossover né test mutabili, ad esempio perché non ci sono test con input associati o tutti i test sono stati già mutati con successo. In quest'ultimo caso sarebbe controproducente agire su test case che hanno già portato a dei miglioramenti per le mutazioni subiti ai cicli precedenti.

Gli esperimenti sono stati effettuati su un PC con sistema operativo Windows 7 installato, un processore 64 bit , frequenza di clock di 3GHz e 4 GBytes di RAM. Le versioni di XAMPP installate sono due: 1.6.8 e 3.2.1. La versione di Selenium utilizzata è la 2.43.1, la versione di Crawljax è la 2.3.3 mentre quella di JSCover la 1.0.13.

5.4 Metodo di analisi

L'analisi dei risultati della sperimentazione e la loro discussione è stata divisa in due parti distinte. Nella prima parte saranno mostrati i risultati ottenuti dalle sei esecuzioni diverse per ogni applicazione testata, raffrontando l'efficacia delle test-suite ottenute con la tecnica genetica rispetto alla tecnica di navigazione sistematica. Saranno riportate inoltre statistiche riguardanti gli impulsi di turnover occorsi.

Nella seconda parte, invece, sarà evidenziato l'andamento dell'efficacia delle sei esecuzioni per ogni applicazione. Sarà inoltre evidenziato in che modo la tecnica genetica è riuscita ad ottenere un incremento di efficacia rispetto alla tecnica di navigazione sistematica.

5.5 Risultati

Per rispondere alla Research Question RQ1, sono state testate 2 applicazioni di cui si è calcolata l'efficienza della test-suite generata dalla tecnica genetica. I risultati ottenuti sono stati comparati con quelli della navigazione sistematica effettuata con Crawljax con lo scopo di valutare l'effettivo miglioramento di qualità della test-suite mutata.

Inoltre per ciascuna delle due applicazioni testate, oltre al confronto con la navigazione sistematica, è stato introdotto un ulteriore termine di paragone. Infatti è stata effettuata una scrupolosa esplorazione "manuale" che fosse in grado di generare tutti i possibili eventi o almeno sull'interfaccia utente dell'applicazione da testare. Questo tipo di crawling è stato effettuato con l'ausilio esclusivo dello strumento JSCover e ha avuto come obiettivo quello di mostrare quanto codice fosse effettivamente testabile durante l'attività di testing rispetto a quella parte di codice definito in gergo tecnico "fantasma". Questa espressione sta ad indicare quelle linee di codice presenti all'interno dell'applicazione ma che effettivamente non possono essere mai eseguite o perché fanno parte di librerie che vengono importate ma che non vengono mai utilizzate o semplicemente perché la logica del flusso di controllo è errata e quindi non esiste alcuna condizione tale da sollecitare determinati blocchi del software.

5.5.1 *TuduList*

La prima applicazione testata è stata TuduList [56], una web application per la gestione di progetti. L'applicazione offre tutte i servizi utili alla gestione dei files "todo": accesso, modifica ed anche condivisione sul web. Tudulist è opensource ed è stata installata in locale con l'ausilio di XAMPP tramite il quale è stato possibile creare localmente l'interfacciamento tra la web app e il proprio database, presenza indispensabile per la maggior parte delle applicazioni web. Tudulist presenta ben 15 files di tipo js (javascript) e molti di questi non sono altro che intere librerie delle quali al massimo viene utilizzato un 20% delle totali funzioni in esse contenute (ad esempio nel caso di *prototype.js* solo il 18% è effettivamente eseguito).

I risultati ottenuti tramite JSCover sono riassunti nella tabella seguente:

Tabella 3 – Linee di codice totali, Linee di codice coperte dalla Navigazione Manuale, Linee di codice coperte dalla Navigazione Sistematica (Crawljax), Numero di esecuzione e Linee di codice coperte dalla tecnica genetica per l'applicazione TuduList

LOCs Totali	LOCs Manuali	LOCs Crawljax	EX	LOCs Genetic	Iterazioni	Impulsi	Dim. Test Suite iniziale
5784	1157	972 (15% totali) (84% manual)	E1	1032 (18%totali) (89%manual)	30	3	50
			E2	1041 (18%totali) (88%manual)	30	4	50
			E3	1027 (18%totali) (90%manual)	30	4	50
			E4	1041 (18%totali) (90%manual)	30	3	50
			E5	1011 (18%totali) (86%manual)	30	4	50
			E6	1032 (18%totali) (89%manual)	30	4	50

I risultati ottenuti sono abbastanza simili e distano tra loro un massimo di 30 LOCs. La crescita di copertura tra la tecnica di navigazione sistematica e la tecnica genetica è piccola ma importante. La test-suite per quest'applicazione si è dimostrata più efficace rispetto a quella generata tramite navigazione sistematica.

