laSalle

UNIVERSITAT RAMON LLULL

Escola Tècnica Superior d'Enginyeria La Salle

Treball Final de Màster

Màster Universitari en Enginyeria Informàtica i la seva gestió

Estudi de tècniques d'optimització per al sistema classificador XCS

Alumne
Jordi Salvador Pont

Professor Ponent
Albert Orriols Puig

ACTA DE L'EXAMEN DEL TREBALL FI DE CARRERA

Reunit el Tribunal qualificador en el dia de la data, l'alumne

D. Jordi Salvador Pont

va exposar el seu Treball de Fi de Carrera, el qual va tractar sobre el tema següent:

Estudi de tècniques d'optimització per al sistema classificador XCS

Acabada l'exposició i contestades per part de l'alumne les objeccions formulades pels Srs. membres del tribunal, aquest valorà l'esmentat Treball amb la qualificació de

Barcelona,

VOCAL DEL TRIBUNAL VOCAL DEL TRIBUNAL

PRESIDENT DEL TRIBUNAL

Jordi Salvador Pont

Treball Final de Master

Estudi de tècniques d'optimització per al sistema classificador XCS

Ponent: Albert Orriols Puig

Enginyeria i Arquitectura La Salle

21 de setembre de 2010

Índex

Índex de taules

Índex de figures

Abstract

El present Treball Final de Master es proposa estudiar i implementar diferents alternatives per a optimitzar el sistema classificador XCS. Es tracta d'una optimització no tan sols de rendiment (velocitat d'execució) sinó també d'ús de memòria: millorant el rendiment es poden tractar problemes més grans, i reduint l'ús de memòria es millora l'escalabilitat de la implementació.

Les alternatives estudiades en aquest treball consisteixen en una representació alternativa de les regles que, a més de ser més eficient en l'ús de la memòria, permetran l'ús d'instruccions SIMD (*Single Instrucion, Multiple Data*) amb la qual es podrà millorar el temps d'execució.

La metodologia de treball consisteix en els següents passos: en primer lloc, s'estudia el funcionament del sistema classificador XCS i se'n fa una primera implementació, avaluant-ne el rendiment. A continuació s'estudien les diferents tècniques d'optimització i es fan implementacions que les emprin. Finalment s'avalua el rendiment d'aquestes noves implementacions i es comparen els resultats amb la implementació base, estudiant-ne els canvis i els seus motius.

Capítol 1

Introducció

1.1 El perquè de l'aprenentatge artificial

Des de la seva aparició a mitjans segle xx, l'ús dels ordinadors ha anat augmentant i extenent-se progressivament ocupant cada cop més àmbits de la vida personal i professional, en entorns científics i industrials. Hi ha una sèrie d'aplicacions, tant per a ús personal com per a ús científic o industrial, que requereixen una certa capacitat d'aprenentatge, especialment aquelles aplicacions que són massa complexes per a poder dissenyar un algorisme de forma manual o bé perquè s'ha d'autocustomitzar un cop l'aplicació s'ha entregat al client.

- Un primer exemple pot ser un sistema de reconeixement facial. Per a un humà, reconèixer un rostre és força senzill (com també identificar-lo amb una persona concreta, com ara un amic o familiar), i es fa gairebé immediatament. Per a un ordinador, en canvi, és una tasca extremadament difícil. Emprant mecanismes d'aprenentatge artificial es pot dissenyar un sistema que aprengui i entrenar-lo per a reconèixer les cares humanes.

- Un segon exemple pot consistir en un sistema de reconeixement de la veu. Aquest sistema té una dificultat afegida: cada persona té una veu diferent, i no es pot entrenar un sistema perquè sigui capaç de reconèixer-les totes. En l'exemple anterior l'aprenentatge finalitzava un cop el producte s'havia entregat a l'usuari final, en aquest cas no: un cop l'usuari disposa del producte, aquest ha de seguir una segona

fase d'entrenament per tal d'adaptar-se a la veu de l'usuari. Per tant, el producte ha de ser capaç de seguir aprenent un cop lliurat al client.

Una tècnica força coneguda d'aprenentatge artificial són els sistemes classificadors (LCS, *learning classifier systems*), ideats per John Henry Holland als anys setanta (Holland [1976], Holland and Reitman [1978]). Un dels sistemes classificadors més coneguts és l'XCS, proposat per Stewart Wilson l'any 1995 (Wilson [1995]).

1.2 De la tractabilitat a la practicabilitat

Actualment els sistemes classificadors s'estan implantant en una gran quantitat d'àmbits (Bull et al. [2008]), com per exemple en l'anàlisi de proteïnes, la detecció de càncers de pulmó o pròstata i la detecció i classificació de vehicles a partir d'imatges obtingudes per satèl·lit (Llorà et al. [2007]). Una dificultat molt important amb què s'està trobant la indústria a l'hora de servir-se dels sistemes classificadors és la gran quantitat de dades amb què acostumen a haver de treballar, posant a prova el rendiment i l'eficiència dels sistemes classificadors.

El fet que s'estiguin usant sistemes classificadors (incloent l'XCS) fa que es pugui afirmar que són sistemes *tractables*, ara bé, amb les quantitats de dades que han de processar posa a prova que siguin *practicables*. L'esforç que cal dur a terme és el de fer que aquests sistemes siguin practicables, de manera que escalin bé amb la mida de les dades i millori la seva aplicabilitat.

1.3 L'aportació d'aquest treball

El present treball pretén unir-se als esforços que es fan per tal de millorar el rendiment de l'XCS (Llorà and Sastry [2006]), amb una aportació pròpia que consisteix en tres implementacions completes de l'XCS amb el problema del multiplexor:

1. La primera d'aquestes implementacions segueix l'enfocament clàssic de la implementació de l'XCS. Servirà de base per a les comparatives de rendiment.

2. La segona implementació consistirà en una optimització en la representació dels classificadors. La resta d'implementacions mantindran aquesta representació.

3. La tercera implementació se servirà d'instruccions SIMD presents a l'arquitectura de la màquina per tal d'accelerar la part més costosa de l'XCS.

1.4 Esquema de la memòria

Un cop vist aquest primer capítol introductori, la memòria evoluciona de la següent manera:

- El capítol 2 entrarà més a fons en la intel·ligència artificial, l'aprenentatge artificial i els sistemes classificadors. Servirà de base teòrica per al lector que no conegui aquest àmbit de coneixement. Malgrat tot, no pretén ser una exposició profunda ni extensa, simplement unes nocions bàsiques de l'XCS i del seu context.
- El capítol 3 exposarà la primera implementació del sistema classificador XCS. Serà el capítol que explicarà el seu funcionament a través d'aquesta implementació sense optimitzar, el rendiment de la qual servirà de base per a les comparatives posteriors amb les implementacions optimitzades.
- Els capítols 4 i 5 exposaran, respectivament, les optimitzacions servint-se de mapes de bits i les instruccions SIMD. Cadascun d'aquests capítols exposa els fonaments teòrics relatius a la tècnica d'optimització que s'empra, quins resultats s'esperen obtenir, com s'ha modificat la implementació anterior i els resultats obtinguts. Finalment s'inclou un apartat de comparativa i reflexions.
- El capítol **??** clou la memòria, exposant les conclusions obtingudes de la realització del treball i quines línies de futur queden obertes.

Aquesta distribució estructura la memòria en dues grans parts, cadascuna de les quals versa sobre els dos eixos fonamentals del treball — l'aprenentatge artificial i l'optimització. Els capítols 1–3 formen la primera part, dedicada a explicar el punt de partida, el *background*, la motivació del treball. Els capítols 4–5 formen la segona part, on es desenvolupa l'objectiu del treball.

Capítol 2

Punt de partida

Havent justificat, al capítol introductori, l'existència i les aplicacions de l'aprenentatge artificial (*machine learning*), l'objectiu d'aquest capítol és presentar quin és el seu estat actual, és a dir, quines són actualment les seves principals tècniques.

En primer lloc se'n farà una introducció, explicant els diferents tipus d'aprenentatge que es poden trobar, i una breu presentació dels algorismes genètics — fonamentals en alguns dels mecanismes d'aprenenatge artificial que es descriuran, incloent el sistema classificador XCS. A continuació es parlarà justament de l'aprenentatge artificial basat en algorismes genètics, citant les seves principals famílies però centrant-se sobretot en els sistemes classificadors. El capítol acabarà amb una presentació del sistema classificador XCS.

2.1 Introducció

2.1.1 Aprenentatge artificial

L'aprenentatge artificial és una disciplina científica que intenta dissenyar aplicacions que puguin simular un procés d'aprenentatge, de manera que la màquina que aprèn, a partir d'una experiència, millori el seu rendiment a l'hora de realitzar una determinada tasca.

Dins de l'aprenentatge artificial (ML: *machine learning*) se'n poden distingir tres tipus:

1. Aprenentatge supervisat (SL: *supervised learning*): consisteix en ex-

treure un model o funció a partir d'unes dades d'entrenament. La característica principal d'aquest tipus d'aprenentatge és que necessita un expert que proveeixi un *feedback* durant el procés d'aprenentatge. Dins de l'aprenentatge supervisat en podem distingir dues variants:

a) Classificació de dades: es parteix d'un conjunt d'elements que s'han de distribuir en diversos grups (ordenar per classes). Per exemple, donat un conjunt de bolets es vol determinar quins són comestibles i quins no. A partir d'uns atributs d'entrada (com ara la mida del coll o el diàmetre), s'ha de determinar un atribut de sortida (comestible o no comestible). Així doncs, la funció de la màquina és prediure a quina classe pertanyen les noves instàncies (en l'exemple, bolets).

b) Regressió de dades: es tracta de trobar una funció que aproximi una sèrie de punts, similar en filosofia al mètode dels mínims quadrats.

2. Aprenentatge no supervisat (UL: *unsupervised learning*): consisteix en extreure un model d'un conjunt de dades d'entrada, que consisteixen únicament en atributs d'entrada i sense cap sortida associada; de manera que el programa no sap què està buscant ni rep cap mena de *feedback*. Generalment l'aprenentatge no supervisat s'aplica quan es vol saber com s'organitzen les dades d'entrada. Les dues principals tècniques en aquesta mena d'aprenentatge són el *clustering* i les regles d'associació.

3. Aprenentatge per reforç (RL: *reinforcement learning*): d'alguna manera se situa entre l'aprenentatge supervisat i no supervisat. Consisteix en aprendre *què s'ha de fer* interactuant amb l'entorn. L'agent que aprèn du a terme accions per tal d'aconseguir un objectiu (o més d'un), i de l'entorn rep recompenses positives o negatives com a conseqüència de les seves accions. Observi's que, per una banda, l'agent no disposa de cap exemple del qual aprendre, per la qual cosa ha d'aprendre de la seva pròpia experiència; per altra banda, interactua amb l'entorn per saber si les seves accions estan mal o ben encaminades de cara a aconseguir l'objectiu.

2.1.2 Algorismes genètics

Els algorismes genètics (GA: *genetic algorithm*, Holland [1975] i Goldberg [1989]) són una família dins de la computació evolutiva. Consisteix en l'estudi del disseny, implementació i anàlisi de tècniques computacionals que s'inspiren en l'evolució de la vida biològica al món natural. La computació

evolutiva parteix d'una sèrie de principis biològics, especialment de la teoria de l'evolució presentada a mitjans del segle XIX, que parteix de la idea segons la qual els individus d'una espècie disposen d'un material genètic (el *genotip*) que defineix la seva constitució genètica. Aquest genotip, conjuntament amb la interacció amb l'entorn, forma el *fenotip*, és a dir, la constitució observable de l'organisme.

Hi ha quatre conceptes bàsics en computació evolutiva:

1. Reproducció: un individu es reprodueix, transferint el genotip de pares a fills.

2. Mutació: durant la transferència del genotip es poden produir errors.

3. Competició: conseqüència de la creació de nous individus en un entorn amb recursos limitats.

4. Selecció: conseqüència de la competició entre els individus, en què uns individus són triats en lloc d'uns altres.

Tot això resulta en un cicle en què els individus es reprodueixen i tenen lloc mutacions, de manera que els individus millor adaptats a l'entorn tenen més probabilitat de sobreviure ("selecció natural"). Els algorismes genètics no són l'única família que es pot trobar dins de la computació evolutiva, però és la que incorporen els sistemes classificadors (com ara l'XCS), i per tant és la que explicarem en aquest apartat.

Per tal d'implementar la selecció natural i la competició, els algorismes genètics incorporen una *funció d'avaluació*, que és responsable d'assegurar la qualitat de cadascun dels individus, que es fa explícita amb un valor de "fitness". S'han emprat una gran varietat de funcions d'avaluació en algorismes genètics, ja siguin funcions matemàtiques o bé funcions subjectives en què els usuaris escullen els millors individus a partir d'un conjunt de candidats. El disseny d'aquestes funcions d'avaluació és d'una importància extrema, ja que és el punt clau per tal que l'algorisme genètic pugui crear individus de qualitat i mantingui a la població els millors individus.

L'evolució dels individus de la població es fa mitjançant l'acció combinada de quatre operadors:

Selecció L'operador de selecció tria els millors individus de la població, simulant el mecanisme de supervivència dels millors. Fins al moment s'han presentat diferents esquemes de selecció: *roulette-wheel selection*, selecció estocàstica (aquests dos mètodes donen a cada individu una probabilitat de supervivència proporcional a la seva qualitat), selecció per torn (*tournament selection*), selecció per truncament, etc.

Encreuament L'operador d'encreuament combina la informació genètica de dues o més solucions per a crear una descendència que sigui possiblement millor. Aquesta recombinació juga un paper important en els algorismes genètics, ja que hauria de detectar trets importants de les solucions i intercanviar-los per tal de generar individus millors que no siguin idèntics als seus progenitors. Òbviament, s'han presentat una gran varietat d'operadors d'encreuament.

Mutació La mutació introdueix errors aleatoris en la transferència de la informació genètica dels progenitors cap a la descendència. Aquest operador, doncs, s'aplica individualment sobre els individus. S'han dissenyat diferents operadors de mutació, la majoria dels quals consisteixen en introduir un o més canvis aleatoris aplicats sobre gens individuals.

Substitució Un cop la població ha passat per l'encreuament i la mutació, la població descendent substitueix l'original. Hi ha diferents esquemes de substitució: n'hi ha que consisteixen en una substitució completa, d'altres en una substitució elitista.

2.2 GBML

GBML són les sigles de *Genetic-Based Machine Learning*, que podríem traduir com "Aprenentatge artificial basat en Algorismes Genètics". Hi podem distingir quatre branques principals:

1. Learning Classifier Systems (LCS)
2. Iterative rule learning (IRL).
3. Genetic Cooperative-Competitive Learning (GCCL).
4. Organizational Classifier System (OCS).

Aquest apartat se centrarà en els sistemes classificadors. Per a una presentació de les altres tècniques es pot consultar el capítol 2 de la tesi del doctor Albert Orriols (Orriols Puig [2008]).

2.2.1 Learning Classifier Systems

Els sistemes classificadors (LCS: *Learning Classifier System*), presentats per John Holland (Holland [1976]), són un mètode d'aprenentatge artificial basat en algorismes genètics i actualment està sent àmpliament emprat (Bull et al. [2008]). Hi ha tres aspectes claus que s'han mantingut des de la concepció dels sistemes classificadors:

1. La representació del coneixement basat en classificadors (generalment implementats com a regles) que permeten al sistema mapejar estats sensorials amb accions.

2. Un mecanisme que permet qualificar els classificadors, és a dir, quins són millors, més precisos, més bons.

3. Un algorisme que fa *evolucionar* la base de coneixement — típicament sol ser un algorisme genètic.

Dins dels sistemes classificadors es poden trobar dues grans famílies: els sistemes d'estil Michigan i els d'estil Pittsburgh. La diferència principal entre ells és que en els primers la solució la formen tots els individus (que són classificadors), mentre que en els segons un individu és una solució — més o menys bona — a tot el problema. Una presentació general (i actual) sobre sistemes classificadors, les seves principals variants i implementacions es pot trobar a Sigaud and Wilson [2007].

2.3 El sistema classificador XCS

El classificador XCS és un sistema classificador d'estil Michigan, proposat per Stewart Wilson l'any 1995 (Wilson [1995]). El seu esquema bàsic es presenta a la figura 2.1, extreta de Wilson [1998].

L'èxit d'aquest sistema classificador rau en la seva estructura simplificada (respecte sistemes classificadors anteriors) i per basar la qualitat dels classificadors en la predicció de la recompensa (és a dir, un classificador no és més o menys bo segons la recompensa que obté, sinó segons com de bé prediu la recompensa que obtindrà). Aquestes són les dues principals novetats de l'XCS sobre d'altres sistemes d'estil Michigan.

La resta d'aquest capítol es basa en el capítol 3 de la tesi del doctor Albert Orriols (Orriols Puig [2008]).

2.3.1 Representació del coneixement

En l'XCS, un classificador és un conjunt format per una regla, una acció i uns atributs. La finalitat de la regla és discriminar quins classificadors són aplicables amb l'exemple (la mostra) que l'entorn proveeix — és a dir, quan es pot usar el classificador—; l'acció indica justament l'acció a prendre o bé la classe de la mostra; els atributs mantenen dades estadístiques i indiquen, entre d'altres coses, la qualitat del classificador.

La regla d'un classificador és un mot en l'alfabet $\{0, 1, \#\}$, de longitud finita i idèntica per a tots els classificadors. Les mostres de l'entorn tenen la mateixa longitud, però estan codificades en l'alfabet binari. El fet que la regla disposi d'un símbol extra és per a permetre la generalització. Per tal

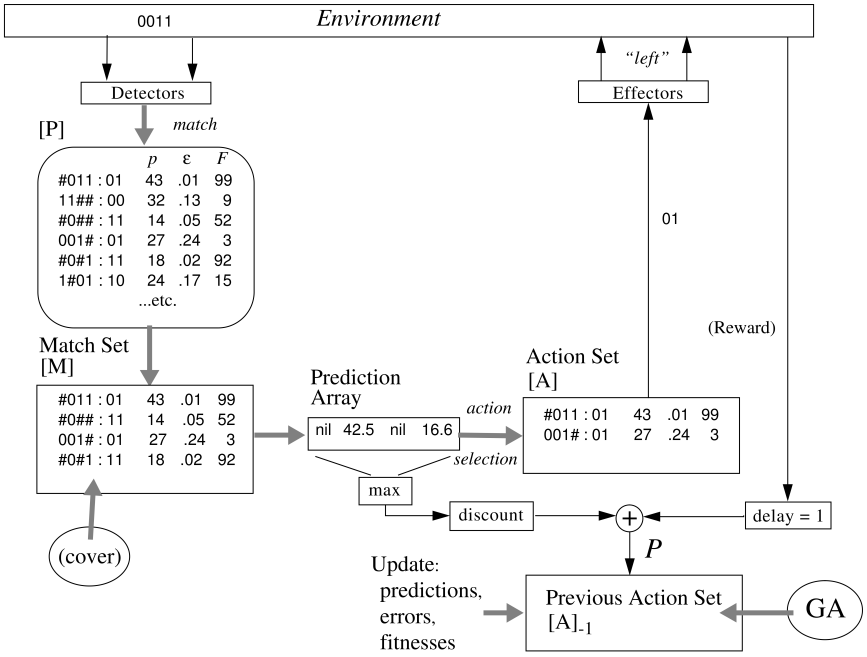

Figura 2.1: Esquema del sistema classificador XCS

de saber si un classificador és aplicable a la mostra, es comparen mostra i regla símbol per símbol. Cal que tots els símbols coincideixin, i el símbol # coincideix tant amb el símbol 0 com amb el símbol 1. Llavors es diu que el classificador fa *match* amb la mostra.

Els principals atributs d'un classificador són:

- Una estimació de la recompensa que el classificador rebrà si el classificador fa *match* amb la regla i es pren l'acció (o la classificació) que proposa; aquest paràmetre se simbolitza amb p.
- Una estimació de l'error mig entre la recompensa que estima i la que realment rep; aquest paràmetre se simbolitza amb ϵ.
- El *fitness* (f), que es pot considerar com la qualitat del classificador, d'alguna manera comparat amb la resta de classificadors; aquesta qualitat es basa sobretot en el paràmetre p.
- L'experiència (exp), que computa la quantitat de mostres amb què el classificador ha fet *match*, durant tota la seva vida.

- La numerositat (*num*), que indica quantes còpies idèntiques conté aquest classificador. A cadascuna d'aquestes còpies se l'anomena *micro-classificador*, i totes elles formen un *macro-classificador*.

