Capítulo 9

Algoritmos Genéticos

9.1 Introducción

El término *algoritmo genético* se le atribuye a Holland, sin embargo, otros científicos trabajaron en ideas parecidas durantes los 60's.

En particular, Ingo Rechenberg y Hans-Paul Schwefel, en Alemania, desarrollaron las ideas de las etrategias evolutivas (*Evolutionsstrategie*), mientras que Bremermann y Fogel desarrollaron las ideas de programación evolutiva.

La parte común en estas estrategias es la mutación y la selección.

A diferencia de las otras dos estrategias evolutivas, los algoritmos genéticos se concentran más en la combinación de soluciones.

Otra diferencia es que se permite tener una representación codificada de las variables del problema, en lugar de las variables mismas.

9.1.1 Programación Evolutiva

Desarrollado principalmente por Lawrence J. Fogel en los 60's.

Algoritmo básico:

- Genera aleatoriamente una población inicial
- Aplica mutación
- Calcula aptitud de cada hijo y selecciona por torneo

Por ejemplo, podemos pensar en cambiar un atómata finito mediante mutación para reconocer ciertas entradas.

La mutación puede cambiar un símbolo de salida, cabiar una transición, agregar un estado, borrar un estado y cambiar el estado inicial.

Estado Actual	A	A	B	B	C	C
Símbolo Entrada	0	1	0	1	0	1
Estado siguiente	B	C	B	C	B	C
Símbolo Salida	N	N	Y	N	N	Y

9.1.2 Estrategias Evolutivas

Propuestas en Alemania en 64 para resolver problemas hidrodinámicos complejos.

La versión original (1+1)-EE usaba un solo padre del cual se generaba un solo hijo.

Se mantenia el hijo solo si era mejor que el padre

Un individuo nuevo es generado introduciendo ruido Gaussiano: $\vec{x}_{t+1} = \vec{x}_t + N(0, \vec{\sigma})$, donde N es una vector de números Gaussianos independientes con una media cero y desviación estandar $\vec{\sigma}$.

Por ejempo, supongamos que queremos optimizar: $f(x_1, x_2) = 100(X_1^2 - X_2)^2 + (1 - X_1)^2$ donde, $-2048 \leq x_1, x_2 \leq 2048$.

Ahora supongamos que tenemos el siguiente individuo generado aleatoriamente: (-1,1) y tenemos las siguientes mutaciones:

$$x_1^{t+1} = x_1^t + N(0, 1.0) = -1.0 + 0.61 = 0.39$$
$$x_2^{t+1} = x_2^t + N(0, 1.0) = 1.0 + 0.57 = 1.57$$
$$f(x_t) = f(-1, 1) = 4$$
$$f(x_{t+1}) = f(-0.39, 1.57) = 201.416$$

Rechenberg introdujo el concepto de población en donde M padres generan 1 hijo, y Schwefel introdujo el uso de múltiples hijos, pudiendo quedarse con los M mejores hijos o los M mejores individuos considerando a los padres y a los hijos.

Rechenberg también formuló una regla para ajustar la desviación estandar durante el proceso evolutivo para garantizar convergencia, conocida como "la regla del éxito 1/5": la razón entre mutaciónes exitosas y el total de mutaciones debe de ser 1/5.

Lo que dice es que la razón entre mutaciones exitosas y el total de mutaciones debe ser 1/5. Si es mayor, incrementa la desviación estandar (divídela entre 0.817), si en menor, decrementala (multiplícala por 0.817).

En las estrategias evolutivas se evoluciona no sólo a las variables del problema, sino también a los parámetros mismos de la técnica (i.e., las desviaciones estándar). A esto se le llama *auto-adaptación*.

9.1.3 Algoritmos Genéticos

Los Algoritmos Genéticos (GA) pueden verse como una familia de procedimientos de búsqueda adaptivos.

Su nombre se deriva de que están basados en modelos de cambio genético en una población de individuos. Esto es:

- Noción Darwiniana de aptitud (*fitness*) que influye en generaciones futuras.
- Apareamiento que produce descendientes en generaciones futuras.

- Operadores genéticos que determinan la configuración genética de los descendientes (tomada de los padres).

Un punto clave de estos modelos, es que el proceso de adaptación no se hace cambiando incrementalmente una sola estructura, sino manteniendo una población de estructuras a partir de las cuales se generan nuevas estructuras usando los operadores genéticos.