5.5.2 *TaskFreak*

TaskFreak [57], la seconda applicazione testata, rappresenta un web-based task manager e todo list. Permette l'organizzazione degli obiettivi, delle priorità e delle deadline di progetti offrendo la possibilità di dividere le attività per contesto e condividere i progressi ottenuti da i vari utenti che collaborano su di un medesimo progetto. Anche TaskFreak è stato installato in locale con l'ausilio di XAMPP, e presenta 5 files.js per un totale di 555 linee di codice valide.

I risultati ottenuti sono riassunti in tabella seguente.

Tabella 4 - Linee di codice totali, Linee di codice coperte dalla Navigazione Manuale, Linee di codice coperte dalla Navigazione Sistematica (Crawljax), Numero di esecuzione e Linee di codice coperte dalla tecnica genetica per l'applicazione TaskFreak

LOCs Total	LOCs Manual	LOCs Crawljax	EX	LOCs Genetic	Iterazioni	Impulsi	Dim. Test Suite iniziale
555	291	257 (46% total) (88% manual)	E1	278 (50%totali) (95% manual)	30	3	50
			E2	262 (47%totali) (90% manual)	30	4	50
			E3	278 (50%totali) (95% manual	30	3	50
			E4	278 (50%totali) (95%manual)	30	3	50
			E5	278 (48%totali) (95%manual)	30	4	50
			E6	262 (47%totali) (90%manual)	30	4	50

A differenza di TuduList, i risultati ottenuti tramite il testing di TaskFreak si sono dimostrati meno discordanti ma più efficaci. Le esecuzioni E1, E3, E4 e E5 hanno infatti ottenuto un importante aumento del 8% rispetto all'efficacia ottenuta dal solo Crawljax, considerando come termine di paragone la copertura raggiunta attraverso l'e

5.6 Discussioni

In questo paragrafo saranno discussi le motivazioni che hanno portato la tecnica genetica ad essere più efficace di quella sistematica e si faranno considerazioni sullo sforzo necessario per ottenere questo miglioramento.

5.6.1 *TuduList*

Come visto nel paragrafo precedente, la tecnica genetica si dimostra efficace con l'applicazione TuduList. Il più importante incremento di efficacia presente in ogni esecuzione è dovuto alla capacità della tecnica genetica di impostare, tramite continue mutazioni di input varie combinazione di valori da inserire nei textbox. Questo ha permesso alla tecnica di scoprire un nuovo stato e di navigare una parte dell'applicazione prima "sconosciuta". D'altra, il tool di navigazione sistematica permette solo l'inserimento di stringhe alfanumeriche i cui caratteri sono generati in modo random e pertanto la fase di esplorazione sistematica dell'applicazione si muove sempre nel medesimo stato relativo ad uno stesso errore di inserimento non corretto. A tal proposito si prenda come esempio la figura 22, che illustra la pagina web relativa all'inserimento di un nuovo "TUDU". L'applicazione richiede all'utente la compilazione di un form dotato di cinque input al termine della quale si può optare tra una "SUBMIT" e "CANCEL". La generazione dell'evento CANCEL porta l'applicazione a spostarsi sempre nello stesso stato sia nel caso in cui dati di input inseriti sono corretti che non. L'evento SUBMIT invece può portare l'applicazione in due stati differenti: "SUCCESS" se tutti gli input sono validi o "WRONG INSERT" in caso contrario. In fase d'esplorazione sistematica viene visitato sempre e solo il secondo stato mentre il caso genetico, attraverso le continue mutazioni,

riesce a trovare la combinazione valida tale da portare l'applicativo nello stato mai esplorato, sollecitando allo stesso tempo nuove linee di codice, ancora non eseguite.