2.3.2 Mètode d'aprenentatge

L'XCS aprèn a base d'anar fent iteracions, en què l'entorn li proveeix una mostra i l'XCS decideix quina acció ha de prendre (o quina classificació ha de fer). L'entorn llavors retorna una recompensa al sistema classificador, el qual actualitza els classificadors en base a la recompensa i als propis atributs dels classificadors. Hi ha dues menes d'iteracions:

1. Les iteracions d'entrenament (*exploration*), que és quan realment l'XCS aprèn.
2. Les iteracions de prova o testeig (*exploitation*), que serveixen per a comprovar com d'acurat és l'XCS en un moment determinat. Durant aquestes iteracions no es modifica la població (és a dir, ni es creen ni es destrueixen nous classificadors, ni se'n modifiquen els atributs).

Hi ha diferents maneres d'alternar iteracions d'entrenament i de testeig. Una ràtio pot ser 1:1, en què per cada iteració d'entrenament se'n fa una de testeig. Per a calcular l'*accuracy* del sistema classificador (és a dir, com de bé està predint), es fa una mitja de la recompensa obtinguda durant una quantitat d'iteracions de testeig, aquesta quantitat d'iteracions rep el nom de *window*.

Per tal de compensar els efectes de l'aleatorietat, un determinat experiment (per exemple, el multiplexor de 37 bits) es repeteix una certa quantitat de cops, típicament 25. Cada experiment comença amb una població completament buida de classificadors.

Noti's que en aquest apartat no s'explica quines accions es duen a terme durant una iteració, sinó que això s'explica al capítol 3. L'avaluació dels classificadors és un aspecte força complex que no es tractarà en aquest treball, els lectors interessats en aquest tema poden consultar l'apartat 3.1.3 de la tesi del doctor Albert Orriols (Orriols Puig [2008]).

2.3.3 Generació de nous classificadors

S'ha dit a l'apartat anterior que cada experiment comença amb una població completament buida. Llavors la pregunta és òbvia: com es creen nous classificadors? Hi ha dos mecanismes:

Covering El procediment de *covering* es duu a terme durant la formació del *match set*, que és el conjunt format pels classificadors que fan *match*

amb la mostra de l'entorn. Un cop format aquest conjunt es mira que, per a cada acció o classe, hi hagi almenys un classificador (en aquest conjunt) que proposi tal acció o classe. Si alguna d'elles no té cap classificador, llavors s'activa el procediment de *covering*, que crea un classificador que fa *match* amb la mostra i proposa la classe o acció que ha quedat "sense representació". La regla del classificador és idèntica a la mostra, però hi ha una certa probabilitat (especificada pel paràmetre $P_\#$)[1] que un símbol de la regla esdevingui el símbol #.

Les iteracions de testeig no criden mai el procediment de *covering*.

Algorisme genètic En algunes generacions es crida l'algorisme genètic, que crea nous classificadors a partir dels millors de la població. Concretament tria dos classificadors d'un conjunt anomenat *action set*, ja sigui mitjançant *roulette-wheel selection*, *tournament selection* o algun altre mecanisme de selecció. Aquests dos classificadors són els pares, que donen lloc a dos nous classificadors (els fills). Hi ha una probabilitat, especificada pel paràmetre χ, que s'apliqui l'operador d'encreuament sobre els fills. El paràmetre μ indica la probabilitat de mutar un símbol de la regla dels fills. Pel que fa a l'encreuament, hi ha dos tipus bàsics possibles: el *two-point crossover* i l'*uniform crossover*. Finalment alguns atributs dels nous classificadors es modifiquen i s'introdueixen a la població.

Les iteracions de testeig no criden mai l'algorisme genètic.

Com que la població té una mida màxima, pot ser que a l'introduir-hi nous classificadors se n'hagin d'eliminar alguns d'existents; en aquest cas el propi sistema classificador disposa d'un mecanisme per a seleccionar quin classificador cal eliminar.

2.3.4 Per què funciona l'XCS?

Explicar per què el sistema classificador XCS funciona és quelcom fora de l'abast d'aquest treball. Si el lector està interessat en aquest tema, pot trobar una bona explicació (sense entrar en detalls matemàtics) a l'apartat 3.1.6 de la tesi del doctor Albert Orriols (Orriols Puig [2008]).

[1] El sistema classificador XCS disposa d'una sèrie de paràmetres que controlen el funcionament d'una sèrie d'aspectes relacionats amb l'aprenentatge, l'algorisme genètic, etc.

Capítol 3

La implementació base

3.1 Introducció

Després d'haver explicat el propòsit d'aquest treball i el seu marc científic (l'aprenentatge artificial), arriba el moment d'explicar en detall el funcionament del sistema classificador XCS. A tal fi es veurà, en primer lloc, una visió global sobre el funcionament del sistema, per tal que els detalls no facin perdre la visió general del seu funcionament. Després s'explicarà quin és l'entorn (el "problema") emprat en aquest treball, i finalment es procedirà a la descripció algorísmica detallada del sistema XCS.

Abans d'entrar en matèria, és molt convenient deixar clar que no hi ha una descripció algorísmica que es pugui considerar *oficial* de l'XCS. Aquest fet ha comportat una gran quantitat de problemes a l'hora d'implementar-lo, endarrerint enormement la marxa del treball i causant molts maldecaps. Hi ha un article (Butz and Wilson [2000]) que serveix de descripció algorísmica, però hi ha errors. De fet, aquest mateix article té diferents versions, que difereixen en alguns aspectes. Finalment, un dels dos autors del mencionat article (concretament en Martin Butz) ha publicat dues implementacions — una en C i l'altra en Java —, que també tenen diferències entre elles i amb l'article en qüestió.

Per a la implementació d'aquest treball s'ha seguit una mescla entre la descripció de l'article, les implementacions d'en Martin Butz i les indicacions donades pel ponent del TFM, el doctor Albert Orriols.

3.2 Funcionament general

La figura 2.1, de la pàgina 10, dóna una visió global de com funciona el sistema XCS. El sistema classificador vol aprendre, per la qual cosa interactua amb l'entorn: aquest proporciona una mostra al sistema classificador, el qual — a través d'un procés que s'explicarà tot seguit — decideix quina acció cal emprendre (o a quina classe pertany la mostra donada), i ho comunica a l'entorn, el qual ofereix al sistema classificador una recompensa en funció de l'acció o classe decidida per l'XCS. La mostra es codifica amb una representació binària.

Com s'ho fa el sistema classificador per decidir la classe o l'acció? Tal com s'ha dit al capítol anterior, l'aprenentatge es realitza a través d'iteracions, amb les quals es va generant una població de classificadors que acabarà per predir perfectament l'acció o classe correcta. Cada iteració d'aprenentatge segueix el següent procediment:

1. L'XCS disposa d'una població de classificadors. Un classificador és un conjunt format per una regla, una acció (o classe) i uns atributs. La regla (que també segueix una representació binària) serveix per a decidir si el classificador és aplicable a la mostra rebuda de l'entorn. L'acció indica quina classe o acció proposa el classificador per a dur a terme.

   Per a cada classificador, l'XCS avalua la seva regla i comprova si és aplicable amb la mostra, és a dir, si fa *match*. Amb tots els classificadors que fan *match* es forma l'*action set*, que és justament el conjunt de classificadors que són aplicables a la mostra rebuda de l'entorn.

   Si per a alguna classe no hi ha cap classificador en el *match set*, llavors es crida el procediment de *covering*, per tal de garantir que al *match set* hi ha representades totes les classes.

2. A partir del *match set*, es decideix quina acció cal prendre. Cal tenir en compte que al *match set* hi sol haver més d'un classificador, i que poden proposar accions o classes diferents. Per a decidir quina de les accions es pren es pot seguir un procés aleatori (triar-ne una a l'atzar) o bé tenir en compte els atributs dels classificadors (formant el que es coneix amb el nom de *prediction array*).

3. Un cop s'ha decidit quina acció es duu a terme, es crea l'*action set*, que és el conjunt de classificadors del *match set* que proposen l'acció o classe que s'ha decidit. Es comunica a l'entorn aquesta decisió i se n'obté la recompensa, que s'utilitza per a actualitzar alguns paràmetres dels classificadors de l'*action set*.

4. De tant en tant es crida un algorisme genètic, per tal de generar nous classificadors que millorin la qualitat de la població.

Segons de quin entorn es tracti caldrà realitza més o menys iteracions per tal d'aconseguir que el sistema *convergeixi*, és a dir, que arribi a predir perfectament l'acció o classe correcta. Per tal de calcular l'*accuracy* (és a dir, com de bo és el sistema predint l'acció la classe correcta) s'intercalen iteracions de testeig amb les iteracions d'aprenentatge. Les iteracions de testeig no creen ni eliminen classificadors, ni en modifiquen els seus atributs; l'acció se selecciona en funció de les característiques dels classificadors (en les iteracions d'aprenentatge se selecciona aleatòriament).

3.3 L'entorn

Per a aquest treball l'entorn ha estat el problema del multiplexor, que és l'entorn generalment emprat a l'hora de crear *benchmarks* de rendiment. El seu funcionament és molt senzill, ja que consisteix en simular justament un multiplexor.

Una mostra consisteix en una cadena de símbols en l'alfabet $\{0, 1\}$, de manera que els primers n símbols serien els bits de selecció o d'adreça, i la resta de símbols (2^n) serien les entrades del multiplexor. Òbviament, hi ha diferents "versions" de l'entorn, segons el número de símbols: així tenim el multiplexor de 6 bits (per a $n = 2$), d'11 bits (per a $n = 3$), de 20 bits (per a $n = 4$), etc. Per a aquest treball s'han usat els multiplexors de 6, 11, 20 i 37 bits.

Exemple Consideri's el cas del multiplexor de 6 bits. Si la mostra està formada pels sis símbols següents: 010110, llavors la sortida del multiplexor seria 1, ja que els dos primers símbols (01) indiquen que la sortida ha de ser el segon símbol (després dels dos primers), que és justament un 1. En canvi, si la mostra fos 101001, llavors la sortida seria 0.

3.3.1 Implementació

La implementació del multiplexor resideix als arxius *multiplexer.c* i *multiplexer.h*. Ofereix quatre funcions públiques (és a dir, accessibles des de qualsevol altra unitat de traducció):

`MUX_init` Aquesta funció inicialitza les estructures de dades internes emprades per a les funcions següents. Rep el número de bits d'adreça del multiplexor, és a dir, el valor de n. La longitud, en símbols, de la mostra serà justament $n + 2^n$.

`MUX_get_value` Aquesta funció retorna una mostra de l'entorn (generada aleatòriament), ubicada en un array estàtic (la funció anterior obté memòria per a aquest array). Diferents crides a aquesta funció sobreescriuran el contingut d'aquest array. No s'ha d'alliberar la memòria apuntada pel valor de retorn d'aquesta funció.

`MUX_end` Aquesta funció realitza la tasca inversa de la funció `MUX_init`; concretament allibera la memòria emprada per l'array que guarda la mostra de l'entorn.

`MUX_get_reward` Aquesta funció rep l'acció predita pel sistema classificador i, en funció d'aquest acció i de l'última mostra obtinguda, retorna la recompensa adequada (0 en cas d'una predicció errònia, 1000 en cas d'una predicció correcta).

S'empren tres variables globals:

`k_bits` Enter que guarda el número de símbols d'adreça del multiplexor.

`n_bits` Enter que guarda la longitud total (en símbols) de les mostres.

`sigma` Punter a la zona de memòria que conté l'última mostra generada per l'entorn.

3.4 Descripció algorísmica

L'objectiu d'aquest apartat és descriure amb detall els passos introduïts al descriure el funcionament general del sistema classificador XCS. Pel que fa l'algorisme genètic, s'ha considerat oportú dedicar-li un apartat propi (el 3.5.

La implementació feta en aquest treball permet realitzar múltiples experiments de forma seqüencial. Per tal d'obtenir dades i gràfiques significatives a l'hora d'avaluar el rendiment i la qualitat del sistema classificador, cal dur a terme diversos experiments per a compensar els efectes dels nombres aleatoris — pot ser que un experiment doni uns resultats molt bons i un altre experiment doni uns resultats diferents, ja que el funcionament de l'XCS depèn, en molts aspectes, de la probabilitat.

3.4.1 Paràmetres de l'XCS

El sistema classificador XCS disposa d'una sèrie de paràmetres que controlen certs aspectes del seu funcionament. No s'explicarà aquí el significat de cadascun d'ells i quin impacte té en el rendiment o en la qualitat de l'aprenentatge, sinó que s'explicaran a mesura que facin acte d'aparició.

A la taula 3.1 s'indica cadascun d'aquests paràmetres, el nom de la macro que els identifica al codi font, el valor típic que sol prendre segons Butz and

Paràmetre	Macro	Valor típic	Valor concret
N	PAR_N	—	1000
β	PAR_BETA	0.1–0.2	0.2
α	PAR_ALPHA	0.1	0.1
ϵ_0	PAR_EPSILON_0	—	10
ν	PAR_NU	5	5
θ_{GA}	PAR_THETA_GA	25–50	25
χ	PAR_CHI	0.5–1.0	0.8
μ	PAR_MU	0.01–0.05	0.04
θ_{del}	PAR_THETA_DEL	20	20
δ	PAR_DELTA	0.1	0.1
θ_{sub}	PAR_THETA_SUB	20	20
$P_\#$	PAR_P_SHARP	0.33	0.5
p_I	PAR_P_I	0	10
ϵ_I	PAR_EPSILON_I	0	0
f_I	PAR_F_I	0	0.01
θ_{mna}	PAR_THETA_MNA	—	2
$dGAS$	PAR_DO_GAS	—	1
τ	PAR_TAU	0.4	0.4
w	PAR_WINDOW	—	100

Taula 3.1: Valors dels paràmetres del sistema XCS

Wilson [2000] i el valor concret definit al codi font (seguint la implementació en Java d'en Martin Butz). Aquests valors s'han mantingut constants al llarg de totes les implementacions. Els paràmetres N i w han canviat segons el problema que, en cada cas, calia resoldre.

L'arxiu *config.h* conté les directives del pre-processador que defineixen i donen valor a aquests paràmetres; cada cop que es canvia el valor d'un ells cal recompilar tota l'aplicació (la qual cosa té una durada aproximada de menys de 5 segons).

3.4.2 Inicialització

Quan l'usuari executa el sistema classificador, pot passar una sèrie de paràmetres per línia de comandes:

`-a` Permet especificar en quin arxiu s'escriuran les dades estadístiques de l'*accuracy*. Aquest arxiu es pot usar després per a generar gràfiques amb eines com ara *gnuplot*. Per defecte l'arxiu es diu *accuracy.dat*.

`-e` Permet especificar el número d'experiments a dur a terme. Per defecte se'n realitza un.

`-i` Permet especificar el número d'iteracions a dur a terme. Per defecte es fa una sola iteració. Cal tenir en compte que per a cada iteració se'n realitzen dues: una d'aprenentatge i una de testeig. Per tant, el nombre total d'iteracions és el doble del valor especificat.

`-k` Permet especificar el número de bits d'adreça del multiplexor. Per defecte val 2, és a dir, es realitza el multiplexor de 6 bits.

`-p` Permet especificar en quin arxiu s'escriuran les dades estadístiques de la població (relació de macro-classificadors respecte del màximd de població en micro-classificadors). Aquest arxiu es pot usar després per a generar gràfiques amb eines com ara *gnuplot*. Per defecte l'arxiu es diu *population.dat*.

`-v` Aquesta opció no rep cap argument, i fa que, al final de cada experiment, es mostri la població de classificadors (regla, acció i atributs).

El primer pas en l'execució, abans d'iniciar el primer experiment, consisteix en parsejar aquestes opcions. A continuació s'inicialitza el generador de números aleatoris a partir de la data i l'hora en el moment d'execució, i a continuació es crida la funció `xcs_run`, que durà a terme l'execució dels experiments. Tot això té lloc en un únic fitxer de codi font: *main.c*.

Exemple Per tal d'executar el problema del multiplexor de 37 bits, fent 25 experiments de mig milió d'iteracions cadascun, es crida l'aplicació de la següent manera:

```
$ ./xcs_run -k 5 -e 25 -i 500000
```

3.4.3 Els experiments

En els projectes de recerca és molt important disposar de dades estadístiques per tal de generar taules i gràfiques. La implementació feta per a aquest treball té en compte aquest fet i permet guardar dades sobre l'evolució de l'*accuracy* i de la població. Per a cada experiment, cada cert nombre d'iteracions es recull l'*accuracy* i la població i es guarda en una zona de memòria dedicada a tal efecte. Un cop realitzats tots els experiments, es calcula la mitjana i la desviació típica i es guarden els resultats en dos arxius. Abans de realitzar els experiments de demana memòria per a guardar les dades estadístiques.

A banda d'això, també cal reservar memòria per a la població i per a estructures de dades auxiliars; també cal inicialitzar l'entorn. Es reserva memòria de tal manera que durant les iteracions no calgui demanar-ne més (a banda d'allò que s'ubiqui a la pila del sistema); al finalitzar tots els experiments s'allibera la memòria reservada. Entre experiment i experiment es

"netegen" les zones de memòria per tal que no quedin rastres dels experiments anteriors i siguin independents entre ells.

La funció `write_statistics` té per finalitat escriure a fitxer les dades estadístiques; es crida un cop s'han realitzat tots els experiments. Aquesta funció, com la funció `xcs_run` (que duu a terme tots els procediments explicats en aquest apartat), es troba a l'arxiu *xcs.c*.

La funció `xcs_run_experiment` és l'encarregada de dur a terme un experiment. S'encarrega d'executar les iteracions (tant d'aprenentatge com de testeig), recollir quan toca les dades estadístiques i, al final de totes les iteracions, mostrar la població (si s'ha passat l'opció `-v` per línia de comandes).

Cada experiment comença amb una població buida de classificadors.

3.4.4 Les iteracions

Les iteracions són el nucli del funcionament del sistema classificador XCS. A grans trets, ja s'ha indicat de quins passos consisteix una iteració, ara és l'hora de descriure detalladament aquests passos. En aquest punt convé recordar que hi ha dues menes d'iteracions:

1. Les iteracions d'aprenentatge són les que s'encarreguen que el sistema XCS aprengui. Les executa la funció `xcs_run_exploration`, definida a l'arxiu *xcs.c*.

```
1  static inline void
2  xcs_run_exploration (const int curr_iter)
3  {
4    MUX_get_value ();
5    int action;
6    int reward;
7    double ts_average = 0;
8    MS_create ();
9    while (!match_set.ms_actions[0] || !match_set.ms_actions[1])
10     {
11       while (micro_popsize == PAR_N)
12         delete_from_population (1);
13       MS_covering (curr_iter);
14     }
15   action = getSingleBinaryDigit ();
16   AS_create (action);
17   reward = MUX_get_reward ('0' + action);
18   AS_update_params (reward);
19   for (size_t i = 0; i < action_set.as_size; i++)
20     {
21       ts_average += action_set.as_elements[i]->cl_ts *
22         action_set.as_elements[i]->cl_num;
```

```
23      }
24    ts_average /= action_set.as_microsize;
25    if (curr_iter - ts_average > PAR_THETA_GA)
26      run_ga (curr_iter);
27  }
```

2. Les iteracions de testeig permeten avaluar el rendiment del sistema classificador fins al moment. Les executa la funció de l'arxiu *xcs.c*, `xcs_run_exploitation`.

```
1   static inline void
2   xcs_run_exploitation (void)
3   {
4     MUX_get_value ();
5     int action;
6     int reward;
7     MS_create ();
8     PA_create ();
9     if (PA_array[0] > PA_array[1])
10      action = 0;
11    else
12      action = 1;
13    AS_create (action);
14    reward = MUX_get_reward ('0' + action);
15    if (reward)
16      hits++;
17  }
```

En aquest apartat l'atenció es focalitzarà en les iteracions d'aprenentatge, fent esment de les diferències amb les iteracions de testeig quan sigui oportú.

3.4.5 Formació del *match set*

El primer pas d'una iteració consisteix en obtenir una mostra de l'entorn (cridant la funció `MUX_get_value`) i, amb aquesta mostra, creant el *match set*. A tal efecte, es recorre la població analitzant cada classificador, comparant la seva regla amb la mostra obtinguda. La representació d'un classificador es troba a l'arxiu *data.h* i consisteix en els atributs de la taula 3.2.