Cada estructura en la población está asociada con una aptitud y los valores se usan en competencia para determinar qué estructuras serán usadas para formar nuevas estructuras.

Una de sus características es su abilidad de explotar información acumulada acerca de un espacio de búsqueda inicialmente desconocido para guiar la búsqueda subsecuente a subespacios útiles.

Su aplicación está enfocada sobretodo a espacios de búsqueda grandes, complejos y poco entendidos.

El precio es que se pueden necesitar un número grande de muestras para que se tenga suficiente información para guiar muestras subsecuentes a subespacios útiles.

En su forma más simple, un GA está orientado a desempeño (i.e., hacer cambios estructurales para mejorar el desempeño).

Una de las ideas más importantes es definir estructuras *admisibles* en el sentido que esten bien definidas y puedan ser evaluadas.

Surgen a finales de los 50ś, principios de los 60ś.

Se le reconoce a Holland como el fundador.

Diferencias con métodos tradicionales de búsqueda y optimización:

- Trabajan con un conjunto de parámetros codificados y no con los parámetros mismos.
- Inician la búsqueda desde un conjunto de puntos, no de uno solo.

- Usan una función a optimizar en lugar de la derivada u otro conocimiento adicional.
- Usan reglas de transición probabilísticas no determinísticas.

9.2 Anatomía de un GA

1. Modulo Evolutivo: mecanismo de decodificación (interpreta la información de un cromosoma) y función de evaluación (mide la calidad del cromosoma). Solo aquí existe información del dominio.
2. Modulo Poblacional: tiene una representación poblacional y técnicas para manipularla (técnica de representación, técnica de arranque, creitrio de selección y de reemplazo). Aquí también se define el tamaño de la población y la condición de terminación.
3. Modulo Reproductivo: contiene los operadores genéticos.

9.3 Algoritmo Básico

La tabla 9.1 muestra los pasos principales de los algoritmos genéticos.

9.4 El teorema de esquemas

Proporciona el fundamento teórico de porqué los GA pueden resolver diversos problemas. En su análisis se considera el proceso de selección y los operadores de cruce y mutación.

Un esquema se construye utilizando un nuevo símbolo (*) para representar un comodín (no importa) que puede aparear ambos valores (0 o 1). E.g., el esquema 11*00* representa las cadenas: 111001, 111000, 110001, 110000.

El orden de un esquema es el número de elementos que no son "*" dentro del esquema.

Tabla 9.1: Algoritmo de Algoritmos Genéticos.

Procedimiento AG
tiempo = 0
Inicializa Población(tiempo)
Evalua Población(tiempo)
Mientras no condición de terminación
  tiempo = tiempo + 1
  Construye Población(tiempo) a partir de Población(tiempo-1) usando:
    - Selección
    - Modifica Población(tiempo) usando Operadores Genéticos
    - Evalua Población(tiempo)
    - Reemplaza
Fin mientras

La longitud que define a un esquema es la distancia entre la primera posición fija y la última posición fija.

El teorema dice que si exiten $N(S,t)$ instancias de un esquema S en una población al tiempo t, en el siguiente tiempo el valor esperado de instancias en la nueva población esta acotado por:

$$E[N(S,t+1)] \geq \frac{F(S,t)}{\overline{F}(t)} N(S,t)[1-\epsilon(S,t)]$$

donde $F(S,t)$ es la aptitud del esquema S, $\overline{F}(t)$ es la aptitud promedio de la población, y $\epsilon(S,t)$ es un término que refleja el potencial del algoritmo genético de destruir instancias del esquema S.

El teorema regresa solo un valor esperado de la siguiente generación, por lo que tratar de extrapolar y decir que los esquemas buenos crecen exponencialmente en las siguientes generaciones (como a veces se dice) es completamente absurdo.

Existen también argumentos sobre la hipótesis de bloques constructores, en donde pequeños esquemas o bloques constructores se usan para construir la solución óptima, sin embargo, no existe evidencia de esto.

9.5 Puntos a considerar en GA

- Codificación de los parámetros de un problema.

  Dentro de la codificación a veces se usan codificaciones que tengan la propiedad de que números consecutivos varíen a lo más en un bit (e.g., codificación de Gray).

  En la codificación se busca idealmente que todos los puntos estén dentro del espacio de solución (sean válidos).

  Tradicionalmente se buscan representaciones que favorescan los esquemas cortos de bajo orden.