Come si può notare quindi i miglioramenti di efficacia per quest'applicazione sono dovuti

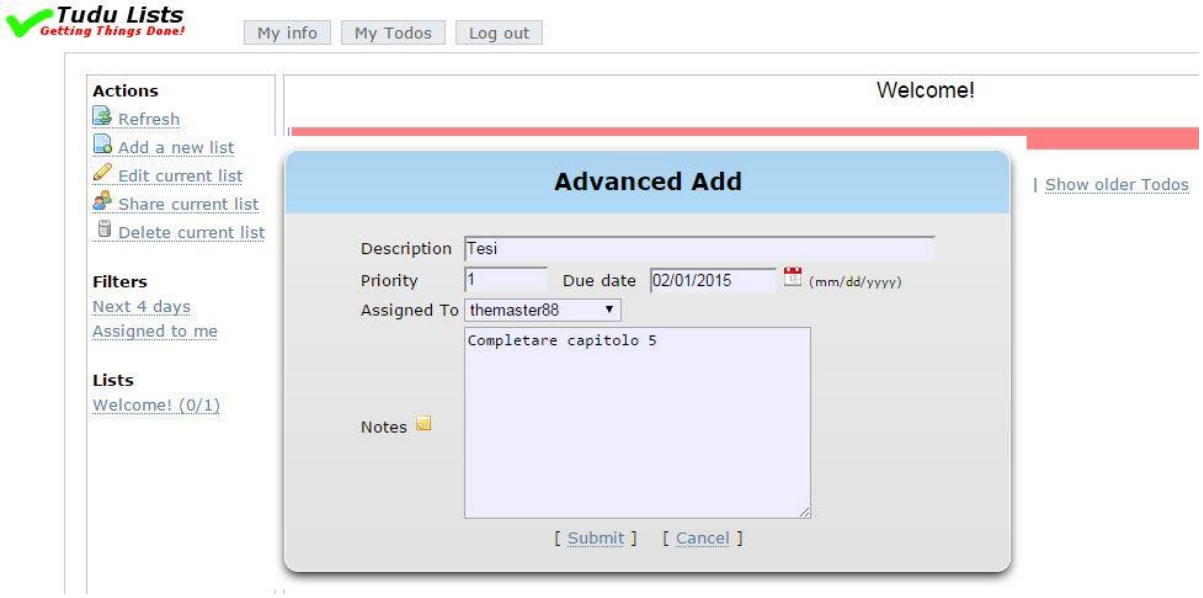


Figura 23

principalmente all'operatore di mutazione.

I risultati ottenuti sono riassunti nel grafico seguente:

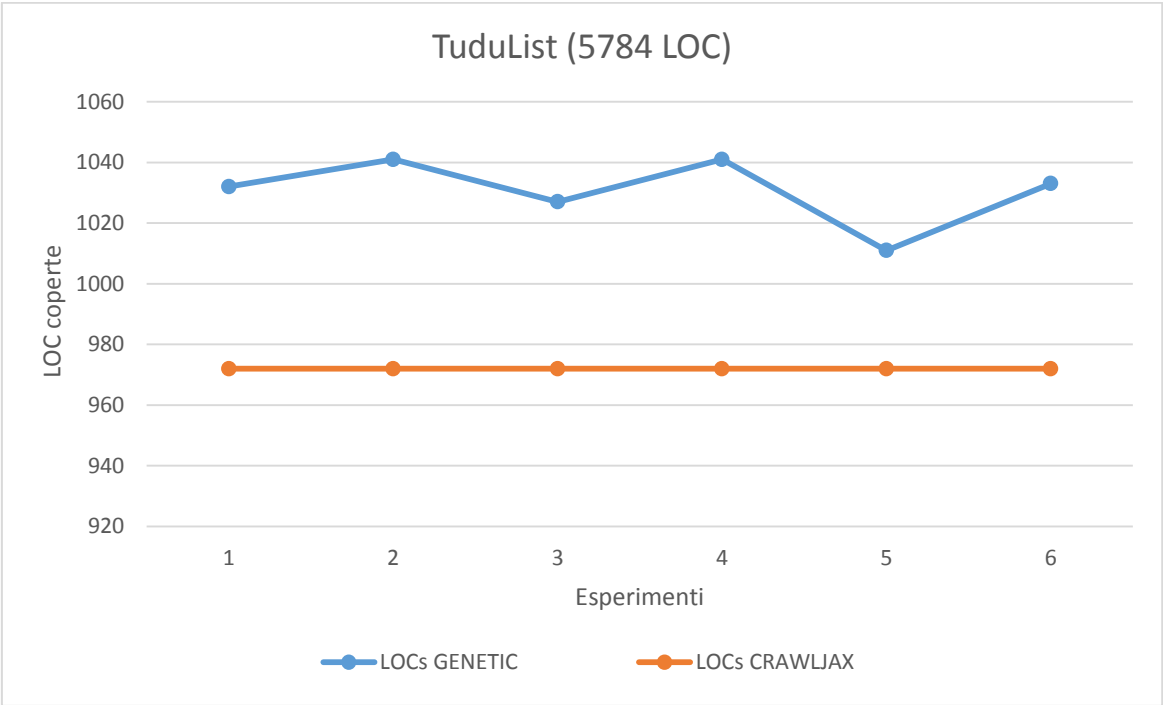


Figura 24

5.6.2 *TaskFreak*

Come accennato nel paragrafo precedente, la tecnica genetica ha mostrato risultati leggermente più discordanti nel testing di quest'applicazione. Anche in questo caso i risultati migliori sono scaturiti dall'operatore di mutazione che ha permesso di esplorare due nuovi stati. Il primo relativo all'inserimento di un nuovo progetto (vedi figura 25) ed il secondo relativo alla creazione di un nuovo utente (vedi figura 26).