Per a crear el *match set* es crida la funció `MS_create`, definida a l'arxiu *match_set.c*. Un *match set* és una estructura de dades definida a l'arxiu *data.h* i conté els següents camps:

`ms_size` Quantitat de macro-classificadors que conté el *match set*. El tipus d'aquest camp és `size_t`.

Atribut	Camp	Tipus
Regla	`cl_rule`	`char *`
Acció	`cl_action`	`int`
p	`cl_p`	`double`
ϵ	`cl_epsilon`	`double`
f	`cl_fitness`	`double`
exp	`cl_exp`	`size_t`
t_s	`cl_timestamp`	`size_t`
as	`cl_as`	`double`
num	`cl_num`	`size_t`

Taula 3.2: Atributs del classificador

`ms_actions` Array de dos elements, cadascun dels quals conté la quantitat de macro-classificadors que proposen les accions 0 o 1 (és a dir, predir que la sortida del multiplexor serà 0 o 1), Cada element de l'array té tipus `size_t`.

`ms_elements` Array de punters a classificadors. En lloc de copiar els classificadors de la població al *match set*, es guarden punters als classificadors, de manera que es guanya en rendiment i en ús de memòria. Es considera el pitjor cas, és a dir, que tots els classificadors de la població pertanyin al *match set*.

S'ha dit que per a determinar si un classificador ha d'entrar al *match set* s'analitza la seva regla i es compara amb la mostra (que se sol representar amb la lletra grega σ). Com es fa aquesta comparació? No cal que la regla i la mostra siguin idèntiques, només cal que facin *match*. Per tal això es produeixi, es comparen regla i mostra símbol per símbol, i per a cada símbol s'ha de complir almenys una de les dues condicions següents:

1. Que el símbol de la regla sigui el mateix que el símbol de la mostra.
2. Que el símbol de la regla sigui #, que significa *don't care*. És a dir, mentre que la mostra està codificada en l'alfabet $\{0, 1\}$, la regla es codifica amb l'alfabet $\{0, 1, \#\}$, en què el símbol # actua com un "comodí".

Val la pena donar el codi font de la funció `MS_create`:

```
1 void
2 MS_create (void)
3 {
4   size_t c, i;
5   match_set.ms_size = 0;
```

```
6     match_set.ms_actions[0] = match_set.ms_actions[1] = 0;
7
8     for (c = 0; c < macro_popsize; c++)
9       {
10        for (i = 0; (int) i < n_bits; i++)
11          if (sigma[i] != population[c].cl_rule[i] &&
12              population[c].cl_rule[i] != '#')
13            break;
14        if ((int) i == n_bits)
15          {
16            match_set.ms_actions[population[c].cl_action]++;
17            match_set.ms_elements[match_set.ms_size] = &population[c];
18            match_set.ms_size++;
19          }
20      }
21  }
```

En aquesta funció apareixen les següents variables globals que encara no s'han explicat:

`match_set` Variable de tipus `match_set_t`, definit a l'arxiu *data.h*, que conté els camps explicats anteriorment. Es reaprofita el *match set* per a totes les iteracions (òbviament al principi de cada iteració es posen a zero els camps `ms_size` i `ms_actions`).

`macro_popsize` Variable de tipus `size_t` que conté la mida de la població en macro-classificadors.

`sigma` Punter a la mostra obtinguda de l'entorn (tipus `char *`).

`population` La població de classificadors, que està definida com un array de classificadors.

Covering En les iteracions d'aprenentatge, es requereix que el *match set* contingui, almenys, un classificador per a cada acció possible. En el cas d'aquest treball, només hi ha dues accions possibles. Si el *match set* no conté cap classificador que proposi una de les dues accions (o bé cap, si el *match set* està buit), llavors es duu a terme un procediment anomenat *covering*. Consisteix en crear classificadors que facin *match* amb la mostra (i que, per tant, formaran part del *match set*), fins que totes les classes estiguin representades al *match set*. Com es crea un classificador per *covering*?

- La regla del nou classificador és idèntica a la de la mostra, però per a cada símbol, i amb una probabilitat igual al paràmetre $P_{\#}$, en lloc de posar el símbol de la mostra es posa el símbol # (*don't care*). La regla segueix fent *match* amb la mostra, però esdevé més general.

Paràmetre	Valor que pren
Acció	Aleatori
p	p_I
ϵ	ϵ_I
f	f_I
exp	0
ts	iteració actual
as	1
num	1

Taula 3.3: Inicialització d'un classificador generat per *covering*

- L'acció del nou classificador és justament aquella que no estava representada al *match set*. Si no n'hi ha cap de representada, s'escull la zero.
- Els atributs s'inicialitzen segons s'indica a la taula 3.3.

La funció `MS_covering`, definida a l'arxiu *match_set.c*, implementa el procediment de *covering*. Val la pena donar el codi font d'aquesta funció:

```
1  void
2  MS_covering (const int curr_iter)
3  {
4    for (int i = 0; i < n_bits; i++)
5      {
6        population[macro_popsize].cl_rule[i] = sigma[i];
7        if (getDoubleNumber () < PAR_P_SHARP)
8          population[macro_popsize].cl_rule[i] = '#';
9      }
10   population[macro_popsize].cl_p = PAR_P_I;
11   population[macro_popsize].cl_epsilon = PAR_EPSILON_I;
12   population[macro_popsize].cl_f = PAR_F_I;
13   population[macro_popsize].cl_exp = 0;
14   population[macro_popsize].cl_ts = curr_iter;
15   population[macro_popsize].cl_as = 1;
16   population[macro_popsize].cl_num = 1;
17   population[macro_popsize].cl_action =
18     (match_set.ms_actions[0]) ?
19     (match_set.ms_actions[1]++, 1) : (match_set.ms_actions[0]++, 0);
20   match_set.ms_elements[match_set.ms_size] = &population[macro_popsize];
21   match_set.ms_size++;
22   macro_popsize++;
23   micro_popsize++;
24 }
```

La variable `micro_popsize` conté la mida de la població en microclassificadors. Abans de cridar el procediment de *covering* es comprova la mida de la població. Si està totalment plena, abans de cridar el *covering* s'elimina un classificador de la població. L'apartat 3.6.1 explica com es duu a terme aquesta eliminació. Cal tenir en compte que les iteracions de testeig no criden mai el procediment de *covering*.

3.4.6 Tria de l'acció

Un cop s'ha format el *match set*, el següent pas és decidir quina acció cal prendre, és a dir, prediure la sortida del multiplexor segons la mostra obtinguda. En les iteracions d'aprenentatge, aquesta tria es fa aleatòriament. En les iteracions de testeig, es crea el *prediction array*. La funció `PA_create`, definida a l'arxiu *xcs.c*, s'encarrega de fer-ho.

L'objectiu del *prediction array* és obtenir una estimació de quina recompensa s'obtindrà per a cadascuna de les accions possibles. Aquesta estimació és el que es guarda al *prediction array*, que té tants elements com accions possibles hi ha.

L'expressió matemàtica per a calcular el valor de cada element del *prediction array* és la següent:

$$PA[i] = \frac{\sum_{cl \in M} p_{cl} \cdot f_{cl}}{\sum_{cl \in M} f_{cl}} \tag{3.1}$$

on i és cadascuna de les accions, cl és un classificador que proposa l'acció i, i M representa el *match set*. Com es pot observar, a l'hora de calcular l'estimació de la recompensa només es tenen en compte els classificadors presents al *match set*.

Val la pena donar el codi font de la funció `PA_create`:

```
1  static inline void
2  PA_create (void)
3  {
4    PA_array[0] = PA_array[1] = 0;
5    FSA_array[0] = FSA_array[1] = 0;
6    for (size_t c = 0; c < match_set.ms_size; c++)
7      {
8        PA_array[match_set.ms_elements[c]->cl_action] +=
9          match_set.ms_elements[c]->cl_p * match_set.ms_elements[c]->cl_f;
10       FSA_array[match_set.ms_elements[c]->cl_action] +=
11         match_set.ms_elements[c]->cl_f;
12     }
13   if (FSA_array[0])
14     PA_array[0] /= FSA_array[0];
```

```
15    if (FSA_array[1])
16      PA_array[1] /= FSA_array[1];
17  }
```

Les variables globals no vistes fins al moment són:

`PA_array` Array de dos elements que contindrà el valor del numerador de l'expressió 3.1. Cada element té tipus `double`.

`FSA_array` Array de dos elements que contindrà el valor del denominador de l'expressió 3.1. Cada element té tipus `double`.

Les iteracions de testeig, un cop decidida l'acció a dur a terme, obtenen la recompensa de l'entorn i, si han encertat la seva predicció, incrementen el comptador d'encerts (que s'empra per a generar les dades estadístiques); amb això s'ha acabat la iteració. En canvi, les iteracions d'aprenentatge, en canvi, cal seguir amb els passos següents.

3.4.7 Creació de l'*action set*

Un cop s'ha decidit quina acció es durà a terme, es crea *l'action set*, que consisteix en els classificadors del *match set* que proposen l'acció seleccionada. És un procediment enormement senzill, dut a terme per la funció `AS_create` de l'arxiu *action_set.c*, el codi font de la qual és el següent:

```
1  void
2  AS_create (const int act)
3  {
4    action_set.as_size = 0;
5    action_set.as_microsize = 0;
6    for (size_t c = 0; c < match_set.ms_size; c++)
7      {
8        if (match_set.ms_elements[c]->cl_action == act)
9          {
10           action_set.as_elements[action_set.as_size] =
11             match_set.ms_elements[c];
12           action_set.as_size++;
13           action_set.as_microsize += match_set.ms_elements[c]->cl_num;
14         }
15     }
16 }
```

La variable `action_set` és una estructura de dades que conté els següents camps:

`as_size` Quantitat de macro-classificadors que conté l'*action set*. El tipus d'aquest camp és `size_t`.

as_microsize Quantitat de micro-classificadors que conté l'*action set*. El tipus d'aquest camp és `size_t`.

as_elements Array de punters a classificadors. En lloc de copiar els classificadors de la població a l'*action set*, es guarden punters als classificadors, de manera que es guanya en rendiment i en ús de memòria. Es considera el pitjor cas, és a dir, que tots els classificadors de la població pertanyin a l'*action set*.

3.4.8 Actualització dels atributs

A partir de la recompensa obtinguda de l'acció duta a terme, s'actualitzen els paràmetres dels classificadors que han entrat a l'*action set*. Les actualitzacions es fan seguint la filosofia del *Q-learning* (Sutton and Barto [1998]) (Widrow-Hoff Delta Rule).

Cada classificador actualitza el valor dels seus atributs de la següent manera:

1. El primer atribut a actualitzar és l'experiència, que s'incrementa en una unitat.
2. El segon paràmetre a actualitzar és p. Aquesta actualització segueix una de les dues expressions següents:
   a) Si l'experiència del classificador és inferior a β^{-1}, llavors segueix l'expressió següent:
   $$p = p + \frac{R - p}{exp} \tag{3.2}$$
   on R és la recompensa i exp és l'experiència del classificador.
   b) Si l'experiència del classificador és igual o superior a β^{-1}, llavors segueix l'expressió següent:
   $$p = p + \beta\,(R - p) \tag{3.3}$$
   on R és la recompensa.
3. El tercer paràmetre a actualitzar és ϵ. Aquesta actualització segueix una de les dues expressions següents:
   a) Si l'experiència del classificador és inferior a β^{-1}, llavors segueix l'expressió següent:
   $$\epsilon = \epsilon + \frac{|R - p| - \epsilon}{exp} \tag{3.4}$$

on R és la recompensa.

b) Si l'experiència del classificador és igual o superior a β^{-1}, llavors segueix l'expressió següent:

$$\epsilon = \epsilon + \beta\left(|R - p| - \epsilon\right) \tag{3.5}$$

on R és la recompensa.

4. El quart paràmetre a actualitzar és *as*. Aquesta actualització segueix una de les dues expressions següents:

a) Si l'experiència del classificador és inferior a β^{-1}, llavors segueix l'expressió següent:

$$as = as + \frac{\left(\sum_{cl \in A} num_{cl}\right) - as}{exp} \tag{3.6}$$

on exp és l'experiència del classificador i el sumatori representa la mida de l'*action set* en micro-classificadors.

b) Si l'experiència del classificador és igual o superior a β^{-1}, llavors segueix l'expressió següent:

$$as = as + \beta\left[\left(\sum_{cl \in A} num_{cl}\right) - as\right] \tag{3.7}$$

on el sumatori representa la mida de l'*action set* en micro-classificadors.

5. L'últim classificador a actualitzar és el *fitness* (f). El mecanisme d'actualització d'aquest paràmetre és especial i se serveix d'un array auxiliar, l'*accuracy vector* (representat per κ) i l'*accuracy sum*, un nombre real. Tant κ com l'*accuracy sum* s'inicialitzen a zero. Es fan dues passades als classificadors de l'*action set*:

a) A la primera passada es dóna valor a κ i a l'*accuracy sum*. Per donar valor a κ se segueix l'expressió següent:

$$\kappa\left(cl\right) = \begin{cases} 1 & \epsilon_{cl} < \epsilon_0 \\ \alpha\left(\dfrac{\epsilon_{cl}}{\epsilon_0}\right)^{-\nu} & \epsilon_{cl} \geq \epsilon_0 \end{cases} \tag{3.8}$$

on $\kappa\left(cl\right)$ és la posició del classificador cl dins de κ i ϵ_{cl} és el valor de l'atribut ϵ del classificador cl.

Per a donar valor a l'*accuracy sum* se segueix l'expressió següent:

$$AS = \sum_{cl \in A} \left(\kappa \left(cl \right) \cdot num_{cl} \right) \tag{3.9}$$

on AS és l'*accuracy sum*, $\kappa \left(cl \right)$ és la posició del classificador cl dins de κ i num_{cl} és la numerositat del classificador cl.

b) A la segona passada és on realment s'actualitza l'atribut f de cada classificador de l'*action set*, segons l'expressió següent:

$$f = f + \beta \left(\frac{\kappa \left(cl \right) \cdot num}{AS} - f \right) \tag{3.10}$$

on AS és l'*accuracy sum*, $\kappa \left(cl \right)$ és la posició del classificador cl dins de κ, f és el *fitness* del classificador que s'actualitza i num és la seva numerositat.

L'arxiu *action_set.c* conté dues funcions que s'encarregen d'actualitzar els paràmetres:

```
1  void
2  AS_update_params (const int reward)
3  {
4    for (size_t c = 0; c < action_set.as_size; c++)
5      {
6        action_set.as_elements[c]->cl_exp++;
7
8        if (action_set.as_elements[c]->cl_exp < (1.00 / PAR_BETA))
9          {
10           // update prediction p_cl
11           action_set.as_elements[c]->cl_p +=
12             (reward - action_set.as_elements[c]->cl_p) /
13             action_set.as_elements[c]->cl_exp;
14
15           // update prediction error epsilon_cl
16           action_set.as_elements[c]->cl_epsilon +=
17             ((xcsabs (reward, action_set.as_elements[c]->cl_p))
18              - action_set.as_elements[c]->cl_epsilon)
19             / action_set.as_elements[c]->cl_exp;
20
21           // update action set size estimate as_cl
22           action_set.as_elements[c]->cl_as +=
23             (action_set.as_microsize - action_set.as_elements[c]->cl_as)
24             / action_set.as_elements[c]->cl_exp;
25         }
26       else
27         {
```

```
28            // update prediction p_cl
29            action_set.as_elements[c]->cl_p +=
30              PAR_BETA * (reward - action_set.as_elements[c]->cl_p);
31
32            // update prediction error epsilon_cl
33            action_set.as_elements[c]->cl_epsilon +=
34              PAR_BETA * ((xcsabs (reward, action_set.as_elements[c]->cl_p))
35                          - action_set.as_elements[c]->cl_epsilon);
36
37            // update action set size estimate as_cl
38            action_set.as_elements[c]->cl_as +=
39              PAR_BETA * (action_set.as_microsize -
40                          action_set.as_elements[c]->cl_as);
41          }
42      }
43
44    update_fitness ();
45  }
```

```
1   static inline void
2   update_fitness (void)
3   {
4     size_t c;
5     double acc_sum = 0;
6
7     for (c = 0; c < action_set.as_size; c++)
8       {
9         acc_vector[c] = 0;
10        if (action_set.as_elements[c]->cl_epsilon < PAR_EPSILON_0)
11          acc_vector[c] = 1;
12        else
13          {
14            acc_vector[c] = PAR_ALPHA *
15              pow (action_set.as_elements[c]->cl_epsilon
16                   / PAR_EPSILON_0, -PAR_NU);
17          }
18        acc_sum += acc_vector[c] * action_set.as_elements[c]->cl_num;
19      }
20
21    for (c = 0; c < action_set.as_size; c++)
22      {
23        action_set.as_elements[c]->cl_f += PAR_BETA * (acc_vector[c]
24                                                       *
25                                                       action_set.as_elements
26                                                       [c]->cl_num /
27                                                       acc_sum -
28                                                       action_set.as_elements
29                                                       [c]->cl_f);
```

```
30      }
31  }
```

La funció `xcsabs` calcula el valor absolut d'una resta:

```
1  static inline double __attribute__ ((__always_inline__))
2  xcsabs (const double d1, const double d2)
3  {
4    return ((d1 - d2 < 0) ? (d2 - d1) : (d1 - d2));
5  }
```

3.4.9 L'algorisme genètic

Finalment, l'últim pas que queda en les iteracions d'aprenentatge és cridar, si cal, l'algorisme genètic. No es crida en totes les iteracions d'aprenentatge, sinó només en algunes, concretament en aquelles iteracions en què es compleixi l'expressió següent:

$$i - \frac{\sum_{cl \in A} (ts_{cl} \cdot num_{cl})}{\sum_{cl \in A} (num_{cl})} > \theta_{GA} \tag{3.11}$$

on i és el número d'iteració actual, A és l'*action set*, ts_{cl} és l'atribut ts del classificador i num_{cl} és la seva numerositat.

L'explicació detallada del funcionament de l'algorisme genètic es veurà a l'apartat següent.

3.5 L'algorisme genètic

L'algorisme genètic té per finalitat fer evolucionar la població, de manera que es crein nous classificadors a partir dels millors i el sistema classificador aumenti l'*accuracy*, és a dir, que aumenti el nivell d'encerts. L'esquema general de funcionament de l'algorisme genètic és el següent:

1. Es generen dos fills, mitjançant *roulette-wheel selection*, *tournament selection* o algun altre mecanisme. Per a aquest treball s'ha escollit *roulette-wheel selection*.

2. Amb una probabilitat igual a χ s'aplica l'operador de *crossover* sobre els dos fills i s'actualitzen alguns dels seus atributs.

3. S'aplica l'operador de mutació sobre els fills.

4. Si la subsumció està activada, s'aplica la subsumció. En cas contrari, els fills s'insereixen directament a la població (eliminant classificadors si és necessari).

3.5.1 Selecció dels fills

Per a generar els fills, s'escullen dos pares i es dupliquen, donant lloc al fills. Per a seleccionar els pares se segueix el procediment de *roulette-wheel selection*. Aquest procediment consisteix en els següents passos:

1. En una variable auxiliar es guarda la suma de l'atribut f (*fitness*) de tots els classificadors de l'*action set*.
2. Es multiplica la suma per un nombre aleatori en l'interval $[0, 1)$. El resultat s'anomena "punt d'elecció".
3. Es recorren tots els classificadors de l'*action set*, sumant el seu atribut f. Quan s'arriba al classificador que fa que la suma sigui superior al punt d'elecció, aquest classificador queda escollit com a pare.

Aquest procediment es repeteix dos cops, ja que s'escullen dos pares. Els fills són una còpia idèntica als pares, excepte en dos atributs:

- L'atribut exp s'inicialitza a zero.
- L'atribut num (numerositat) s'inicialitza a 1.