  Pueden existir problemas de interdependencia (problemas para los GA si existe mucha y es preferible usar otro método si es casi nula).

- Función de aptitud.

  Es la base para determinar que soluciones tienen mayor o menor probabilidad de sobrevivir.

  Se tiene que tener un balance entre una función que haga diferencias muy grandes (y por lo tanto una convergencia prematura) y diferencias muy pequeñas (y por lo tanto un estancamiento).

- Criterios de tamaño de la población.

  Balance entre una población muy pequeña (y por lo tanto convergencia a máximo local) y una población muy grande (y por lo tanto requerimiento de muchos recursos computacionales).

  Los primeros intentos de tratar de estimar el tamaño de población óptimo basados en el teorema de esquemas resultaban en crecimiento exponenciales de la población con respecto al tamaño de gen.

  Experimentalmente se vió que esto no es necesario y se buscó el tamaño mínimo para poder alcanzar todos los puntos en el espacio de búsqueda.

  Se requiere solo que existe al menos cada una de las posibles instancias de los alelos en el conjunto de genes. Para genes binarios, la probabilidad de que existe al menos un alelo en cada punto es:

  $$P_2 = (1 - (1/2)^{M-1})^l$$

  donde M es el tamaño de la población y l es el tamaño del gen.

Esto sugiere que con poblaciones de tamaño $O(logl)$ es suficiente.

Normalmente las poblaciones se seleccionan aleatoriamente, sin embargo, esto no garantiza una selección que nos cubra el espacio de búsqueda uniformemente.

Se pueden introducir mecanismos más sofisticados para garantizar diversidad en la población.

Aunque normalmente se elige una población de tamaño fijo, también existen esquemas de poblaciones de tamaño variable.

- Criterio de selección.

  Individuos son copiadas de acuerdo a su evaluación en la función objetivo (aptitud). Los más aptos tienen mayor probabilidad a contribuir con una o más copias en la siguiente generación (se simula selección natural).

  Se puede implementar de varias formas, sin embargo, la más común es la de simular una ruleta, donde cada cadena tiene un espacio en ella proporcional a su valor de aptitud.

$$Pr(h) = \frac{Aptitud(h)}{\sum_{j=1}^{N} Aptitud(h_j)}$$

  En lugar de seleccionar uno a la vez, se pueden generar N selecciones teniendo en la ruleta N apuntadores que estan separados uniformemente, por lo que un giro en la ruleta nos porporciona N individuos. A esto se le conoce como *stochastic universal selection*.

  Otra alternativa es usar selección por torneo. Se seleccionan aleatoriamente N individuos que compiten entre sí seleccionando el mejor. En esta alternativa el mejor individuo siempre es seleccionado.

  Una ventaja de este esquema es que solo necesitamos comparar si un gen es mejor que otro, por lo que posiblemente no tenemos que evaluar la función de aptitud.

  Sin embargo, haciendo muestreo con reemplazo, existe una probabilidad de aproximadamente 0.368 ($\approx e^{-1}$) de que un gen no sea seleccionado.

Finalmente, el otro tipo de selección es por ranqueo. Simplemente se ordenan los genes. La probabilidad de seleccionar un gen ranqueado como el k-ésimo ($P(k)$), en el caso de ranqueo lineal, es:

$$P(k) = \alpha + \beta k$$

donde α y β son constantes y

$$\sum_{k=1}^{M} (\alpha + \beta k) = 1$$

- Nueva población.

  Se pueden seleccionar individuos de la población actual, generar una nueva población y reemplazar con ella completamente a la población que se tenia (esquema generacional).

  También a veces se mantienen los N mejores individuos de una población a la siguiente (esto parece ser la mejor opción). Si se conserva solo el mejor individuo se llama estrategia elitista, si se mantiene un subconjunto se habla de poblaciones traslapadas.

- Criterio de paro.

  Normalmente cuando un porcentaje alto de la población converge a un valor o después de un número fijo de evaluaciones en la función de aptitud. Si con ese valor no se llega a la medida esperada (si es que se conoce) o simplemente para tratar de mejorar la solución, entonces se toma una pequeña proporción y se inyecta "diversidad genética" (se generan aleatoriamente nuevos individuos), o inclusive se reemplaza completamente la población.

- Operadores genéticos.

  Normalmente se hace cruza seguida de mutación, pero se podría hacer una o la otra, por ejemplo.

  Algunas personas proponen hacer mucha cruza al principio e incrementar la mutación conforme pasa el tiempo.