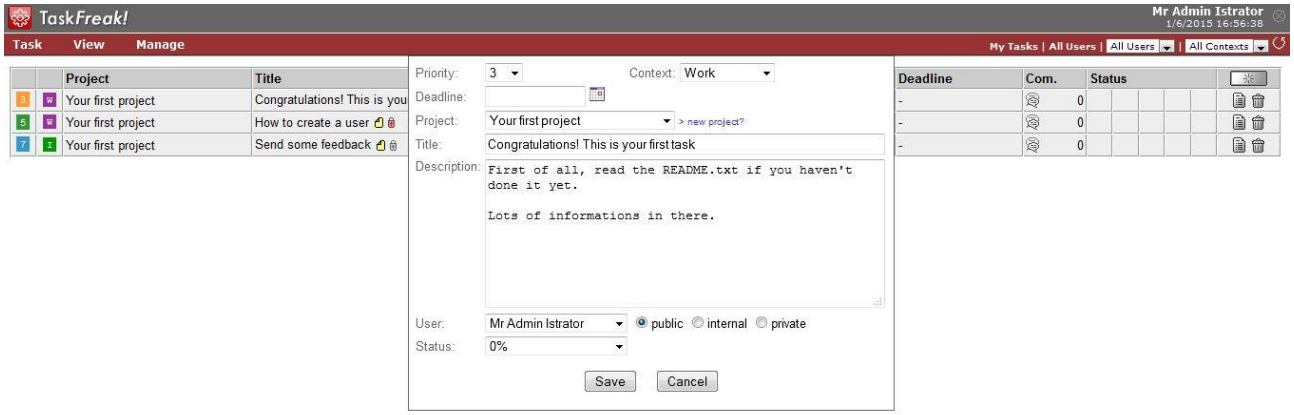


Figura 25

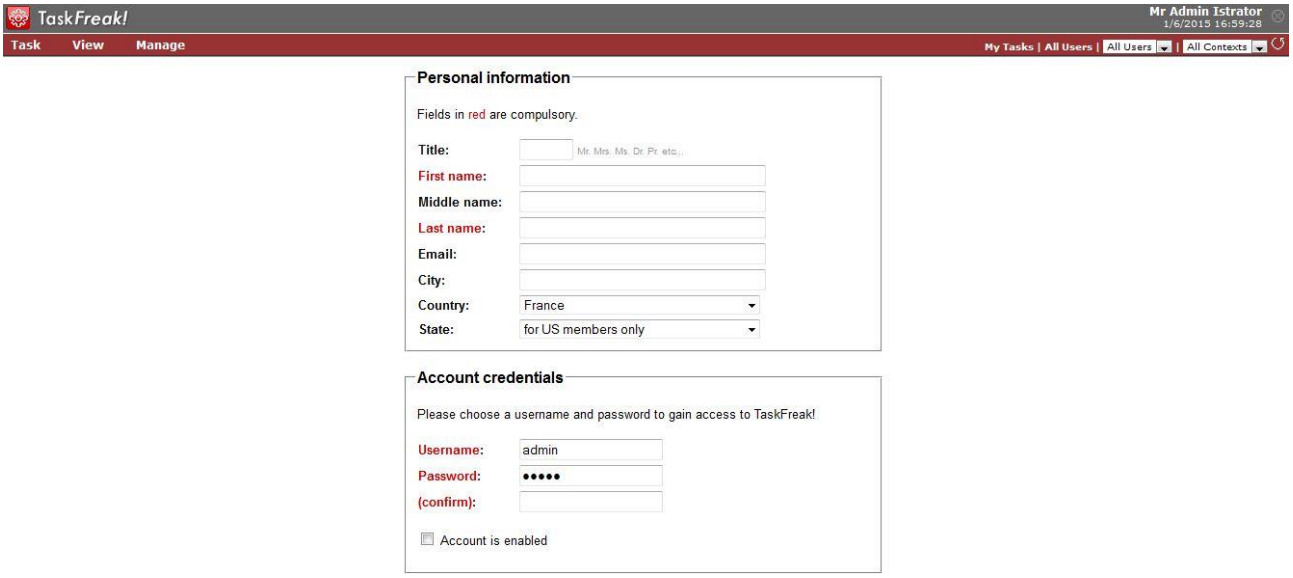


Figura 26

Di seguito è mostrato il grafico dell'andamento dell'efficienza al variare del numero delle iterazioni per le esecuzioni effettuate.

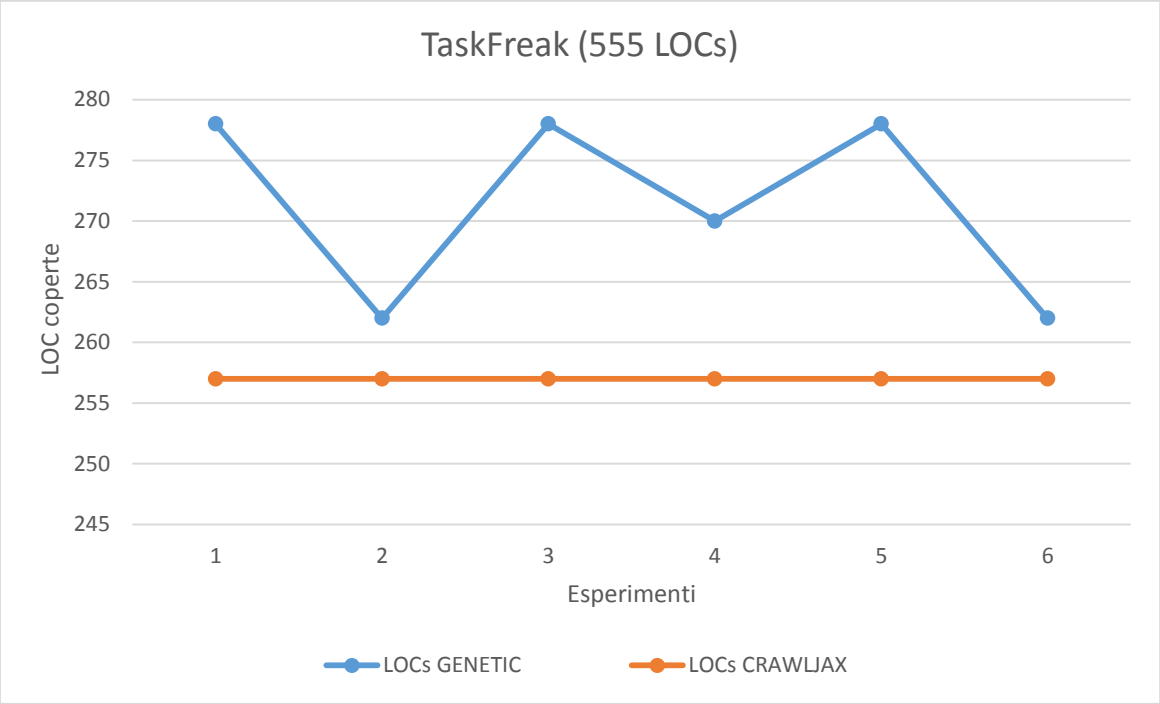


Figura 26

5.6.6 Unione delle esecuzioni

Per saperne di più circa il comportamento delle esecuzioni si è deciso, per ogni applicazione, di unire le test-suite migliori generate da ogni esecuzione per scoprire se la loro unione crea una test-suite ancora più efficace. Ciò è utile per conoscere se ogni esecuzione prende strade di ricerca diverse per cercare di convergere poi verso lo stesso ottimo globale o se gli ottimi trovati dalle diverse esecuzioni sono tanti ottimi globali differenti. In tabella sono riportati i risultati di tale esperimento.