La funció `rws_select_offspring`, definida a l'arxiu *ga.c*, implementa aquest procediment:

```
1  static inline class_t *
2  rws_select_offspring (class_t * ptr)
3  {
4    double fitnessSum = 0;
5    double choicePoint = 0;
6    size_t i;
7    for (i = 0; i < action_set.as_size; i++)
8      fitnessSum += action_set.as_elements[i]->cl_f;
9    choicePoint = getDoubleNumber () * fitnessSum;
10   fitnessSum = 0;
11   i = -1;
12   while (choicePoint > fitnessSum)
13     {
14       i++;
15       fitnessSum += action_set.as_elements[i]->cl_f;
16     }
17   strcpy (ptr->cl_rule, action_set.as_elements[i]->cl_rule);
18   ptr->cl_action = action_set.as_elements[i]->cl_action;
19   ptr->cl_p = action_set.as_elements[i]->cl_p;
20   ptr->cl_epsilon = action_set.as_elements[i]->cl_epsilon;
21   ptr->cl_f = action_set.as_elements[i]->cl_f;
22   ptr->cl_exp = 0;
```

```
23    ptr->cl_ts = action_set.as_elements[i]->cl_ts;
24    ptr->cl_as = action_set.as_elements[i]->cl_as;
25    ptr->cl_num = 1;
26    return action_set.as_elements[i];
27  }
```

3.5.2 Operació d'encreuament

L'operador d'encreuament funciona de la següent manera:

1. S'escullen dos separadors, s_1 i s_2, de la següent manera: s_1 és el resultat de multiplicar la mida en símbols d'una mostra per un nombre aleatori; els decimals es trunquen. s_2 és el resultat de multiplicar la mida en símbols d'una mostra (incrementada la mida en una unitat) per un nombre aleatori; els decimals també es trunquen.
2. Si $s_1 > s_2$, llavors es fa un *swap* entre s_1 i s_2.
3. Si $s_1 = s_2$ llavors s'incrementa el valor de s_2.
4. Es prenen les regles dels dos fills, intercanviant els símbols que hi hagi entre s_1 i s_2.

Per exemple, si la mida d'una regla és de 37 símbols, $s_1 = 3$ i $s_2 = 20$, els símbols 3 a 20 de la regla del primer fill s'intercanvien amb els símbols 3 a 20 de la regla del segon fill.

La funció `crossover` duu a terme aquest operador:

```
1   static inline void
2   crossover (class_t * ptr1, class_t * ptr2)
3   {
4     int sep1 = getDoubleNumber () * n_bits;
5     int sep2 = getDoubleNumber () * n_bits + 1;
6     if (sep1 > sep2)
7       {
8         int aux = sep1;
9         sep1 = sep2;
10        sep2 = aux;
11      }
12    else if (sep1 == sep2)
13      sep2++;
14    for (int i = sep1; i < sep2; i++)
15      {
16        if (ptr1->cl_rule[i] != ptr2->cl_rule[i])
17          {
18            char aux = ptr1->cl_rule[i];
19            ptr1->cl_rule[i] = ptr2->cl_rule[i];
20            ptr2->cl_rule[i] = aux;
```

```
21            }
22        }
23    }
```

Quan es duu a terme l'operador d'encreuament, es modifiquen alguns atributs dels fills:

1. L'atribut p dels dos fills passa a ser la mitjana aritmètica del valor que tenien:

$$p_1 = p_2 = \frac{p_1 + p_2}{2} \tag{3.12}$$

   on p_1 és l'atribut p del primer fill i p_2 és l'atribut p del segon fill.

2. L'atribut ϵ dels dos fills passa a ser la quarta part de la mitjana aritmètica del valor que tenien:

$$\epsilon_1 = \epsilon_2 = \frac{1}{4} \cdot \frac{\epsilon_1 + \epsilon_2}{2} \tag{3.13}$$

   on ϵ_1 és l'atribut ϵ del primer fill i ϵ_2 és l'atribut ϵ del segon fill.

3. L'atribut f dels dos fills passa a valer l'expressió següent:

$$f_1 = f_2 = \frac{\frac{f_{p1}}{num_{p_1}} + \frac{f_{p_2}}{num_{p_2}}}{2} \tag{3.14}$$

   on f_{p_1} és l'atribut f del primer pare, f_{p_2} és l'atribut f del segon pare, num_{p_1} és la numerositat del primer pare i num_{p_2} és la numerositat del segon pare.

Tant si hi ha hagut encreuament com si no, l'atribut f dels dos fills es divideix entre 10.

3.5.3 Mutació

L'operador de mutació s'aplica individualment sobre cadascun dels dos fills. Per a cada símbol, amb una probabilitat μ, es duu a terme el següent:

1. Si el símbol és #, llavors aquest símbol passa a valdre el símbol corresponent de la mostra.

2. Si el símbol no és #, llavors aquest símbol passa a valdre #.

Amb una probabilitat μ s'inverteix l'acció associada al classificador (és a dir, que si era 0 passa a ser 1, i si era 1 passa a ser 0).

La funció `mutation` s'encarrega de dur a terme aquest operador:

```
1  static inline void
2  mutation (class_t * ptr)
3  {
4    for (int i = 0; i < n_bits; i++)
5      {
6        if (getDoubleNumber () < PAR_MU)
7          {
8            if (ptr->cl_rule[i] == '#')
9              ptr->cl_rule[i] = sigma[i];
10           else
11             ptr->cl_rule[i] = '#';
12         }
13       if (getDoubleNumber () < PAR_MU)
14         ptr->cl_action = (ptr->cl_action) ? 0 : 1;
15     }
16 }
```

3.5.4 Subsumció

La subsumció és un mecanisme que analitza si un fill aporta valor a la població. Es comprova si hi ha un classificador, suficientment experimentat, la regla del qual sigui més general que la del fill. En cas afirmatiu, s'incrementa la numerositat d'aquest classificador, i no s'insereix el fill a la població, ja que tot el que pot aportar ja ho aporta el classificador més general.

El paràmetre `PAR_DO_GAS` (definit a l'arxiu *config.h*) controla si es durà a terme la subsumció o no; cada cop que es modifica el valor del paràmetre cal recompilar tota l'aplicació. En cas que no hi hagi subsumció, els dos fills s'insereixen directament a la població.

Per tal que un classificador c pugui subsumir un classificador f cal que es donin les següents condicions:

1. L'acció proposada per c i per f ha de ser la mateixa.
2. L'experiència (atribut exp) del classificador c ha de ser superior al paràmetre θ_{sub}.
3. L'atribut ϵ del classificador c ha de ser inferior al paràmetre ϵ_0.
4. La regla del classificador c ha de ser més general que la regla del classificador f. És a dir, que per a cada símbol de la regla de f, el símbol de la regla de c és igual o bé és `#`.

La funció `does_subsume` comprova si un classificador pot subsumir un altre classificador:

```
static inline int
does_subsume (const class_t * small, const class_t * big)
{
  if (big->cl_action == small->cl_action && big->cl_exp > PAR_THETA_SUB
      && big->cl_epsilon < PAR_EPSILON_0)
    {
      for (int i = 0; i < n_bits; i++)
        {
          if (big->cl_rule[i] != '#' && big->cl_rule[i] != small->cl_rule[i])
            return 0;
        }
      return 1;
    }
  return 0;
}
```

La subsumció d'un classificador es realitza en aquest ordre:

1. Primer es comprova si algun dels dos pares pot subsumir el classificador (que sempre serà un fill).
2. Si cap pare no pot subsumir el fill, es busca si algun classificador de l'*action set* pot subsumir-lo. Si hi ha més d'un candidat, se n'escull un de forma aleatòria.
3. Si a l'*action set* no hi ha cap candidat, se'n busca un a la població. Si hi ha més d'un candidat, se n'escull un de forma aleatòria.
4. Si a la població tampoc hi ha cap candidat, llavors el fill s'insereix directament a la població.

```
static inline void
subsume (const class_t * child, class_t * parent1, class_t * parent2)
{
  if (does_subsume (child, parent1))
    {
      parent1->cl_num++;
      micro_popsize++;
      return;
    }
  else if (does_subsume (child, parent2))
    {
      parent2->cl_num++;
      micro_popsize++;
      return;
```

```
15      }
16    else
17      {
18        if (subsume_action_set (child))
19          return;
20        else if (subsume_population (child))
21          return;
22      }
23    insert_classifier (child);
24  }
```

```
1   static inline int
2   subsume_action_set (const class_t * child)
3   {
4     size_t e, q = 0;
5     size_t candidates[action_set.as_size];
6
7     for (size_t s = 0; s < action_set.as_size; s++)
8       {
9         if (does_subsume (child, action_set.as_elements[s]))
10          {
11            candidates[q] = s;
12            q++;
13          }
14      }
15    if (q == 0)
16      return 0;
17    e = q * getDoubleNumber ();
18    action_set.as_elements[candidates[e]]->cl_num++;
19    micro_popsize++;
20    return 1;
21  }
```

```
1   static inline int
2   subsume_population (const class_t * child)
3   {
4     size_t e, q = 0;
5     size_t candidates[macro_popsize];
6
7     for (size_t s = 0; s < macro_popsize; s++)
8       {
9         if (does_subsume (child, population + s))
10          {
11            candidates[q] = s;
12            q++;
13          }
14      }
15    if (q == 0)
16      return 0;
```

```
17    e = q * getDoubleNumber ();
18    population[candidates[e]].cl_num++;
19    micro_popsize++;
20    return 1;
21  }
```

3.6 Procediments auxiliars

3.6.1 Eliminar un classificador

L'eliminació d'un classificador és un procediment complex, que implica realitzar uns càlculs per tal de decidir quin classificador s'eliminarà; si el procediment es crida des del *covering* llavors cal comprovar també si el classificador eliminat pertanyia a l'*action set*. El procediment és el següent:

1. Es calcula la mitjana del *fitness* de la població, segons l'expressió següent:

$$f_\mu = \frac{\sum_{cl \in P} f_{cl}}{\sum_{cl \in P} num_{cl}} \tag{3.15}$$

   on P és la població, f_{cl} és el *fitness* del classificador i num_{cl} és la seva numerositat.

2. Es fa una passada per la població, generant l'array de vots i la suma de vots. Cada element de l'array és el resultat de sumar el valor de l'element anteior de l'array (zero en el cas del primer element) amb el *deletion vote* del classificador corresponent. La suma de vots és el valor de l'últim element de l'array de vots.

3. Es multiplica la suma de vots per un nombre aleatori en l'interval $[0, 1)$, aquest valor s'anomena *punt d'elecció*.

4. Es fa una passada sobre l'array de vots. En el moment que es troba un valor superior al punt d'elecció, s'elimina el classificador corresponent a aquell element de l'array de vots.

```
1  void
2  delete_from_population (const int ms_flag)
3  {
4    double av_fitness = 0;
5    double vote_sum = 0;
6    double choice_point = 0;
7    size_t i;
8
```

```
9     for (i = 0; i < macro_popsize; i++)
10      av_fitness += population[i].cl_f;
11    av_fitness /= micro_popsize;
12    for (i = 0; i < macro_popsize; i++)
13      {
14        vote_sum += deletion_vote (population + i, av_fitness);
15        vote_array[i] = vote_sum;
16      }
17    choice_point = vote_sum * getDoubleNumber ();
18    for (i = 0; i < macro_popsize; i++)
19      {
20        if (vote_array[i] > choice_point)
21          {
22            delete_classifier (i, ms_flag);
23            return;
24          }
25      }
26  }
```

Aquesta funció està definida a l'arxiu *deletion.c*, ja que la funció es crida tant des de l'algorisme genètic com des d'altres parts del sistema classificador (per exemple, des del *covering*). La funció següent és l'encarregada d'eliminar el classificador de la població (i, si cal, del *match set*):

```
1   static inline void
2   delete_classifier (const int idx, const int ms_flag)
3   {
4     class_t *src = population + macro_popsize - 1;
5     class_t *dst = population + idx;
6     size_t i;
7     if (dst->cl_num > 1)
8       {
9         // the classifier has a numerosity greater than one,
10        // it is only necessary to decrement that classifier
11        dst->cl_num--;
12        micro_popsize--;
13      }
14    else
15      {
16        // the whole classifier must be deleted
17        macro_popsize--;
18        micro_popsize--;
19        if (idx != (int) macro_popsize)
20          {
21            // the classifier to be deleted is not the last one
22            strcpy (dst->cl_rule, src->cl_rule);
23            dst->cl_action = src->cl_action;
24            dst->cl_p = src->cl_p;
```

```
25            dst->cl_epsilon = src->cl_epsilon;
26            dst->cl_f = src->cl_f;
27            dst->cl_exp = src->cl_exp;
28            dst->cl_ts = src->cl_ts;
29            dst->cl_as = src->cl_as;
30            dst->cl_num = src->cl_num;
31          }
32        // the classifier no longer belongs to the population
33        if (ms_flag)
34          {
35            // the match set must be examined
36            for (i = 0; i < match_set.ms_size; i++)
37              if (match_set.ms_elements[i] == dst)
38                break;
39            if (i < match_set.ms_size)
40              {
41                // that classifier must be removed from the match set
42                match_set.ms_actions[dst->cl_action]--;
43                match_set.ms_elements[i] =
44                  match_set.ms_elements[match_set.ms_size - 1];
45                match_set.ms_size--;
46              }
47          }
48      }
49  }
```

Cal tenir en compte que quan es crida el procediment d'eliminació d'un classificador des del *covering*, cal eliminar el classificador no només de la població sinó també del *match set*, si el classificador eliminat en formava part.

Finalment, queda per veure com es calcula el *deletion vote*. Per norma general, el seu valor ve definit per l'expressió següent:

$$v_{cl} = as_{cl} \cdot num_{cl} \tag{3.16}$$

És a dir, el *deletion vote* d'un classificador és el resultat de multiplicar l'atribut *as* del classificador per la seva numerositat. Si es compleix la condició següent:

$$(exp_{cl} > \theta_{del}) \wedge \left(\frac{f_{cl}}{num_{cl}} < \delta \cdot f_{\mu} \right) \tag{3.17}$$

on exp_{cl} és l'atribut exp del classificador, f és el seu *fitness*, num_{cl} la seva numerositat i f_{μ} és la mitjana del *fitness* de la població (calculada a l'expressió 3.15), llavors:

$$v_{cl} = \frac{as_{cl} \cdot num_{cl} \cdot f_{\mu}}{\frac{f_{cl}}{num_{cl}}} \tag{3.18}$$

```
static inline double
deletion_vote (const class_t * ptr, const double av_fitness)
{
  double vote = ptr->cl_as * ptr->cl_num;
  if (ptr->cl_exp > PAR_THETA_DEL
      && (ptr->cl_f / ptr->cl_num) > (PAR_DELTA * av_fitness))
    vote *= av_fitness / (ptr->cl_f / ptr->cl_num);
  return vote;
}
```

3.6.2 Inserir un classificador

Quan es vol inserir un classificador a la població cal comprovar abans que no estigui repetit. Es considera que un classificador està repetit si coincideix la regla i l'acció (no cal que també coincideixin els atributs). Si el classificador a inserir ja existeix, se n'augmenta la numerositat. En cas contrari, s'insereix a la població i s'augmenten convenientment els comptadors de macro-classificadors i micro-classificadors.

```
static inline void
insert_classifier (const class_t * ptr)
{
  for (size_t i = 0; i < macro_popsize; i++)
    {
      if (ptr->cl_action == population[i].cl_action &&
          strcmp (ptr->cl_rule, population[i].cl_rule) == 0)
        {
          population[i].cl_num++;
          micro_popsize++;
          return;
        }
    }
  strcpy (population[macro_popsize].cl_rule, ptr->cl_rule);
  population[macro_popsize].cl_action = ptr->cl_action;
  population[macro_popsize].cl_p = ptr->cl_p;
  population[macro_popsize].cl_epsilon = ptr->cl_epsilon;
  population[macro_popsize].cl_f = ptr->cl_f;
  population[macro_popsize].cl_exp = ptr->cl_exp;
  population[macro_popsize].cl_ts = ptr->cl_ts;
  population[macro_popsize].cl_as = ptr->cl_as;
  population[macro_popsize].cl_num = ptr->cl_num;
  micro_popsize += ptr->cl_num;
```

Paràmetre	mux-6	mux-11	mux-20
$N \times 1000$	0.5	1	5
milers d'iter.	5	20	100
Window	50	50	100
$P_{\#}$	0.5	0.5	0.5
Experiments	25	25	25

Taula 3.4: Paràmetres emprats a les proves de funcionament

```
24    macro_popsize++;
25  }
```

Aquesta funció està definida a l'arxiu *ga.c*, ja que la funció es crida des de l'algorisme genètic. La inserció dels classificadors creats per *covering* no es fa mitjançant aquesta funció, sinó que es creen directament sobre la població, havent fet (si cal) espai mitjançant l'eliminació d'un classificador.

3.7 Proves de funcionament

Aquest apartat té per objectiu descriure quines proves s'han fet per tal de verificar el correcte funcionament de la implementació base del sistema classificador. Les proves han consistit en executar el problema del multiplexor en quatre versions diferents: el multiplexor de 6, 11, 20 i 37 bits (símbols). Les gràfiques corresponents són les de les figures 3.1, 3.2 i 3.3. En color vermell s'aprecia l'evolució de l'*accuracy*: com més alt, més bona és la predicció feta pel sistema classificador (és a dir, la qualitat de l'aprenentatge). En color verd s'aprecia l'evolució de la població, expressada en quants macroclassificadors hi ha respecte el màxim total de micro-classificadors. Tal i com es pot observar a les figures, la implementació base aconsegueix resoldre tots els problemes plantejats. Els paràmetres emprats són els mateixos de la taula 3.1; la taula 3.4 mostra els paràmetres que han variat segons cada versió del multiplexor.

3.8 Proves de rendiment

Un cop feta i descrita la implementació de l'XCS, arriba el moment d'analitzar-ne el seu rendiment per tal de buscar oportunitats d'optimització. A tal fi, s'executaran proves amb els multiplexors de 6, 11 i 20 bits. Al compilar el codi font, s'indicarà al compilador que insereixi codi d'instrumentació per tal que, posteriorment, amb l'ajuda del *profiler*, es puguin identificar les parts del codi que més temps consumeixen. Per tal de no interferir amb la instrumentació, no s'han activat les opcions d'optimització del compilador.

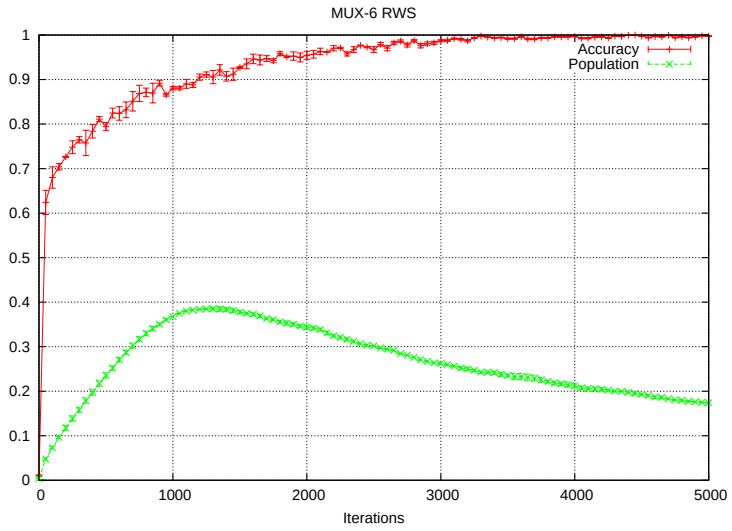

Figura 3.1: Multiplexor de 6 bits

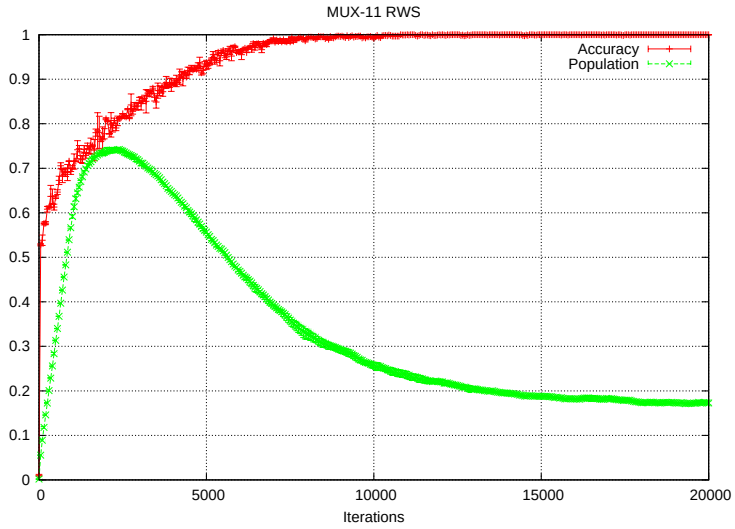

Figura 3.2: Multiplexor d'11 bits

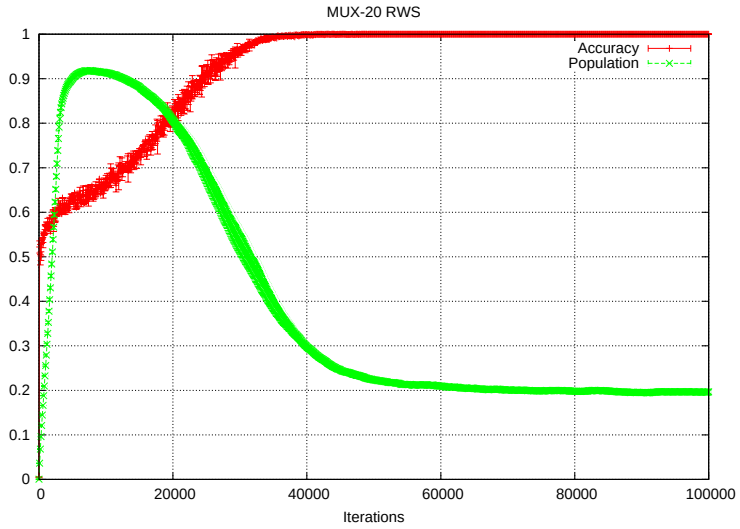

Figura 3.3: Multiplexor de 20 bits

Al cap i a la fi, no es tracta ara de saber com de ràpid és la present implementació, sinó buscar quina part del codi val la pena esforçar-se a optimitzar.

Les opcions del compilador activades són les següents:

`-std=gnu99` Indica al compilador que ha de seguir l'estàndar ISO 9899:1999 ("C99") amb les extensions pròpies de GNU C.

`-Wall` Indica al compilador que sigui mostri tants *warnings* com sigui possible.

`-Wextra` Indica al compilador que mostri encara més *warnings* que amb l'opció anterior.

`-ggdb` Indica al compilador que generi símbols de depuració per al *debugger* `gdb`.

`-pg` Indica al compilador que insereixi codi per a la mesura del temps d'execució, de manera que es pugui emprar l'eina `gprof` per a obtenir dades sobre el temps que consumeix cada funció.

`-pipe` Indica al compilador que faci servir *pipes* en lloc de fitxers temporals.

Funció	Experiments		
	1	*5*	*25*
`MS_create`	100.05	70.87	64.79
`deletion_vote`	0	8.34	9.02
`does_subsume`	0	4.17	4.92
`delete_from_population`	0	4.17	8.20

Taula 3.5: Resultats per a les proves del *profiling* (MUX-6)

Funció	Experiments		
	1	*5*	*25*
`MS_create`	42.40	59.33	63.68
`deletion_vote`	18.65	13.95	11.34
`delete_from_population`	7.42	4.17	6.58
`does_subsume`	7.63	5.28	5.16

Taula 3.6: Resultats per a les proves del *profiling* (MUX-11)

L'opció `-lm` indica al compilador que enllaci contra la llibreria matemàtica, ja que es fa servir una de les funcions que proveeix.

Les compilacions i execucions s'ha fet en una màquina amb les següents característiques principals:

CPU Intel Core 2 Quad Q9300 2.5 GHz

Memòria 4 GB

Cache 128 KB de primer nivell, 6 MB de segon nivell

S.O. Linux 2.6.33.4 (Slackware 13.1)

Compilador `gcc` 4.5.0

Glibc Versió 2.11.1

3.8.1 Resultats del *profiling*

Amb els paràmetres de la taula 3.4 s'han dut a terme execucions del multiplexor per tal d'analitzar quines parts del sistema classificador són les que, proporcionalment, més temps consumeixen.

Les taules 3.5, 3.6 i 3.7 no deixen lloc a cap mena de dubte: la creació el *match set* és, amb diferència, la part més costosa de tot el sistema classificador XCS. Per a cada tipus de problema (les diferents versions del multiplexor), les taules mencionades mostren el percentatge de temps dedicat per a

Funció	Experiments		
	1	*5*	*25*
`MS_create`	59.20	57.75	57.95
`deletion_vote`	12.24	13.10	13.12
`delete_from_population`	9.33	4.17	9.10
`does_subsume`	7.31	8.31	7.37

Taula 3.7: Resultats per a les proves del *profiling* (MUX-20)

cadascuna de les funcions indicades, segons es faci 1, 5 o 25 experiments. La conclusió és clara: és la creació del *match set* la que ha de rebre els esforços d'optimització.

3.8.2 Conclusions

Les dades ofereixen una conclusió clarísima: més de la meitat del temps d'execució de l'XCS es gasta en formar el *match set*. Per tant caldrà buscar maneres per tal d'optimitzar-lo. En aquest punt val la pena recuperar el codi de la funció `MS_create`:

```
1  void
2  MS_create (void)
3  {
4    size_t c, i;
5    match_set.ms_size = 0;
6    match_set.ms_actions[0] = match_set.ms_actions[1] = 0;
7
8    for (c = 0; c < macro_popsize; c++)
9      {
10       for (i = 0; (int) i < n_bits; i++)
11         if (sigma[i] != population[c].cl_rule[i] &&
12             population[c].cl_rule[i] != '#')
13           break;
14       if ((int) i == n_bits)
15         {
16           match_set.ms_actions[population[c].cl_action]++;
17           match_set.ms_elements[match_set.ms_size] = &population[c];
18           match_set.ms_size++;
19         }
20     }
21 }
```

Quina part del codi pot ser la que més temps consumeixi?

- La declaració de les variables no té cap cost temporal en temps d'execució, a menys que el sistema operatiu es vegi obligat a ampliar l'espai reservat per al segment de pila.
- A continuació hi ha tres assignacions, que tenen un cost temporal fix.
- Finalment hi ha un bucle, que s'executa tantes vegades com macro-classificadors hi hagi a la població.
- Dins del bucle trobem un altre bucle, que s'executa tants cops com símbols tingui una mostra de l'entorn.
- El bucle interior conté dues branques `if`.

La complexitat d'aquesta funció ve donada, clarament, pels dos bucles, de manera que si P és la mida de la població en macro-classificadors i n és la mida en símbols d'una mostra de l'entorn, la complexitat de la funció és $O(n \cdot P)$. Què es pot fer davant d'aquesta complexitat?

1. No es pot reduir la complexitat, ja que la formació del *match set* consisteix justament en aquests dos bucles: per a cada macro-classificador es comprova si la regla fa *match* amb la mostra, i per veure si la regla fa *match* cal recórrer tots els símbols de la regla i comparar-los amb els de la mostra.
2. No es pot reduir la mida de la població, ja que llavors s'estaria reduint l'eficàcia del sistema classificador. Per dir-ho d'alguna manera, si en lloc de recórrer tots els classificadors només se'n recorressin alguns, l'*accuracy* quedaria greument reduït.
3. Tampoc es pot reduir la mida de la mostra, ja que per a determinar si la regla d'un classificador fa *match* amb la mostra cal analitzar tots els símbols.

   Estrictament parlant, no sempre cal comparar *tots* els símbols de la regla, ja que tan aviat com se'n troba un que no fa *match*, ja es pot aturar la comparació i determinar que el classificador no fa *match* amb la mostra. En el pitjor dels casos, però, caldrà comparar tots els símbols.

Abans de donar-ho tot per perdut, convé fixar-se en un aspecte del bucle intern. Es pot observar fàcilment que cada iteració comprova un sol símbol. Cada símbol està codificat en un caràcter. Tenint en compte que les regles treballen amb un alfabet ternari (requeriria dos bits per a codificar cada símbol) i que un caràcter ocupa vuit bits, s'està desperdiciant el 25% de la capacitat d'emmagatzematge.

La clau de la solució consisteix en intentar que cada iteració del bucle intern comprovi més d'un símbol a la vegada, d'aquesta manera se'n redueix el nombre d'iteracions. A tal fi, caldrà pensar en una nova representació dels símbols de la mostra i de la regla d'un classificador.

Capítol 4

Optimització de la representació

L'objectiu d'aquest capítol és presentar i explicar la primera optimització que s'ha fet del sistema classificador XCS. Es començarà parlant dels fonaments teòrics de la tècnica d'optimització, com ha canviat la present implementació, quin és el seu rendiment i com ha millorat respecte la implementació base.

4.1 Fonaments teòrics

En la implementació base, la codificació de les regles i de les mostres de l'entorn es feia amb arrays de caràcters, on per a cada símbol s'emprava un caràcter. Tenint en compte que un caràcter pot guardar 256 símbols diferents, i que una regla pot tenir tres símbols diferents, és obvi que s'està desperdiciant molt d'espai a memòria. Tanmateix, no és aquest el principal problema.

Tal com s'ha vist a l'apartat 3.8.1, la formació del *match set* és la part que més temps consumeix d'una iteració. El procés de *matching* consisteix en una comparació de la mostra amb la regla d'un classificador. És igual en quin ordre es comparin els símbols, per tant com més símbols es comparin per cicle de rellotge, més ràpid serà el procés de *matching*. El problema de malgastar espai a memòria és que aquest fet impedeix que es comparin diversos símbols mitjançant una sola instrucció. Si es compara caràcter a caràcter, una instrucció de comparació només pot comparar un sol símbol. Si en lloc d'emprar un caràcter (8 bits) per símbol s'empren 2 bits, podrem

b_1	b_0	**símbol**
0	0	#
0	1	0
1	0	1
1	1	#

Taula 4.1: Nova representació binària

comparar 4 símbols per instrucció, obtenint (en números rodons) un augment de quatre cops en la velocitat del procés de *matching*. També cal tenir en compte que el fet de reduir l'ús de memòria fa més eficient l'ús de la *cache*, la qual cosa també pot comportar un aument en la velocitat. Finalment, com que els processadors actuals disposen d'instruccions per a comparar operands de 32 bits (o fins i tot 64 bits) sense sobrecost afegit, es poden arribar a comparar 16 o 32 símbols simultàniament.

4.2 La nova representació

Per a poder emprar mapes de bits cal decidir una representació per a cada símbol de les regles i de les mostres de l'entorn. Les mostres es codifiquen en l'alfabet binari, mentre que les regles es codifiquen en l'alfabet $\{0, 1, \#\}$. Per tant cal emprar dos bits per símbol, segons la taula 4.1. A la pràctica, el símbol # es codificarà només amb la seqüència 11.

Amb aquesta nova representació, el procediment de *matching* canvia lleugerament. Amb la representació amb caràcters es considerava que una regla feia *match* amb la mostra si per a cada símbol de la regla es complia, almenys, una de les dues condicions següents:

1. El símbol de la regla és idèntic al símbol de la mostra.
2. El símbol de la regla és #.

Amb la nova representació, hi ha una sola condició: es fa l'operació AND binària entre els símbols de la regla i els de la mostra, i el resultat ha de coincidir amb la mostra. Aquest procediment és equivalent a l'anterior:

1. En cas que el símbol de la mostra sigui 0 (representat pels bits 01), només quedarà igual si es fa l'AND binària amb els bits 01 (símbol 0) o amb els bits 11 (símbol #).
2. En el cas que el símbol de la mostra sigui 1 (representat pels bits 10), només quedarà igual si es fa l'AND binària amb els bits 10 (símbol 1) o amb els bits 11 (símbol #).

Hi ha un detall que cal tenir en compte. Cada byte permet codificar 4 símbols. Si considerem que l'ALU d'un processador (extensions vectorials a banda) pot treballar amb operands de 64 bits, això resulta en una codificació de 32 símbols que poden ser operats a la vegada. Per cada enter (de 64 bits) extra, tenim 32 símbols addicionals. Ara bé, el problema del multiplexor treballa amb mostres de 6, 11, 20, 37, 70… bits. Per a posar un exemple es prendrà el cas del multiplexor de 20 bits. Queden 12 símbols lliures: amb quin valor convé omplir-los? Això depèn de quines operacions a nivell de bit s'emprim per al procediment de *matching*. Tal com s'ha vist anteriorment, es fa una operació d'AND binària i es compara amb la mostra. Si els bits de la mostra són zeros, qualsevol operació d'AND binària generarà el mateix resultat, la comparació serà certa i farà *match*.

4.3 Impacte en la implementació

Què cal canviar en la implementació per tal d'aplicar la nova representació?

1. La representació de les mostres i de les regles dels classificadors ha de canviar, ja que no seran un array de caràcters sinó un array d'enters — i no hi ha correspondència entre un símbol per enter, com sí hi havia correspondència entre un símbol per caràcter.

2. L'entorn: la implementació del multiplexor ha de canviar, ja que les mostres es codifiquen segons la nova representació.

3. El procés de *matching*: justament el *quid* de la qüestió. Les comparacions ja no seran entre caràcters, sinó entre enters. Els bucles faran menys iteracions.

4. El procés de *covering*: la creació de nous classificadors també es veurà afectada, ja que la representació de les regles ha canviat.

5. L'algorisme genètic: tots els operadors que actuen amb la regla (encreuament i mutació) també se'n veuran afectats.

4.3.1 La representació dels classificadors

En la implementació base, cada classificador és un `struct` on hi ha un punter a caràcter que apunta a una zona de memòria que conté la regla. Aquesta zona de memòria es crea abans d'executar els experiments, i no s'allibera fins que tots els experiments s'han dut a terme. Amb la nova representació cal substituir el punter a caràcter per un punter a enter de 64 bits.

```
1 typedef struct
2 {
```

```
3     uint64_t *cl_rule;
4     int cl_action;
5     double cl_p;
6     double cl_epsilon;
7     double cl_f;
8     size_t cl_exp;
9     size_t cl_ts;
10    double cl_as;
11    size_t cl_num;
12  } class_t;
```

4.3.2 El multiplexor

Els canvis en el multiplexor es deuen al canvi de la representació de la mostra. En concret canvien les funcions següents:

Inicialització La variable global `k_bits` conté el valor del paràmetre k en la funció del multiplexor, segons la coneguda expressió següent:

$$n = k + 2^k \tag{4.1}$$

La variable global `n_bits` conté el valor de n. En la implementació base, la variable global `str` era un array de $n+1$ caràcters que contenia la mostra. En la nova implementació cada símbol es representa amb dos bits, per tant es calcula quants enters de 64 bits calen per a representar tota la mostra, segons l'expressió següent:

$$s = \left\lceil \frac{2n}{64} \right\rceil \tag{4.2}$$

Exemples: els multiplexors de 6, 11 i 20 bits (símbols) empren un sol enters de 64 bits, ja que necessiten tan sols 12, 22 i 40 bits per a representar els seus símbols. El multiplexor de 37 bits (símbols) necessita dos enters, ja que empra 74 bits; el multiplexor de 70 bits (símbols) necessita 3 enters per a encabre-hi 140 bits.

Tenint en compte que el multiplexor treballa en l'alfabet $\{0, 1\}$ hom es podria preguntar per què s'empren dos bits per símbol. El motiu és senzill: els símbols de la mostra es comparen amb els símbols de la regla, i aquests últims treballen amb un alfabet ternari, cal que tant les mostres com les regles emprin la mateixa codificació. Les regles necessiten almenys dos bits per símbol, per tant el multiplexor també.

El codi font de la inicialització queda de la següent manera:

```
1  void
2  MUX_init (const int _k_bits)
3  {
4    k_bits = _k_bits;
```

```
5    real_bits = n_bits = k_bits + (1 << k_bits);
6    size = n_bits / 32;
7    if (n_bits % 32)
8      size++;
9    sigma = calloc (size, sizeof (uint64_t));
10   assert (sigma);
11 }
```

Obtenció d'una mostra L'obtenció d'una mostra no es beneficia del canvi de representació — de fet, en resulta perjudicada: només cal comparar el codi font d'aquesta funció amb el de la implementació base. Les variables `sym_zero` i `sym_one` contenen les codificacions binàries pels als símbols zero i u, respectivament, segons la seva posició dins de la mostra.

```
1  void
2  MUX_get_value (void)
3  {
4    int s = 0, c = 0, i;
5    bzero (sigma, size * sizeof (uint64_t));
6    for (i = 0; i < n_bits; i++)
7      {
8        if (getSingleBinaryDigit ())
9     sigma[s] |= sym_zero[i % 32];
10       else
11    sigma[s] |= sym_one[i % 32];
12       c++;
13       if (c == 32)
14    {
15      c = 0;
16      s++;
17    }
18     }
19 }
```

Obtenció de la recompensa Per a obtenir la recompensa a partir d'una mostra i la predicció feta pel sistema classificador es descodifica la mostra convertint-la a una representació basada en caràcters, opcionalment es comprova la integritat d'aquesta nova representació i es calcula la recompensa segons la predicció hagi estat encertada o equivocada.

```
1  int
2  MUX_get_reward (const char act)
3  {
4    char aux[n_bits + 1];
5    bitmap_to_string (aux, sigma);
6    assert (check_string_sigma (aux));
7    return MUX_get_reward_old (aux, act);
8  }
9
10 static int
```

```
MUX_get_reward_old (const char *aux, const char act)
{
  int i, j;
  int v = 0;

  for (i = 0, j = k_bits - 1; i < k_bits; i++, j--)
    v += (1 << j) * (aux[i] - '0');
  return (act == aux[k_bits + v]) ? 1000 : 0;
}
```

4.3.3 El procés de *matching*

En el procés de *matching* cal aplicar el criteri explicat al principi d'aquest capítol (apartat 4.2). El codi font queda de la següent manera:

```
void
MS_create (void)
{
  size_t c;
  int i;
  match_set.ms_size = 0;
  match_set.ms_actions[0] = match_set.ms_actions[1] = 0;

  for (c = 0; c < macro_popsize; c++)
    {
      for (i = 0; i < size; i++)
   if ((sigma[i] & population[c].cl_rule[i]) != sigma[i])
     break;
      if (i == size)
   {
     match_set.ms_actions[population[c].cl_action]++;
     match_set.ms_elements[match_set.ms_size] = &population[c];
     match_set.ms_size++;
   }
    }
}
```

La creació del *match set* és l'única part de l'XCS que es beneficia dràsticament de la nova representació.

4.3.4 El procés de *covering*

El procés de *covering* es veu modificat pel fet que pot alterar la regla del nou classificador. El codi font queda de la següent manera:

```
void
MS_covering (const int curr_iter)
{
  int c = 0, s = 0, i;

  bcopy (sigma, population[macro_popsize].cl_rule, size * sizeof (uint64_t));
```

```
7     for (i = 0; i < n_bits; i++)
8       {
9         if (getDoubleNumber () < PAR_P_SHARP)
10     population[macro_popsize].cl_rule[s] |= mask[i % 32];
11        c++;
12        if (c == 32)
13     {
14       c = 0;
15       s++;
16     }
17      }
18    population[macro_popsize].cl_p = PAR_P_I;
19    population[macro_popsize].cl_epsilon = PAR_EPSILON_I;
20    population[macro_popsize].cl_f = PAR_F_I;
21    population[macro_popsize].cl_exp = 0;
22    population[macro_popsize].cl_ts = curr_iter;
23    population[macro_popsize].cl_as = 1;
24    population[macro_popsize].cl_num = 1;
25    population[macro_popsize].cl_action =
26      (match_set.ms_actions[0]) ?
27      (match_set.ms_actions[1]++, 1) : (match_set.ms_actions[0]++, 0);
28    match_set.ms_elements[match_set.ms_size] = &population[macro_popsize];
29    match_set.ms_size++;
30    macro_popsize++;
31    micro_popsize++;
32  }
```

Com que es pot alterar la regla en alguns dels seus símbols, cal crear símbol per símbol, per la qual cosa el procés de *covering* no es beneficia del canvi de representació.

4.3.5 L'algorisme genètic

L'algorisme genètic conté diverses rutines que treballen amb les regles dels classificadors. Desgraciadament, totes aquestes rutines necessiten operar símbol per símbol, com es veurà a continuació.

Operador d'encreuament Aquest operador pren dues regles i intercanvia alguns dels símbols de les seves regles:

```
1   static inline void
2   crossover (class_t * ptr1, class_t * ptr2)
3   {
4     int sep1 = getDoubleNumber () * n_bits;
5     int sep2 = getDoubleNumber () * n_bits + 1;
6     if (sep1 > sep2)
7       {
8         int aux = sep1;
9         sep1 = sep2;
10        sep2 = aux;
```

```
11      }
12    else if (sep1 == sep2)
13      sep2++;
14    for (int i = sep1; i < sep2; i++)
15      {
16        if ((ptr1->cl_rule[i / 32] & mask[i % 32]) !=
17       (ptr2->cl_rule[i / 32] & mask[i % 32]))
18     {
19       binary_swap (ptr1->cl_rule, ptr2->cl_rule, i);
20     }
21      }
22  }
23
24  static inline void //__attribute__ ((__always_inline__))
25  binary_swap (uint64_t * s1, uint64_t * s2, const int c)
26  {
27    uint64_t tmp1 = s1[c / 32] & mask[c % 32];
28    uint64_t tmp2 = s2[c / 32] & mask[c % 32];
29    uint64_t aux = tmp1;
30    tmp1 = tmp2;
31    tmp2 = aux;
32    s1[c / 32] &= ~mask[c % 32];
33    s2[c / 32] &= ~mask[c % 32];
34    s1[c / 32] |= tmp1;
35    s2[c / 32] |= tmp2;
36  }
```

Com que l'operador no treballa amb tota la regla sencera, i a més a més pot modificar individualment alguns dels seus símbols, no es pot beneficiar de la nova representació — de fet, se'n pot veure perjudicat.