Tabella - Confronto tra la migliore Test-Suite e l'unione delle migliori Test-Suite

Applicazione	Copertura migliore ottenuta dall'algoritmo genetico	Unione delle coperture delle esecuzioni
TuduList	1041	1041
TaskFreak	279	279

È possibile notare che la mancanza di incrementi possa essere considerata significativa per affermare che le esecuzioni non vanno verso punti di ottimo diversi. Infatti, come già fatto notare durante la discussione, il numero massimo di 30 iterazioni può essere considerato limitativo per alcune applicazioni più complesse e quindi più difficili da testar, tanto che aumentando tale numero si può pensare che le esecuzioni possano convergere verso il valore ottenuto dall'unione delle esecuzioni.

5.7 L'efficienza

Per completezza di informazione, inoltre, sono riportati ulteriori dettagli circa le esecuzioni migliori della tecnica genetica per verificare l'efficienza delle test-suite generate.

Nella tabella seguente sono mostrati:

- *Tempo di esecuzione totale*
- *Numero di eventi generati da Crawljax*
- *Numero di eventi generati dalla tecnica genetica*
- *Numero di nuovi stati attraversati*

Tabella 5 - Tempo di esecuzione dell'algoritmo, eventi scatenati da Crawljax, eventi scatenati dalla tecnica genetica, nuovi stati scoperti dalla tecnica genetica e test totali eseguiti

Applicazioni	Tempo esecuzione algoritmo (in ore)	Eventi scatenati da Crawljax	Eventi scatenati dalla tecnica genetica	Nuovi stati scoperti dalla tecnica genetica	Numero di Test differenti eseguiti
TuduList	7h 30m (~ 15m a iteraz.)	55	1110	1	~ 500
TaskFreak	6h (~ 12m a iteraz.)	35	1950	2	~ 500

Nella tabella sono riportate le stime per l'esecuzione dell'intero algoritmo per le applicazioni testate. Si ricorda, però, che in casi di test-regression non c'è bisogno di rieseguire l'intera tecnica genetica ma basta rieseguire solo la test-suite migliore generata dalla tecnica per abbattere i tempi di esecuzione.

I tempi riportati sono solo delle stime in quanto ovviamente dipendenti da diversi fattori, quali soprattutto l'hardware della macchina. In tabella sono, inoltre, riportati gli eventi scatenati durante tutta l'esecuzione della tecnica. Il numero di eventi scatenati dalla tecnica, così come il tempo di esecuzione, sono dipendenti dalla complessità dell'applicazione. Sono riportati anche il numero di stati scoperti dalla tecnica genetica e l'eventuale numero di eventi scatenati da Crawljax dopo la scoperta del nuovo stato. Infine, è riportato il numero di test eseguiti durante l'intera esecuzione.

5.8 Minacce alla validità

In questo paragrafo sono analizzate e discusse le minacce alla validità relative alle conclusioni risultanti dai casi di studio.

5.8.1 Minacce alla validità esterne

Nel corso dei casi di studio effettuati sono state analizzate 2 applicazioni web di differenti tipi e dimensioni che possono essere considerate un piccolissimo campione del mondo reale di applicazioni. Ovviamente non si esclude che risultati differenti si possono ottenere con un campione più largo di applicazioni. Dal momento che il tool non richiede l'intervento umano lungo tutta la durata della sua esecuzione, si possono escludere minacce derivanti da errori umani.

5.8.2 Minacce alla validità interne

Per evitare minacce interne alla validità, ogni esperimento appartenente alla medesima applicazione è stato eseguito con le stesse precondizioni. Altre minacce minori alla validità sono rappresentate dalla stato del sistema (come il valore della carica della batteria) o la data e l'ora di sistema.

Conclusioni

In questo lavoro si è discusso del problema del testing per le applicazioni web moderne e si è proposta una soluzione tramite un algoritmo genetico automatizzato che, utilizzando una tecnica di navigazione sistematica come punto di partenza, riuscisse a migliorare l'efficacia della test suite iniziale.

Dopo le sperimentazioni effettuate, è possibile concludere che l'algoritmo genetico ha concretamente permesso di creare test-suite più efficaci di una tecnica di navigazione sistematica.