Operador de mutació Aquest operador pren un classificador i altera els seus símbols (i l'acció) amb una probabilitat μ. Per tant, doncs, ha d'operar a nivell de símbol, amb la qual cosa no es pot beneficiar de la nova representació.

```
1   static inline void
2   mutation (class_t * ptr)
3   {
4     for (int i = 0; i < n_bits; i++)
5       {
6         if (getDoubleNumber () < PAR_MU)
7      {
8        if ((ptr->cl_rule[i / 32] & mask[i % 32]) == mask[i % 32])
9          {
10           ptr->cl_rule[i / 32] &= ~mask[i % 32];
11           ptr->cl_rule[i / 32] |= (sigma[i / 32] & mask[i % 32]);
12         }
13       else
14         ptr->cl_rule[i / 32] |= mask[i % 32];
15     }
```

```
16      }
17    if (getDoubleNumber () < PAR_MU)
18      ptr->cl_action = (ptr->cl_action) ? 0 : 1;
19  }
```

Subsumció Tot i que no és pròpiament un operador de l'algorisme genètic, sí en forma part, ja que es crida des d'aquí. La part més costosa de la subsumció és determinar si la regla d'un classificador és més general que la d'un altre. Cal comparar les regles símbol a símbol, ja que no n'hi ha prou que facin *match* — la condició que imposa la subsumció és més estricta. El codi font és el següent:

```
1   static inline int
2   does_subsume (const class_t * small, const class_t * big)
3   {
4     int s = 0, c = 0, i;
5     if (big->cl_action == small->cl_action && big->cl_exp > PAR_THETA_SUB
6         && big->cl_epsilon < PAR_EPSILON_0)
7       {
8         for (i = 0; i < n_bits; i++)
9       {
10        if ((big->cl_rule[s] & mask[i % 32]) != mask[i % 32] &&
11            (big->cl_rule[s] & mask[i % 32]) !=
12            (small->cl_rule[s] & mask[i % 32]))
13          return 0;
14        c++;
15        if (c == 32)
16          {
17            c = 0;
18            s++;
19          }
20      }
21         return 1;
22       }
23    return 0;
24  }
```

4.4 Proves de funcionament

Abans de passar a veure les millores de rendiment que s'aconsegueixen amb aquesta implementació, cal garantir que el funcionament segueix sent correcte. A tal fi, s'han executat exactament les mateixes proves que en l'apartat 3.7, amb els mateixos paràmetres de la taula 3.4. Les figures 4.1, 4.2 i 4.3 mostren els resultats, que es poden comparar amb les gràfiques de la implementació base.

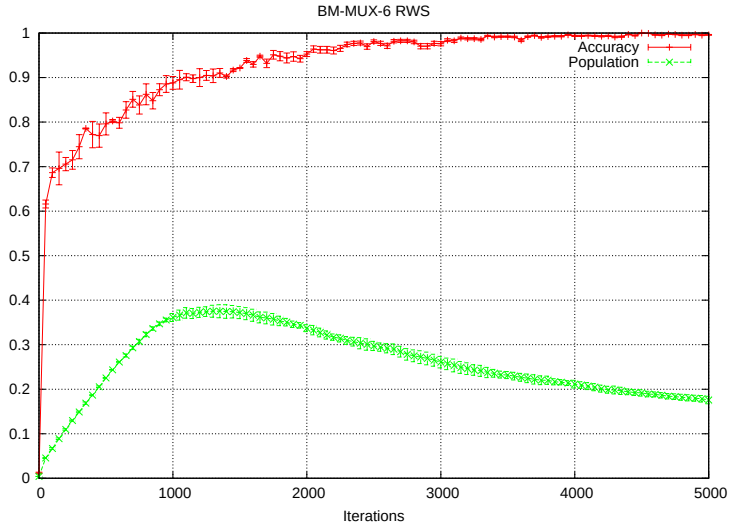

Figura 4.1: Multiplexor de 6 bits

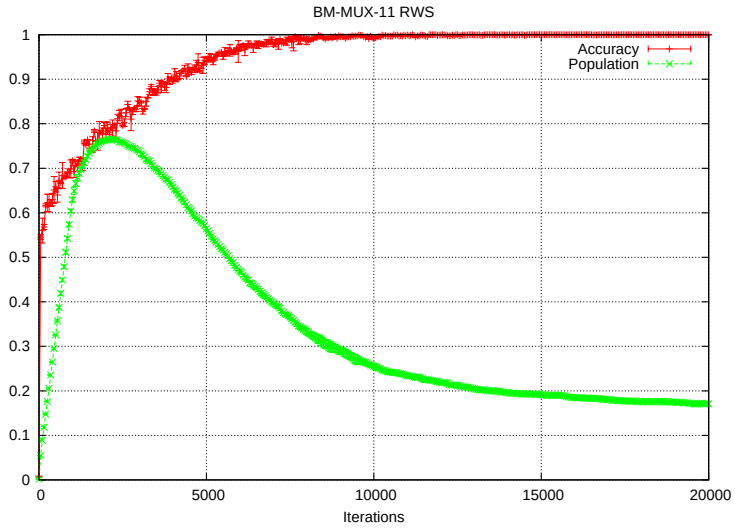

Figura 4.2: Multiplexor d'11 bits

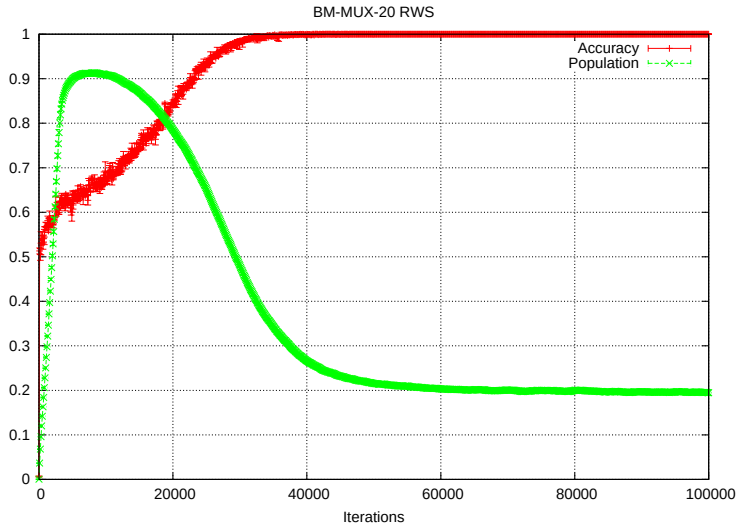

Figura 4.3: Multiplexor de 20 bits

4.5 Proves de rendiment

La finalitat d'aquest apartat — probablement el més interessant del capítol — és oferir una anàlisi del rendiment de la nova implementació, comparant-lo amb el de la implementació base. Cal tenir en compte, però, que no és l'objectiu fer una comparativa al detall, sinó que es busca trobar si, globalment, surt a compte canviar la representació — i la feina que això implica — i, en cas que surti a compte, fins a quin punt i en quina mesura. Per tant, no es tracta de saber si una implementació és un o dos segons més ràpida, sinó mirar proporcions, ordres de magnitud.

Segons Llorà and Sastry [2006], s'espera observar una gran millora en el rendiment: ells fan un experiment només amb el *matching*, i tenint en compte que aquesta part de l'XCS és la que té, de lluny, més pes en el temps d'execució, per llei d'Amdahl es podria esperar que la nova implementació fos substancialment més ràpida que l'anterior. En aquest apartat es faran proves que permetran determinar si tal afirmació és certa i, cas de ser-ho, en quina mesura.

Les compilacions i execucions s'ha fet en una màquina amb les següents característiques principals:

CPU Intel Core 2 Quad Q9300 2.5 GHz

Prova	Població aleatòria	Matching	Crossover	AS + act. params.	GA	Encreuament	Mutació	Subsumció
1	×	×						
2		×	×					
3		×	×	×				
4		×	×	×	×			
5		×	×	×	×	×		
6		×	×	×	×	×	×	
7		×	×	×	×	×	×	×

Taula 4.2: Resum de les proves

Memòria 4 GB

Cache 64 KB de primer nivell (32 K per a dades i 32 K per a instruccions), 3072 KB de segon nivell (segons `lscpu`).

S.O. Linux 2.6.33.4 (Slackware 13.1)

Compilador `gcc` 4.5.0

Glibc Versió 2.11.1

Al llarg d'aquest apartat es fan una sèrie de proves, de manera que, progressivament, es van activant diverses parts de l'XCS. La intenció de fer les proves d'aquesta manera és poder analitzar, amb més detall, l'impacte (en el rendiment) de la nova representació en cadascuna de les parts del sistema classificador. La taula 4.2 ofereix un resum de les proves fetes i quines parts s'han inclòs. Tots els temps que es mostren s'han obtingut amb el programa `time` (no s'ha de confondre amb la comanda interna del mateix nom de què disposa el shell).

Per a cada prova es fa una taula amb els resultats dels temps d'execució. Per a estalviar espai, els encapçalaments de les columnes són molt breus, el significat dels quals es pot consultar a la taula 4.3. No totes les taules tenen totes les columnes indicades. Tots els temps s'indiquen en segons.

4.5.1 Primera prova: el *matching*

La primera prova que s'ha dut a terme ha estat reproduir l'experiment dut a terme per Llorà and Sastry [2006], i analitzar si la implementació d'aquest treball experimenta el mateix increment de rendiment. A tal fi s'ha creat

Títol	Significat
n	Mida, en símbols, de mostres i regles
t_c	Temps d'execució, representació basada en caràcters
t_b	Temps d'execució, representació basada en mapes de bits
s	*Speedup* entre les dues representacions ($\frac{t_c}{t_b}$)
T_C	Com t_c però optimitzat pel compilador
T_B	Com t_b però optimitzat pel compilador
S	*Speedup* entre les versions optimitzades ($\frac{T_C}{T_B}$)
S_c	*Speedup* entre t_c i T_c ($\frac{t_c}{T_C}$)
S_b	*Speedup* entre t_b i T_B ($\frac{t_b}{T_B}$)

Taula 4.3: Significat de les columnes

una implementació retallada, on es genera una població aleatòria i, a partir d'aquesta població, les iteracions (tant d'aprenentatge com de testeig) consisteixen, únicament, en crear el *match set*.

Generació de la població Per a generar la població es disposa de les dues funcions següents (una per a la representació basada en caràcters i l'altra per a la basada en mapes de bits), que es criden abans de dur a terme un experiment (Les dues funcions es diuen igual perquè es troben en dos arxius diferents de codi font):

```
1  static void
2  fill_population (void)
3  {
4    for (size_t i = 0; i < PAR_N; i++)
5      {
6        for (int j = 0; j < n_bits; j++)
7     {
8       population[i].cl_rule[j] = getSingleBinaryDigit ()? '0' : '1';
9       if (getDoubleNumber () < PAR_P_SHARP)
10        population[i].cl_rule[j] = '#';
11    }
12       population[i].cl_num = 1;
13     }
14   micro_popsize = macro_popsize = PAR_N;
15 }
```

```
1  static void
2  fill_population (void)
3  {
4    for (size_t i = 0; i < PAR_N; i++)
5      {
6        int s = 0, c = 0;
```

```
7         bzero (population[i].cl_rule, sizeof (uint64_t) * size);
8         for (int j = 0; j < n_bits; j++)
9     {
10      if (getSingleBinaryDigit ())
11        population[i].cl_rule[s] |= sym_zero[j % 32];
12      else
13        population[i].cl_rule[s] |= sym_one[j % 32];
14      if (getDoubleNumber () < PAR_P_SHARP)
15        population[i].cl_rule[s] |= mask[j % 32];
16      c++;
17      if (c == 32)
18        {
19          c = 0;
20          s++;
21        }
22    }
23       population[i].cl_num = 1;
24     }
25   micro_popsize = macro_popsize = PAR_N;
26 }
```

Com es pot observar, cada símbol de la regla s'estableix, aleatòriament, a zero o u, i amb una probabilitat $P_{\#}$ el símbol queda establert a *don't care*.

Les iteracions queden reduïdes a obtenir una mostra de l'entorn i a la creació del *match set*, sense cridar, en cap cas, el procediment de *covering*.

***Matching* complet** S'han dissenyat dues implementacions del procés de *matching*: la primera d'elles segueix la filosofia de Llorà and Sastry [2006], en la qual es comparen tots els símbols de regla i mostra. Si el primer símbol ja no coincideix se segueixen comparant la resta de símbols, tot i que no caldria.

El codi font del procediment de *matching* és el següent (primer en la representació basada en caràcters i després la basada en mapes de bits):

```
1  void
2  MS_create (void)
3  {
4    size_t c, i;
5    match_set.ms_size = 0;
6    match_set.ms_actions[0] = match_set.ms_actions[1] = 0;
7
8    for (c = 0; c < macro_popsize; c++)
9      {
10       int v = 1;
11       for (i = 0; (int) i < n_bits; i++)
12     if (sigma[i] != population[c].cl_rule[i] &&
13         population[c].cl_rule[i] != '#')
14       v = 0;
15       if (v)
16     {
```

```
      match_set.ms_actions[population[c].cl_action]++;
      match_set.ms_elements[match_set.ms_size] = &population[c];
      match_set.ms_size++;
    }
     }
}
```

```
void
MS_create (void)
{
  size_t c;
  int i;
  match_set.ms_size = 0;
  match_set.ms_actions[0] = match_set.ms_actions[1] = 0;

  for (c = 0; c < macro_popsize; c++)
    {
      int v = 1;
      for (i = 0; i < size; i++)
    if ((sigma[i] & population[c].cl_rule[i]) != sigma[i])
      v = 0;
       if (v)
    {
      match_set.ms_actions[population[c].cl_action]++;
      match_set.ms_elements[match_set.ms_size] = &population[c];
      match_set.ms_size++;
    }
     }
}
```

La taula 4.4 mostra els resultats obtinguts amb aquest *matching*.

Tal com es pot observar, l'*speedup* que s'observa supera amb escreix el que es podria esperar, tal com també passa a Llorà and Sastry [2006]. L'explicació que donen és que, a banda de comparar més símbols per instrucció, s'aprofiten més les memòries *cache* de la màquina, la qual cosa contribueix a augmentar l'*speedup*.

***Matching* eficient** Ara bé, a la pràctica el procediment de *matching* no s'implementa d'aquesta manera; a la pràctica, en el moment que, al comparar un símbol (o grup de símbols) es veu que no fan *match*, es descarten la resta de símbols i es passa al següent classificador.

El codi font del procediment de *matching* és el següent (primer en la representació basada en caràcters i després la basada en mapes de bits):

```
void
MS_create (void)
{
  size_t c, i;
  match_set.ms_size = 0;
```

n	*a*	*b*	*c*	*d*	*e*	*f*
6	5.26	5.20	0.74	0.62	8.39	7.11
10	8.98	8.92	0.67	0.55	16.22	13.4
50	38.02	37.56	1.43	1.15	32.66	26.59
100	74.47	73.47	2.47	1.55	47.4	30.15
500	361.26	357.41	9.31	5.03	71.06	38.80
1000	721.53	713.15	18.47	9.98	71.46	39.06

Taula 4.4: Resultats per al *matching* complet

La columna *n* indica la mida, en símbols, de regles i mostres. Les columnes *a* i *c* indiquen el temps total d'execució de la prova per a la versió amb caràcters i amb mapes de bits, respectivament. Les columnes *b* i *d* indiquen el temps corresponent a la creació del *match set*. La columna *e* indica quantes vegades la versió optimitzada és més ràpida que la implementació base (pel que fa la creació del *match set*, mentre que la columna *f* mostra l'*speedup* global.

```
  match_set.ms_actions[0] = match_set.ms_actions[1] = 0;

  for (c = 0; c < macro_popsize; c++)
    {
      for (i = 0; (int) i < n_bits; i++)
   if (sigma[i] != population[c].cl_rule[i] &&
       population[c].cl_rule[i] != '#')
     break;
      if ((int) i == n_bits)
   {
     match_set.ms_actions[population[c].cl_action]++;
     match_set.ms_elements[match_set.ms_size] = &population[c];
     match_set.ms_size++;
   }
    }
}
```

```
void
MS_create (void)
{
  size_t c;
  int i;
  match_set.ms_size = 0;
  match_set.ms_actions[0] = match_set.ms_actions[1] = 0;

  for (c = 0; c < macro_popsize; c++)
    {
      for (i = 0; i < size; i++)
   if ((sigma[i] & population[c].cl_rule[i]) != sigma[i])
     break;
```

n	*a*	*b*	*c*	*d*	*e*	*f*
6	2.55	2.59	0.69	0.65	3.98	3.7
10	2.96	2.81	0.59	0.49	5.73	5.02
50	3.37	2.88	0.87	0.49	5.88	3.87
100	3.85	3.01	1.32	0.43	7	2.92
500	7.17	2.96	4.81	0.54	5.48	1.49
1000	11.17	3.07	9.13	0.55	5.58	1.22
5000	43.9	2.99	43.93	0.42	7.12	0.99
10000	84.67	3.26	87.09	0.53	6.15	0.97

Taula 4.5: Resultats per al *matching* eficient

La columna *n* indica la mida, en símbols, de regles i mostres. Les columnes *a* i *c* indiquen el temps total d'execució de la prova per a la versió amb caràcters i amb mapes de bits, respectivament. Les columnes *b* i *d* indiquen el temps corresponent a la creació del *match set*. La columna *e* indica quantes vegades la versió optimitzada és més ràpida que la implementació base.

```
14        if (i == size)
15    {
16      match_set.ms_actions[population[c].cl_action]++;
17      match_set.ms_elements[match_set.ms_size] = &population[c];
18      match_set.ms_size++;
19    }
20     }
21 }
```

Fent proves amb una metodologia idèntica a les anteriors, els resultats que s'obtenen són els de la taula 4.5.

Al comparar les dues taules, la primera observació que es pot fer és que la primera optimització ha estat aplicar un *matching* eficient: quan es troba un símbol que no fa *match* no cal seguir comparant la resta de símbols. Els temps d'execució són enormement més baixos. La segona observació és que els *speedups* són molt més modestos. El fet de canviar la representació segueix comportant un augment de velocitat. Sempre? No: les dues últimes proves mostren que la versió amb mapes de bits és més lenta, globalment, que la versió basada en caràcters. Com pot ser? Les taules 4.6 i 4.7 mostren la sortida del *profiler* per a les dues últimes proves amb l'XCS basat en mapes de bits, i les taules 4.8 i 4.9 mostren la sortida del *profiler* per a les dues últimes proves amb l'XCS basat en caràcters.

La creació del *match set* ha caigut en picat en el rànking de funcions més costoses. Les que s'enduen la palma són, ara, la generació de nombres aleatoris i, sorprenentment, dues funcions que aparentment haurien de ser molt modestes: l'obtenció d'una mostra i la generació de la població aleatò-

a	*b*	Funció
48.82	12.99	`getUnsignedLongNumber`
19.26	5.12	`getSingleBinaryDigit`
14.78	3.93	`MUX_get_value`
10.64	2.83	`getDoubleNumber`
1.88	0.50	`fill_population`
1.65	0.44	`init_by_array`
1.65	0.44	`MS_create`
1.39	0.37	`getLongNumber`

Taula 4.6: *Profiler* per a N=5000, XCS basat en mapes de bits

La columna *b* indica, en segons, el temps que ha tardat en executar-se la funció corresponent. La columna *a* indica quin percentatge representa sobre el total del temps d'execució.

a	*b*	Funció
47.78	25.44	`getUnsignedLongNumber`
20.73	11.04	`getSingleBinaryDigit`
15.36	8.18	`MUX_get_value`
10.21	5.43	`getDoubleNumber`
1.88	1.00	`fill_population`
1.75	0.93	`init_by_array`
1.48	0.79	`getLongNumber`
0.90	0.48	`MS_create`

Taula 4.7: *Profiler* per a N=10000, XCS basat en mapes de bits

a	*b*	Funció
48.59	12.79	`getUnsignedLongNumber`
18.25	4.81	`getSingleBinaryDigit`
11.52	3.03	`MS_create`
10.23	2.69	`getDoubleNumber`
6.75	1.78	`MUX_get_value`
1.75	0.46	`init_by_array`
1.56	0.41	`getLongNumber`
1.44	0.38	`fill_population`

Taula 4.8: *Profiler* per a N=5000, XCS basat en caràcters

a	*b*	Funció
51.74	26.53	`getUnsignedLongNumber`
20.09	10.30	`getSingleBinaryDigit`
10.74	5.51	`getDoubleNumber`
6.95	3.56	`MUX_get_value`
6.01	3.08	`MS_create`
1.85	0.95	`init_by_array`
1.50	0.77	`getLongNumber`
1.23	0.63	`fill_population`

Taula 4.9: *Profiler* per a N=10000, XCS basat en caràcters

ria. Aquest últim cas és especialment escandalós, ja que aquesta funció tan sols es crida 10 cops. Com es pot explicar?