Sviluppi Futuri

Si è visto durante la trattazione della tesi come l'algoritmo genetico proposto disponga di diversi parametri che possono modificare le prestazioni dell'algoritmo. Uno sviluppo futuro riguarderà innanzitutto la modifica di questi o l'eventuale aggiunta di altri.

Durante questo lavoro, infatti, non ci si è posto l'obiettivo di trovare la configurazione ottimale dell'algoritmo che può dunque ancora essere migliorato tarando opportunamente i valori associati ai parametri o, laddove è possibile, modificando le funzioni per il calcolo di questi. Ad esempio, in riferimento alle applicazioni testate, si è visto come si è avuto un numero di impulsi di turnover molto elevato che può portare a pensare che la percentuale di turnover utilizzata per quell'applicazione sia stata troppo bassa.

Un ulteriore sviluppo futuro potrebbe consistere nel migliorare i tempi di esecuzione dell'algoritmo parallelizzando l'esecuzione dei test su più macchine diverse.

Bibliografia

[1] A. Bagnall, V. Rayward-Smith, and I. Whittley. The next release problem. Information and Software Technology, 43(14):883–890, Dec. 2001.

[2] J. Aguilar-Ruiz, I. Ramos, J. C. Riquelme, and M.Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43(14):875-882, Dec. 2001.

[3] E. Alba and J. F. Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers and Operations Research (COR) focused issue on Search Based Engineering*.

[4] *Learning Test-Driven Development by Counting Lines*; Bas Vodde & Lasse Koskela; IEEE Software Vol. 24, Issue 3, 2007

G. Antoniol, M. Di Penta, and M. Harman. A robust search–based approach to project management in the presence of abandonment, rework, error and uncertainty. In 10*th International Software Metrics Symposium (METRICS 2004)*, pages 172–183, Los Alamitos, California, USA, Sept. 2004. IEEE Computer Society Press.

[5] A. Baresel, D. W. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, Omni Parker House Hotel, Boston, Massachusetts, July 2004. Appears in Software Engineering Notes, Volume 29, Number 4.

[6] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation*

Conference, pages 1337–1342, New York, 9-13 July 2002. Morgan Kaufmann Publishers

[7] L. C. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *SEKE*, pages 43–50, 2002.

[8] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1021–1028. ACM, 2005.

[9] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 2, pages 1885–1892, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[10] M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers.

[11] Harman, Mark. "The current state and future of search based software engineering." *2007 Future of Software Engineering*. IEEE Computer Society, 2007.

[12] G. Bruno. Gli Algoritmi Genetici - *http://www.federica.unina.it/ingegneria/ricerca-operativa-ing-2/algoritmi-genetici/*

[13] Holland, John H. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. U Michigan Press, 1975.

[14] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings — Software*, 150(3):161– 175, 2003.

[15] Goldberg, David E., and Kalyanmoy Deb. "A comparative analysis of selection schemes used in genetic algorithms." *Urbana* 51 (1991): 61801-2996.

[16] Selection of the fittest: *http://www.edc.ncl.ac.uk/highlight/rhjanuary2007g02.php/*

[17] Whitley, L. Darrell. "The GENITOR Algorithm and Selection Pressure: Why Rank-

Based Allocation of Reproductive Trials is Best." *ICGA*. Vol. 89. 1989.

[18] Goldberg, D. E. "Genetic Algorithms in Search, Optimization and Machine Learning".

[19] Grefenstette, John J., and James E. Baker. "How genetic algorithms work: A critical look at implicit parallelism." *Proceedings of the third international conference on Genetic algorithms*. Morgan Kaufmann Publishers Inc., 1989.

[20] Brindle, Anne. "Genetic algorithms for function optimization." (1981).

[21] Algoritmo genetico. (21 luglio 2014). *Wikipedia, L'enciclopedia libera*. Tratto da: http://it.wikipedia.org/wiki/Algoritmo_genetico

[22] Jones, Bryan F., H-H. Sthamer, and David E. Eyres. "Automatic structural testing using genetic algorithms." *Software Engineering Journal* 11.5 (1996): 299-306.

[23] Baresel, André, Harmen Sthamer, and Michael Schmidt. "Fitness Function Design To Improve Evolutionary Structural Testing." *GECCO*. Vol. 2. 2002.

[24] Bottaci, Leonardo. "Instrumenting Programs With Flag Variables For Test Data Search By Genetic Algorithms." *GECCO*. Vol. 2. 2002.

[25] Briand, Lionel C., Yvan Labiche, and Marwa Shousha. "Stress testing real-time systems with genetic algorithms." *Proceedings of the 2005 conference on Genetic and evolutionary computation*. ACM, 2005.

[26] Srivastava, Praveen Ranjan, and Tai-hoon Kim. "Application of genetic algorithm in software testing." *International Journal of software Engineering and its Applications* 3.4 (2009): 87-96.

[27] Michael, Christoph C., Gary McGraw, and Michael A. Schatz. "Generating software test data by evolution." *Software Engineering, IEEE Transactions on*27.12 (2001): 1085-1110.

[28] Ali Mesbah, Arie van Deursen, and Stefan Lenselink (2012). *"Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes."* ACM Transactions on the Web (TWEB)

[29] *"Selenium WebDriver"*. Simon Stewart, "*The Architecture of Open Source*

Applications" Retrieved June 29, 2014.

[30] *ACM Transactions on the Web*, A. Mesbah , March 2012.

[31] Kent Beck, Test-Driven Development: By Example, Addison-Wesley Professional, 2002, ISBN 0-321-14653-0.

[32] Utting, Mark, and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[33] Hamlet, Dick. "When Only Random Testing Will Do." (2006).

[34] Hamlet, Richard. "Random testing." *Encyclopedia of software Engineering*(1994).

[35] Musa, John D. "Operational profiles in software-reliability engineering."*Software, IEEE* 10.2 (1993): 14-32.

[36] JSCoverage: Code Coverage for JavaScript, *http://siliconforks.com/jscoverage*

[37] GNU General Public License, version 2, http://www.gnu.org/licenses/gpl-2.0.html

[38] E. Dustin, J. Rashka, and J. Paul - *Automated software testing: introduction, management, and performance*. Boston: Addison-Wesley, 1999.

[39] Myers, Glenford J., Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[40] John Ferguson Smart, *Jenkins: the definitive guide*, Sebastopol, O'Reilly, 2011.

[41] Tramontana P., "Testing: definizioni",
http://www.federica.unina.it/ingegneria/ingegneria-del-software-ingegneria/testing-definizioni/

[42] Sommerville, Ian. "Software Engineering, 2010." *P005* 1.

[43] Whittaker, James A. *How to Break Software: A Practical Guide to Testing with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[44] M.Harman, P.McMinn, J.Texeira de Souza, S.Yoo – Search Based Software Engineering: Techniques, Taxonomy, Tutorial

[45] Wegener, Joachim, André Baresel, and Harmen Sthamer. "Evolutionary test environment for automatic structural testing." *Information and Software Technology* 43.14 (2001): 841-854.

[46] Gross, Florian, Gordon Fraser, and Andreas Zeller. *"Search-based system testing: high coverage, no false alarms."* Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM, 2012.

[47] Inkumsah, Kobi, and Tao Xie. *"Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution.", Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 2008.

[48] Grefenstette, John J. *"Genetic algorithms for changing environments."* *PPSN*. Vol. 2. 1992.

[49] Srinivas, M., and Lalit M. Patnaik. *"Adaptive probabilities of crossover and mutation in genetic algorithms.", Systems, Man and Cybernetics, IEEE Transactions on* 24.4 (1994): 656-667.

[50] *"Crawling AJAX-Based Web Applications through Dynamic Analysis of User Interface State Changes",* Ali Mesbah (University of British Columbia), Arie Van Deursen, Stefan Lenselink (Delft University of Technology)

[51] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, "*The Java Language Specification"*, Java SE 8 Edition, *https://www.java.com*

[52] Damerau–Levenshtein distance: *http://en.wikipedia.org/wiki/Damerau-Levenshtein_distance*

[52] JDOM: *www.jdom.org* ; *http://www.studytrails.com/java/xml/jdom2/java-xml-jdom2-introduction.jsp*

[53] M.Mitchell, S.Forrest, J.H.Holland – "*The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance"*

[54] Java DataBase Connectivity: *http://www.oracle.com/technetwork/java/overview-141217.html*

[55] XAMPP: *https://www.apachefriends.org/it/index.html*

[56] Tudulist: *http://www.oracle.com/technetwork/articles/java/securityperf-rest-ajax-177520.html https://app.rasc.ch/tudu/tudu/welcome*

[57] TaskFreak: *http://www.taskfreak.com/*