- Per a obtenir una mostra cal donar valor a *tots i cadascun* dels símbols de la mostra. En el cas de l'XCS amb mapes de bits aquesta funció té un sobrecost afegit, ja que cal treballar amb màscares de bits i operacions binàries, cosa que no cal fer amb la representació basada en caràcters. Per a la prova amb mostres de cinc mil símbols, s'observa que l'obtenció de les mostres consumeix 3.93 segons, mentre que en l'XCS amb caràcters només en consumeix 1.78. Per a la prova amb deu mil símbols la diferència és de 8.18 segons versus 3.56. Si es fan e experiments i n iteracions, aquesta funció es crida $2en$ cops.

- Per a omplir la població cal donar valor a *tots i cadascun* dels símbols de la regla de *tots i cadascun* dels macroclassificadors. En el cas de l'XCS amb mapes de bits aquesta funció té un sobrecost afegit, ja que cal treballar amb màscares de bits i operacions binàries, cosa que no cal fer amb la representació basada en caràcters. Per a la prova amb mostres de cinc mil símbols, s'observa que la generació de la població triga 0.50 segons, mentre que en l'XCS amb caràcters només en consumeix 0.38. Per a la prova amb deu mil símbols la diferència és de 1 segon versus 0.63. Si es fan e experiments i n iteracions, aquesta funció es crida e cops.

La conclusió ve a ser la següent: la nova representació millora la creació del *match set*, però arriba un moment en què aquesta millora no és suficient per a *amagar* el sobrecost que aquesta mateixa representació imposa en d'altres parts de l'XCS – en aquest cas, l'obtenció de la mostra i la creació de la població aleatòria. Cal estar atents a aquesta última conclusió: les dues parts penalitzades són aquelles que han de treballar símbol a símbol, i en

l'XCS hi ha moltes rutines que han de treballar així (com ara els operadors de l'algorisme genètic, per posar un exemple).

Abans de passar a la següent prova, un últim comentari. Les proves fetes per Llorà and Sastry [2006] poden portar confusions. Si es llegeixen els resultats sense rigor, pot semblar que els autors d'aquest estudi afirmin que es poden obtenir millores del rendiment de més de noranta vegades, quan en realitat no diuen això. Part de la culpa la tenen els propis autors, ja que fan unes proves la significativitat de les quals és més que dubtosa: en primer lloc, perquè fan servir un *matching* completament ineficient; i en segon lloc, perquè les seves proves es basen únicament en comparar la millora sobre el *match set* i no sobre tot el sistema classificador XCS en conjunt. Les taules mostren que quan s'aplica un *matching* eficient, la millora obtinguda (en el millor dels casos) cau dràsticament (de 71 a 7 cops més ràpid).

A partir de la segona prova, el procés de *matching* es farà de forma eficient, és a dir, sense comparar tots els símbols si no hi ha necessitat de fer-ho.

4.5.2 Segona prova: el *covering*

L'objectiu d'aquesta i de les següents proves és anar-se apropant a l'XCS complet, analitzant pas a pas com influeixen, en el rendiment, els diferents components del sistema classificador. La segona prova presenta les següents diferències amb la primera:

1. Es partirà d'una població buida, en lloc d'una població generada aleatòriament.

2. S'activarà el procés de covering, forçant el valor del paràmetre $P_\#$ per tal que la quantitat de símbols específics (és a dir, aquells símbols que no valen *don't care*) de les regles es mantingui constant, sigui quina sigui la mida de regles i mostres. La taula 4.10 mostra aquests valors, calculats segons l'expressió següent:

$$P_\# = 1 - \frac{3}{n} \tag{4.3}$$

   ja que per al multiplexor de 6 bits amb $P_\# = 0.5$ hi ha 3 símbols específics a les regles creades per *covering*.

Les proves consisteixen en l'execució del multiplexor de 6 bits en les seves dues implementacions (representació amb caràcters i amb mapes de bits), activant únicament la formació del *match set* i el *covering*; es fan deu experiments de cinc mil iteracions i cada experiment comença amb una població completament buida. La taula 4.11 mostra els resultats de les execucions.

n	p
6	0.5
10	0.7
50	0.94
100	0.97
500	0.994
1000	0.997
5000	0.9994
10000	0.9997

Taula 4.10: Valor del paràmetre $P_{\#}$ segons la mida n de mostres i regles

n	t_c	t_b	s
6	0.14	0.08	1.75
10	0.28	0.11	2.55
50	1.11	0.47	2.36
100	1.76	0.90	1.96
500	8.64	4.78	1.81
1000	19.35	8.44	2.29
5000	90.09	41.17	2.19
10000	186.58	82.71	2.26

Taula 4.11: Resultats per a la prova del *covering*

Es pot observar que l'*speedup* ha caigut dràsticament. A les taules 4.4 i 4.5 s'aprecien uns increments de velocitat força considerables, especialment per al *matching* complet. Com és que en aquesta prova s'ha perdut aquesta millora tan considerable? La taula 4.12 pot donar pistes al respecte.

La mencionada taula indica la quantitat de vegades que s'ha cridat el procediment de *covering* al llarg dels deu experiments. Això vol dir que, de mitjana, la població ha contingut entre 22 i 37 macro-classificadors. En la prova anterior la població estava completament plena, a totes les iteracions, ja que s'havia creat una població aleatòria al començar cada experiment. En canvi, en aquesta prova els experiments començaven amb una població buida.

4.5.3 Tercera prova: XCS complet l'algorisme genètic

Aquesta prova ha consistit en una implementació completa de l'XCS sense l'algorisme genètic (i, per tant, sense els seus operadors). Els resultats dels temps totals d'execució es poden veure a la figura 4.13.

n	c	d
6	222	239
10	304	282
50	369	278
100	262	241
500	290	344
1000	357	322
5000	316	274
10000	349	332

Taula 4.12: Crides a la funció de *covering*

n	t_c	t_b	s
6	0.18	0.12	1.5
10	0.39	0.18	2.17
50	1.06	0.58	1.83
100	2.16	1.04	2.08
500	9.81	4.66	2.11
1000	18.06	9.58	1.89
5000	95.86	46.11	2.08
10000	183.15	91.61	1.99

Taula 4.13: Resultats per a la tercera prova

Es pot observar que, respecte la prova anterior, els *speedups* tendeixen a ser inferiors. En alguns casos són una mica superiors, que es pot deure a la influència dels nombres aleatoris en algunes parts del sistema classificador, però la tendència és a ser inferiors. L'explicació és senzilla: la creació de l'*action set* i l'actualització dels paràmetres són processos independents de la representació de les regles i les mostres, i l'obtenció del *reward* (necessari per a l'actualització dels paràmetres) no es beneficia del canvi de representació, la qual cosa fa que baixi la influència de la creació del *match set* sobre el temps total d'execució, i per llei d'Amdahl l'*speedup* també disminueixi.

Es pot observar que la creació del *match set* i l'actualització dels paràmetres dels seus classificadors no influeix significativament en el rendiment.

4.5.4 Quarta prova: introducció de l'algorisme genètic

En aquesta quarta prova s'activa l'algorisme genètic, però es deshabiliten els seus operadors. Els resultats dels temps totals d'execució es poden veure a la figura 4.14.

n	t_c	t_b	s
6	0.21	0.15	1.4
10	0.38	0.22	1.73
50	1.05	0.59	1.78
100	2.04	1.07	1.91
500	9.88	4.68	2.11
1000	15.69	10.66	1.47
5000	79.79	45.19	1.77
10000	169.65	96.98	1.75

Taula 4.14: Resultats per a la quarta prova

En quant a funcionament, aquesta quarta prova afegeix la creació de dos classificadors fills, idèntics (pel que fa la regla) als seus pares; quan els fills s'insereixen a la població, la numerositat dels seus pares augmenta. No hi ha cap altra diferència.

És sorprenent, sens dubte, que en la implementació basada en caràcters els temps d'execució siguin inferiors als de la prova anterior. Tenint en compte que no s'han reduït parts de l'XCS, sinó tot al contrari, com és possible? En el fons, tal com s'ha dit, la prova tercera i quarta són enormement similars. Les diferències en el rendiment són molt petites fins a mil símbols per regla (i mostra), i augmenten a partir d'aquesta mida. Que en mides petites les diferències siguin escasses és perfectament comprensible: les proves són molt semblants. Però per què tals diferències augmenten amb regles grans? Aquí l'efecte dels nombres aleatoris és molt més fort, ja que no és el mateix que, al fer *matching*, s'hagin de comparar dos símbols que se n'hagin de comparar cinc-cents. Aquest efecte queda amplificat pel fet de fer deu experiments.

Per a la implementació en mapes de bits, els temps d'execució són molt similars. És força comprensible que no es notin els efectes que es comentaven anteriorment, pel fet de comparar molts símbols en una sola instrucció. Per altra banda, els *speedups* són una mica inferiors.

4.5.5 Cinquena prova: introducció de l'encreuament

En aquesta quarta prova s'activa l'algorisme genètic, habilitant únicament l'operador d'encreuament (en la prova anterior no hi havia cap operador habilitat). Els resultats dels temps totals d'execució es poden veure a la figura 4.15.

Per a la implementació basada en caràcters, els temps d'execució augmenten considerablement, mentre que per a la implementació basada en mapes de bits, es mantenen força semblants, superiors en alguns casos i

n	t_c	t_b	s
6	0.59	0.46	1.28
10	1.16	0.57	2.04
50	2.42	0.74	3.27
100	3.86	1.34	2.88
500	17.24	5.10	3.38
1000	43.94	9.94	4.42
5000	119.39	49.84	2.40
10000	348.08	95.48	3.65

Taula 4.15: Resultats per a la cinquena prova

n	t_c	t_b	s
6	1.30	0.71	1.83
10	2.59	1.02	2.54
50	8.27	2.66	3.11
100	12.18	3.44	3.54
500	24.86	6.72	3.70
1000	33.42	11.92	2.80
5000	108.15	53.15	2.03
10000	189.66	105.99	1.79

Taula 4.16: Resultats per a la sisena prova

inferiors en d'altres. Aquesta prova introdueix una diferència significativa amb l'anterior: l'algorisme genètic crea fills diferents dels pares (ja que s'hi aplica l'operador d'encreuament, amb una probabilitat del 80%). La conseqüència d'això és, a més del temps que es gasta en executar l'operador, que es creen nous classificadors, a més dels generats per *covering*. I com més classificadors, més tarda el *matching* (a més d'altres parts de l'XCS), i més tarda en executar-se una iteració del sistema classificador.

Pel fet d'augmentar la població i guanyar pes el procediment de *matching*, és lògic que els *speedups* s'hagin incrementat.

4.5.6 Sisena prova: introducció de la mutació

En aquesta prova, a més de l'operador d'encreuament, s'activa també l'operador de mutació. Els resultats dels temps totals d'execució es poden veure a la figura 4.16.

El fet d'afegir l'operador de mutació fa que els fills creats per l'algorisme genètic presentin més variacions, a part de les ocasionades per l'operador

d'encreuament. Evidentment, això afegeix cost computacional al sistema classificador. Per tant és d'esperar que els temps d'execució siguin superiors.

Pel que fa la versió basada en caràcters, els temps d'execució en part són superiors, però no del tot. Fins a cinc-cents símbols, els temps són sensiblement superiors. Però a partir de mil, són sensiblement inferiors. En canvi, en la versió basada en mapes de bits, els temps sempre són superiors.

4.5.7 La prova final

L'última prova consisteix en executar tot l'XCS sencer, tant per al multiplexor de 6 símbols com per al d'11. S'ha optat, en aquest cas, per fer dues versions de la prova: la primera s'ha fet amb el codi compilat sense opcions d'optimització del compilador (només per al multiplexor de 6 bits), la segona s'ha fet amb el codi optimitzat pel compilador, segons les opcions següents:

`-std=gnu99` Indica al compilador que ha de seguir l'estàndar ISO 9899:1999 ("C99") amb les extensions pròpies de GNU C. D'aquesta manera es disposa de les últimes característiques del llenguatge i les extensions del compilador `gcc`.

`-Wall` Indica al compilador que sigui mostri tants *warnings* com sigui possible.

`-Wextra` Indica al compilador que mostri encara més *warnings* que amb l'opció anterior.

`-O3` Indica al compilador que activi el màxim nivell d'optimització.

`-pipe` Indica al compilador que faci servir *pipes* en lloc de fitxers temporals.

`-fgcse-sm` Indica al compilador que intenti treure dels bucles les instruccions de *store*.

`-fgcse-las` Indica al compilador que intenti eliminar instruccions de *load* redundants que apareixen després d'instruccions de *store* a la mateixa adreça de memòria.

`-fgcse-after-reload` Indica al compilador que faci una passada extra per tal d'intentar eliminar instruccions de *load* redundants.

`-march=native` Indica al compilador que optimitzi el codi tenint en compte la màquina on s'està compilant el codi.

`-msse` Activa l'ús del conjunt d'instruccions SSE.

`-msse2` Activa l'ús del conjunt d'instruccions SSE2.

`-msse3` Activa l'ús del conjunt d'instruccions SSE3.

`-mfpmath=sse` Indica al compilador que empri el conjunt d'instruccions SSE a l'hora de generar instruccions amb nombres en coma flotant.

`-static` Indica al compilador que no empri llibreries dinàmiques, sinó estàtiques.

`-combine` Indica que tots els arxius de codi font s'han de passar al compilador d'un sol cop. Aquesta opció és necessària per al correcte funcionament de la següent opció.

`-fwhole-program` Assumeix que l'actual unitat de compilació representa tot el programa sencer. Com que al compilador se li passen *tots* els arxius de codi font que formen el programa, aquesta assumpció és certa.

`-flto` Indica al compilador que activi les optimitzacions en temps de *linkage* (enllaçat).

`-fexcess-precision=fast` Indica al compilador que, per als nombres en coma flotant, segueixi la precisió oferta pels registres del processador, més enllà del que digui l'estàndar de C.

Cal tenir en compte que el fet d'indicar al compilador que empri instruccions SSE no significa que efectivament hi haurà instruccions d'aquest conjunt al codi generat. El compilador pot no tenir tota la informació (o pot no fer segons quines assumpcions amb total seguretat) a l'hora de generar codi. Cal recordar que la norma fonamental d'un compilador és generar codi correcte.

La taula 4.17 ofereix els resultats dels temps d'execució per al multiplexor de 6 símbols, la taula 4.18 per al d'11 símbols. El valor del paràmetre $P_{\#}$ ha anat augmentant a mesura que augmentava la mida de les mostres.

A l'analitzar les dades que es mostren a les taules 4.17 i 4.18, es pot arribar a les següents conclusions:

1. El canvi de representació sempre surt a compte, tot i que la millora obtinguda varia segons la mida de mostres i regles. Aquest *speedup* és més alt quan la mida se situa entre els 100 i els 500 símbols. Cal tenir en compte que tot aquest treball ha versat sobre el multiplexor; d'altres problemes (com el de la paritat) oferiran d'altres conclusions.

   Malgrat tot, les millores obtingudes són relativament modestes. Assumint que qualsevol programador activarà les optimitzacions del compilador, el canvi de representació duplica el temps d'execució. Tot i

n	t_c	t_b	T_C	T_B	s	S	S_c	S_b
6	1.18	0.65	0.47	0.27	1.82	1.74	2.51	2.41
10	2.26	1.05	0.91	0.42	2.15	2.17	2.48	2.5
50	9.01	3.47	2.74	1.32	2.60	2.08	3.29	2.63
100	12.87	4.06	3.44	1.61	3.17	2.14	3.74	2.52
500	24.10	6.64	5.13	2.22	3.63	1.38	4.70	2.99
1000	35.29	11.82	6.51	3.50	2.99	1.86	5.42	3.38
5000	98.29	51.69	17.07	14.25	1.90	1.20	5.76	3.63
10000	190.09	101.40	31.11	27.56	1.87	1.13	6.11	3.68
50000	–	–	148.71	134.28	–	1.11	–	–

Taula 4.17: Prova final per al multiplexor de 6 símbols

n	p	t_c	t_b	s
11	0.5	5.96	3.45	1.73
50	0.89	23.72	12.51	1.90
100	0.945	28.00	13.85	2.02
500	0.989	42.17	15.60	2.70
1000	0.9945	53.98	21.08	2.56
5000	0.9989	128.45	67.36	1.91
10000	0.99945	204.18	127.08	1.61

Taula 4.18: Prova final per al multiplexor d'11 símbols

que no és despreciable, un es pregunta si és suficient pel que la indústria necessita.

2. El compilador fa una feina magnífica a l'hora d'optimitzar el codi. Es pot objectar que el codi font creat per l'autor del treball pot ser poc eficient o mal dissenyat; fins i tot si això fos cert, cal recordar que un compilador no pot corregir errors de disseny — per exemple, si s'ordena mitjançant l'algorisme de la bombolla, el compilador no substituirà tal algorisme per un *quicksort*.

4.6 Sumari

A l'analitzar els resultats del *profiling* per a la implementació base es va observar que el procediment de *matching* era el que més temps consumia, i es va optar per millorar-ne el rendiment. A tal fi, es va decidir fer un canvi de la representació, emprant dos bits per símbol en lloc de vuit, i comparant 32 símbols (en arquitectures de 64 bits) per instrucció. S'ha analitzat l'impacte que té aquest canvi en la implementació, i s'han fet mesures de rendiment

comparant la nova implementació amb l'anterior. Ha valgut la pena? Llorà and Sastry [2006], al seu paper, diuen:

> We have proposed an encoding that provides a smaller memory footprint and exploits hardware vector operations via Altivec and SSE2 instruction sets to dramatically reduce the amount of time spend on the matching process. Remarkable speedups were obtained, suggesting great time benefits to any LCS that requires intensive matching stages, as XCS requires, makint it possible to deal large number of conditions—500,000 conditions—in a reasonable time.
>
> Further work should focus on introducing the proposed matching approaches to some of the freely available XCS implementations and benefit from the reduction of the execution time.

Aquest treball justament ha consistit en aplicar el nou *matching* a una implementació de l'XCS, creada *ex professo* per al propi treball. Per què no ho fan fet els propis autors del paper, és una pregunta que queda a l'aire. Tal article és del 2006, hi ha hagut temps per a fer-ho. D'alguna manera, han deixat la feina a mitges.

Pel que fa als "great time benefits", les proves parlen per si soles. Ni de lluny s'arriben a les millores que s'obtenen a l'article, bàsicament pels motius següents:

1. La implementació que fan del *matching* consisteix en comparar tots els símbols, fins i tot quan ja no és necessari. Si la implementació es limita a comparar fins que es troba un símbol que no fa *match*, es guanya molt més temps i, conseqüentment, l'*speedup* cau. A més, aquesta manera de realitzar el *matching* fa que tal procediment perdi molt de pes en el temps total d'execució de l'XCS.

2. Tot i que el canvi de representació beneficia el *matching*, aquesta part de l'XCS no representa tot el seu temps d'execució (tot i que sí una bona part); a banda que el canvi de representació perjudica d'altres parts del sistema classificador.

Malgrat tot, no es pot considerar que no hagi valgut la pena canviar la representació: les millores obtingudes, semblin o no modestes, justifiquen l'esforç que suposa.

Capítol 5

Optimització amb instruccions SIMD

L'objectiu d'aquest capítol és presentar i explicar la segona optimització que s'ha fet del sistema classificador XCS. Es començarà parlant dels fonaments teòrics de la tècnica d'optimització, com ha canviat la present implementació, quin és el seu rendiment i quines millores s'han produït respecte la implementació base i la primera optimització.

5.1 Fonaments teòrics

5.1.1 Apunt històric

Les sigles SIMD signifiquen *Single Instruction, Multiple Data*, i tal com indica el seu nom, consisteix en realitzar la mateixa operació en un conjunt de dades — contràriament a l'enfocament típic, en què es realitza una operació sobre una sola dada. Neixen a principis dels anys setanta per a donar resposta a aplicacions científiques, que realitzaven una gran quantitat de càlculs sobre grans quantitats de dades. El fet de poder realitzar aquests càculs amb més d'una dada a la vegada permetia aumentar dràsticament el rendiment de tals aplicacions. L'elevadíssim cost dels processadors amb aquesta mena d'instruccions els feia únicament viables en hardware enfocat al sector científic i a les empreses amb més recursos econòmics.

Els grans avenços en electrònica i en arquitectura de computadors han permès que actualment es disposi de microprocessadors molt potents a preus

realment assequibles. Tècniques que fa uns quants anys eren exclusives dels xips més cars ara són habituals fins i tot en els més barats (per exemple, el *pipelining* o l'execució fora d'ordre). L'any 1996 Intel va introduir les extensions MMX, i al cap d'uns dos anys IBM va introduir Altivec a la seva línia de processadors PowerPC. L'any 1998 AMD va crear les extensions 3DNow! per a la seva línia de CPUs. Intel ha seguit creant extensions, anomenades SSE1, SSE2, etc.

5.1.2 Les instruccions SIMD

Les implementacions més típiques d'aquestes instruccions consisteix en afegir registres de gran capacitat (64 o 128 bits, típicament) i instruccions que permeten fer les operacions en paral·lel, prenent el registre, per exemple, com un vector de 2 enters de 32 bits, 4 enters de 16 bits o 8 enters de 8 bits (cas que el registre sigui de 64 bits).

Hi ha dues maneres de fer servir aquestes instruccions: mitjançant extensions del llenguatge de programació o bé deixant que sigui el compilador qui emeti les instruccions. Aquest segon enfocament requereix de compiladors suficientment bons en optimització i generació de codi, i pocs compiladors actualment són capaços de vectoritzar el codi. Un compilador no pot sacrificar mai la semàntica del codi, és a dir, el fet d'optimitzar-lo o vectoritzar-lo no pot alterar el comportament del programa. Sovint el compilador no té prou informació sobre el codi o l'algorisme per tal d'optimitzar més agressivament.

Cal tenir en compte, finalment, que no totes les aplicacions es poden beneficiar per igual d'aquestes instruccions vectorials. Només les aplicacions que facin gran quantitat de càlculs i les operacions actuïn sobre diverses dades alhora se'n veuran beneficiades. Típicament els bucles són els candidats per excel·lència a beneficiar-se de les instruccions vectorials, ja que un bucle es pot arribar a substituir per una o unes quantes d'aquestes instruccions.

5.1.3 Les instruccions SSE

Per a aquest treball s'han emprat les instruccions SSE per a l'optimització del sistema classificador. Per què aquestes instruccions i per què es considera que podran millorar el rendiment de l'XCS?

1. Tal com es va veure a l'apartat 4.1, l'XCS dedica molt de temps a la formació del *match set*, ja que per a cada classificador present en la població ha de comprovar si la seva regla fa *match* amb la mostra obtinguda de l'entorn. En la implementació amb mapes de bits es va aconseguir millorar el temps d'execució canviant la representació de regles i mostres, la qual cosa accelerava la formació del *match set*

(justament perquè accelerava el procediment de *matching*). Les instruccions vectorials permetran optimitzar més agressivament.

2. El conjunt d'instruccions SSE (*SIMD Streaming Extensions*) es troba disponible tant en les CPUs d'Intel com en les d'AMD, per la qual cosa es pot assegurar una màxima compatibilitat. Cal tenir en compte que aquestes CPUs són àmpliament usades tant en entorns domèstics com professionals (industrials, científics, investigació, etc). D'altres arquitectures disposen d'altres conjunts d'instruccions similars (com ara Altivec per a PowerPC).

SSE afegeix a la CPU 8 nous registres de 128 bits, `xmm0–xmm7`. En mode de 64 bits s'afegeixen 8 nous registres més, `xmm8–xmm15`. El contingut de cadascun d'aquests registres es pot interpretar com:

- Quatre reals (*floating-point*) de 32 bits.
- Dos reals de 64 bits.
- Dos enters de 64 bits.
- Quatre enters de 32 bits.
- Vuit enters de 16 bits.
- Setze enters (o caràcters) de 8 bits.

A partir d'aquí s'afegeix un nou joc d'instruccions per a operar amb aquests registres i controlar diferents aspectes del seu funcionament. Per tal que el software pugui fer ús d'aquestes extensions cal que el sisteam operatiu ho permeti, ja que en cas contrari hi podria haver problemes a l'hora de guardar i recuperar el context dels processos en execució.

El joc d'instruccions SSE és realment gran, permetent molta flexibilitat i una gran varietat d'operacions. De totes maneres, el que el sistema classificador XCS és realment simple: poder comparar el màxim nombre de símbols per instrucció. Per tant, doncs, el que realment importa és la mida dels registres `xmm`[1] i no tant la flexibilitat o varietat del joc d'instruccions.

5.2 La nova representació

La representació segueix sent la mateixa que la de la taula 4.1. El fet d'usar instruccions SSE no canvia la filosofia de la representació en mapes de bits, simplement permet comparar més símbols per instrucció.

[1] Intel té "en ment" ampliar la mida d'aquests registres a 256 bits en un futur no gaire llunyà, i més endavant es podria ampliar a 512 o 1024 bits.

5.3 Impacte en la implementació

En el capítol dedicat a la representació en mapes de bits s'ha detallat extensament com afecta el canvi a la implementació del sistema classificador. El fet d'emprar instruccions SSE permet comparar 128 bits amb una sola instrucció (versus els 64 bits de la implementació anterior), la qual cosa significa comparar fins a 64 símbols per instrucció. Com afecta tot això a la implementació? Únicament al procediment de *matching*, ja que, com s'ha vist al capítol anterior, és únicament el *matching* el que es pot beneficiar de comparar més símbols alhora.

La implementació amb instruccions SSE és calcada a l'anterior, fent únicament els canvis que es descriuen a continuació:

5.3.1 Multiplexor

Ha calgut fer únicament un canvi al multiplexor, per tal d'assegurar que hi haurà prou espai per a enters de 128 bits per tal que el procediment de *matching* pugui fer bé la seva feina:

```
1  void
2  MUX_init (const int _k_bits)
3  {
4    k_bits = _k_bits;
5    real_bits = n_bits = k_bits + (1 << k_bits);
6  #if (PAR_PADDING > 0)
7    n_bits = PAR_PADDING;
8  #endif
9    size = n_bits / 32;
10   if (n_bits % 32)
11     size++;
12   if (n_bits % 64)
13     size++;
14   sigma = calloc (size, sizeof (uint64_t));
15   assert (sigma);
16 }
```

La variable `n_bits` conté la quantitat de bits total de regles i mostres, i la variable `size` conté la quantitat d'enters de 64 bits que es reserven per a guardar regles i mostres. Per a la implementació anterior, si la quantitat de bits no era múltiple de 32 (la quantitat de símbols que es podien comparar per instrucció), s'arrodonia el valor de `size`. Ara cal arrodonir-lo si `n_bits` no és múltiple de 64, perquè amb les instruccions SSE podem comparar 64 símbols alhora.

Pot sorprendre el lector el fet de fer dues comprovacions i de calcular `size` dividint `n_bits` per 32 en lloc de 64. Té la seva explicació: tal com s'ha dit anteriorment, la implementació amb instruccions SSE és clavada a

l'anterior, i per tant s'ha de mantenir la resta de les parts de l'XCS funcionant perfectament.

5.3.2 *Matching*

La funció de creació del *match set* queda com segueix:

```
void
MS_create (void)
{
  size_t c;
  int i;
  match_set.ms_size = 0;
  match_set.ms_actions[0] = match_set.ms_actions[1] = 0;
  __m128i *sigma128 = (void *) sigma;
  __m128i *rule128;
  __m128i vir, vii;

  for (c = 0; c < macro_popsize; c++)
    {
      rule128 = (void *) population[c].cl_rule;
      for (i = 0; i < size / 2; i++)
   {
     vir = _mm_load_si128 ((__m128i *) & rule128[i]);
     vii = _mm_load_si128 ((__m128i *) sigma128);
     vir = _mm_and_si128 (vir, vii);
     vii = _mm_cmpeq_epi32 (vir, vii);
     if (bcmp (msk, &vii, 16) != 0)
       break;
   }
      if (i == size / 2)
   {
     match_set.ms_actions[population[c].cl_action]++;
     match_set.ms_elements[match_set.ms_size] = &population[c];
     match_set.ms_size++;
   }
    }
}
```

5.4 Proves de funcionament

Tal com s'ha fet per a la implementació anterior, cal garantir que la nova versió segueix funcionant correctament. A tal fi, s'han executat exactament les mateixes proves que en l'apartat 3.7, amb els mateixos paràmetres de la taula 3.4. Les figures 5.1, 5.2 i 5.3 mostren els resultats, que es poden comparar amb les gràfiques de la implementació anterior.

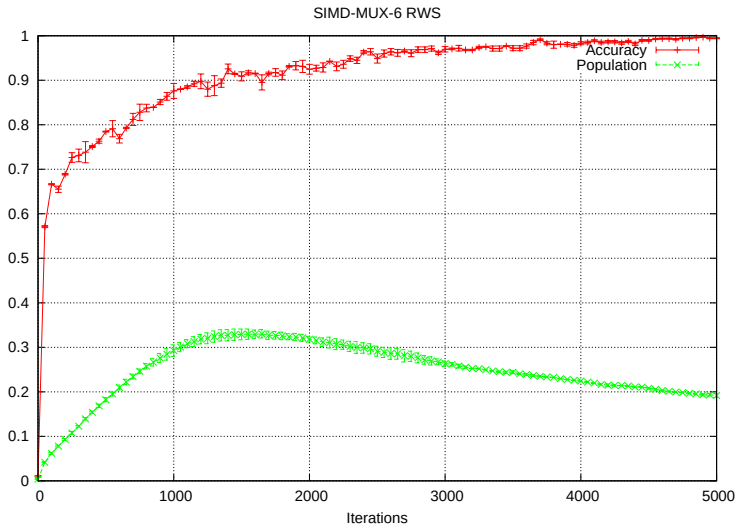

Figura 5.1: Multiplexor de 6 bits

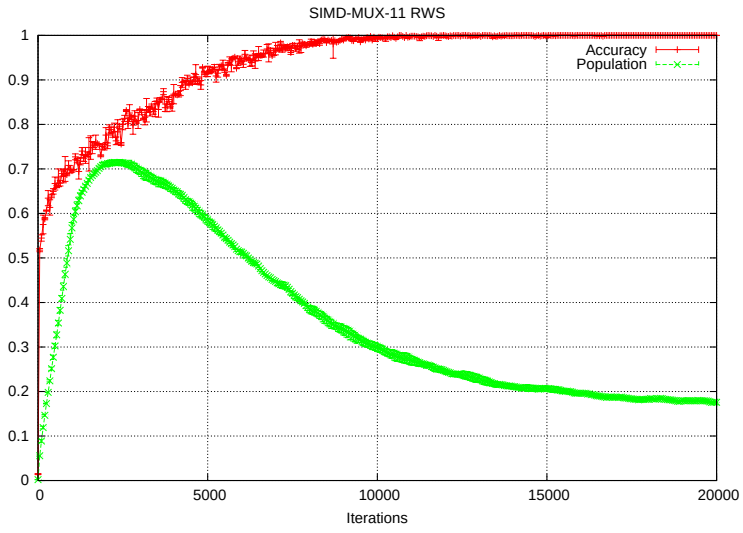

Figura 5.2: Multiplexor d'11 bits

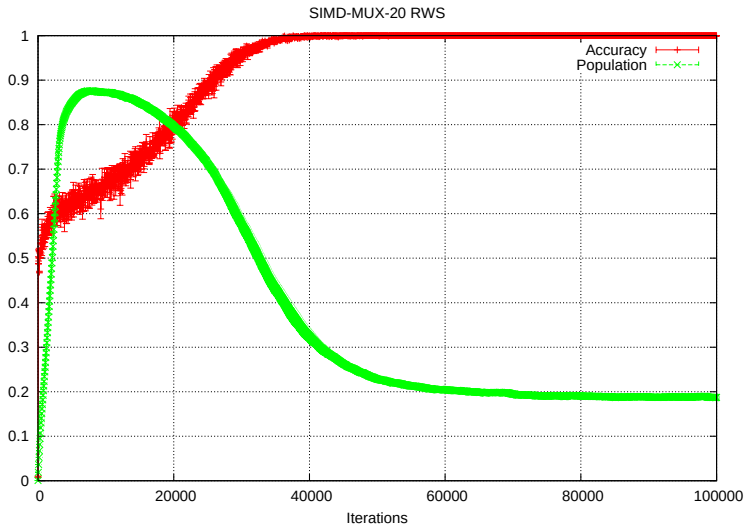

Figura 5.3: Multiplexor de 20 bits

n	t	s_b	s_c
6	0.60	1.08	1.97
10	1.08	0.97	2.09
50	2.41	1.44	3.74
100	3.00	1.35	4.29
500	9.98	0.66	2.41
1000	18.98	0.62	1.86
5000	90.38	0.57	1.09
10000	178.85	0.57	1.06

Taula 5.1: Prova final per al multiplexor de 6 símbols

5.5 Proves de rendiment

L'objectiu d'aquesta secció és mostrar com s'ha millorat el rendiment a través de les instruccions SSE. S'ha decidit passar directament a les proves "finals" (com les de l'apartat 4.5.7. Això es deu al fet que la present implementació guarda una enorme similitud amb l'anterior. Aquestes proves s'han fet en les mateixes condicions que les de l'esmentat apartat.

A la taula 5.1, la columna t indica els segons que ha tardat la prova, la columna s_b indica l'*speedup* respecte la implementació anterior, i la columna t_c implica l'*speedup* respecte la implementació base. Els resultats de les proves són decepcionants, la qual cosa es pot deure a dos factors: a una implementació deficient o bé al fet que les instruccions SSE no són tan bones com semblaven.

Pot ser que la implementació sigui deficient, sens dubte. Els autors de paper motivador d'aquest treball (Llorà and Sastry [2006] amb prou feines ofereixen un exemple de com fer servir les instruccions SSE per al *matching*, que és tot el que han optimitzat de l'XCS. Per tant, segur que hi ha coses que es poden millorar. Però part de la culpa també se les emporten les instruccions SSE. Ja es va comentar, en el capítol anterior, que el canvi de representació beneficia poderosament una política de *matching* poc eficient, política que compara tots els símbols encara que no sigui necessari. Ja es va veure que quan es va passar a una política més eficient, la millora introduïda per la nova representació queia en picat.

Amb les instruccions SSE passa el mateix. Poden beneficiar quan es comparen molts símbols a la vegada, cosa que passa poc sovint. Per això els resultats són tan decebedors: les millores respecte la implementació anterior són modestes, i només per a regles de 50 a 100 símbols. A partir de 500, acaba sent contraproduent. El lector es podria preguntar com és que empitjora, en lloc de simplement *no millorar*. És una bona pregunta, l'explicació de la qual no acaba de ser gaire clara.

Capítol 6

Conclusions i línies de futur

A mesura que els equips informàtics (processadors, memòries, busos, etc) han anat guanyant en rendiment, la *informàtica* ha anat incrementant aplicacions i ha penetrat cada cop més camps. Els ordinadors d'avui en dia fan coses impensables (o difícilment imaginables) deu o quinze anys enrere, i l'equip que fa trenta anys costava milions de dòlars és avui assequible en l'àmbit domèstic.

El món de la intel·ligència artificial cada cop és més conscient de la importància del rendiment de les seves aplicacions i dels seus algorismes. Des de l'any 2008, el GECCO (el congrés de més renom dins de la intel·ligència artificial) ha vist presentar molts papers sobre com millorar el rendiment d'algorismes i tècniques existents, fins i tot s'han obert competicions de CUDA (una interfície de programació que permet aprofitar el hardware gràfic de nVidia). Aquest treball s'ha unit a aquests esforços, continuant una primera aproximació (Llorà and Sastry [2006]) a l'optimització de l'XCS.

En aquesta memòria s'ha començat donant una introducció del que és l'aprenenatge artificial, per tal de situar el domini del coneixement en què es troba l'aplicació objecte d'estudi, o sigui, el sistema classificador XCS. S'ha descrit en detall el seu funcionament a través d'una implementació base, que ha estat optimitzada a través d'un canvi de representació, seguint la proposta de Llorà and Sastry [2006]. A diferència del mencionat article, en aquest treball s'ha fet una implementació completa del sistema classificador, no només una execució limitada al procediment de *matching*. Com és que

els autors del citat article no van fer una implementació completa, ni tampoc l'han feta des de llavors? La pregunta queda sense respondre.

Després de les proves realitzades, quina conclusió es pot extreure sobre el canvi de representació proposat? Ni de lluny s'arriben a les millores aconseguides per Llorà and Sastry [2006], i ja s'ha explicat abastament per què. De totes maneres, tampoc cal tirar-ho tot per terra: aconseguir multiplicar el rendiment per 1.5, 2 o 2.5 val la pena, sense cap dubte. Ara bé, convé no perdre de vista que la implementació de l'XCS d'aquest treball s'ha basat sobre el problema del multiplexor, que, tot i que és un *benchmark* molt emprat, sense cap mena de dubte no és un *problema real*.

La viabilitat d'una optimització se sol determinar avaluant-ne el cost i el benefici, és a dir, si la millora obtinguda justifica el cost que suposa l'optimització. A més d'aquests dos criteris, convindria avaluar també l'impacte que es causa en el manteniment de l'aplicació optimitzada. En el cas de l'XCS, el cost de canviar la representació ha estat relativament baix, i la millora obtinguda és força bona. Per tant, es pot considerar recomanable fer aquest canvi de representació, almenys per al problema del multiplexor.

Què passa per a d'altres tipus de problemes? Dependrà, bàsicament, de la representació que emprin. El multiplexor empra un alfabet ternari, que permet codificar símbols amb només dos bits, i permet fer *matching* simplement amb instruccions de comparació. En la mesura que d'altres entorns mantinguin aquestes facilitats, es podrà esperar beneficiar-se de millores similars a les experimentades pel problema del multiplexor. Ara bé, si les representacions són més complexes, llavors les millores poden no ser tan significatives.

La norma general seria que no es poden donar normes generals. Cada problema s'ha d'analitzar en particular, cas per cas, i aquesta és la principal línia de futur: aplicar el canvi de representació a d'altres entorns — i, a poder ser, en entorns *reals*, propers a la indústria.

Bibliografia

L. Bull, E. Bernadó-Mansilla, and J. Holmes, editors. *Learning classifier systems in data mining*. Studies in Computational Intelligence. Springer, 2008.

Martin V. Butz and Stewart W. Wilson. An algorithmic description of xcs. Technical Report 2000017, Illinois Genetic Algorithms Laboratory, 2000.

D. E. Goldberg. *Genetic algorithms in search, optimization & machine learning*. Addison Wesley, 1 edition, 1989.

J. H. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, 1975.

J. H. Holland. Adaptation. In R. Rosen and F. Snell, editors, *Progress in Theoretical Biology*, volume 4, pages 263–293. New York: Academic Press, 1976.

J. H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In D.A. Waterman and F. Hayes-Roth, editors, *Pattern-directed inference systems*, pages 313–329. Academic Press, San Diego, USA, 1978.

Xavier Llorà and Kumara Sastry. Fast rule matching for learning classifier systems via vector instructions. Technical Report 2006001, Illinois Genetic Algorithms Laboratory, 2006.

Xavier Llorà, Rohith Reddy, Brian Matesic, and Rohit Bhargava. Toward better than human capability in diagnosing prostate cancer using infrared spectroscopic imaging. Technical Report 2007006, Illinois Genetic Algorithms Laboratory, 2007.

Albert Orriols Puig. *New Challenges in Learning Classifier Systems: Mining Rarities and Evolving Fuzzy Models*. PhD thesis, Enginyeria i Arquitectura La Salle. Universitat Ramon Llull, Novembre 2008.

Olivier Sigaud and Stewart W. Wilson. Learning classifier systems: A survey. *Soft Computing*, 11(11):1065–1078, Setembre 2007.

Ruchard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1 edition, 1998. ISBN 0-262-19398-1.

Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

Stewart W. Wilson. Generalization in the xcs classifier system. *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 1998.