  - Cruce: tiene una alta probabilidad de ser utilizado y es considerado como el más importante dentro de los AG. Permite la generación de nuevos individuos tomandos características de individuos padres. Consiste en seleccionar dos individuos después del

proceso de selcción, determinar una posición de cruce aleatoria e intercambiar las cadenas entre la posición inicial y el punto de cruce y el punto de cruce y la posición final.

Existen diferentes tipos de cruza. (i) Cruza simple: un solo punto de cruza (una máscara de 1's seguida de 0's), (ii) cruza de dos puntos y (iii) cruza uniforme (generando una máscara con 1's y 0's siguiendo una distribución de Bernoulli).

Existe evidencia empírica que sugiere que cruza en un punto no es la mejor opción.

- Mutación: tiene baja probabilidad de ser utilizado y permite introducir nueva información no presente en la población. Opera sobre un solo individuo, determina una posición y la invierte con cierta probabilidad. Permite salir de máximos locales.

  En lugar de generar un número aleatorio para cada alele y ver si se muta o no, se puede generar un número que nos de el número de mutaciones a realizar (M) y simplemente generar M números aleatorios entre 1 y el tamaño del gen.

  Lo único que falta por determinar es que valor usar cuando existan más de dos posibles valores.
- Inversión: tiene baja probabilidad. Incrementa la capacidad de exploración. Permite generar cadenas que serían difíciles de obtener con los otros dos operadores. Opera en un individuo, determina dos posiciones dentro de la cadena e invierte la subcadena.
- Existen más...

Vamos a ver tres formas de uso común de algortimos genéticos: (i) cambio de parámetros (ii) cambio de estructuras de datos (iii) cambio de programas

9.6 GA para Cambiar Parámetros

Una forma simple e intuitiva es identificar los parámetros clave que controlan el comportamiento de un sistema y cambiarlos para mejorar su desempeño.

La idea viene desde Samuel (59) (checkers) hasta cambio de pesos en redes neuronales.

Idea: parámetros = genes
Cadenas fijas de genes para cada parámetro.

El operador de cruce genera nuevas combinaciones de parámetros y la mutación nuevos valores.

Pero... hay que considerar el número de valores distintos que los genes (parámetros) pueden tomar.

Las poblaciones normalmente representan una fracción pequeña de los posibles valores.

Si mutación es muy baja, podemos caer en máximos locales, si los aumentamos nos lleva a una búsqueda aleatoria que disminuye la probabilidad de que individuos nuevos tengan un desempeño alto.

Los GA son más efectivos cuando cada gen puede tomar pocos valores (en este sentido genes binarios son óptimos), i.e., un parámetro se representa como grupos de genes.

A GA le va mejor, porque aunque el espacio sea el mismo, aparte de mutación, el operador de cruce ahora también puede generar nuevos valores.

e.g., en lugar de representar un parámetreo que puede tomar 2^{30} valores (dejando a mutación explorarlos todos!!!), lo representamos con 30–genes binarios.

Otro punto a considerar es el de convergencia al óptimo global. En teoría todos los puntos en el espacio de búsqueda tienen una probabilidad diferente de cero de ser visitados. En la práctica la espera puede ser impráctica.

Podemos ver que los GA constituyen heuristicas de muestreo poderosas que pueden encontrar rápidamente soluciones de buena calidad en espacios complejos.

9.7 GA para Cambiar Estructuras de Datos

Se puede aplicar a cambiar mecanismos de control como agendas. Por ejemplo, el TSP.

A primera vista, uno puede pensar en "linearizar" las estructuras de datos y mapearlas a cadenas binarias.

Un punto fundamental es que debemos de cuidar que la representación no nos represente, en su mayoría, estructuras de datos ilegales.

e.g., en el TSP una representación directa sería representar una ciudad por gen. Sin embargo, los operadores de mutación y cruce nos explorarían todos las tuplas con N ciudades, cuando lo que nos interesa son las permutaciones de las N ciudades.

Para evitar esto podemos:

- Diseñar representaciones alternativas.
- Usar diferentes operadores genéticos (e.g., inversión).
- Restricciones para asegurar que mutación y cruce nos producen solo puntos válidos.

9.8 GA para Cambiar Código

El espacio descrito por cambios en código es más grande y más complejo.

Tenemos que escoger el lenguaje de programación adecuado. Una representación "natural" de programas puede ser desastrosa, ya que mutación y cruce producirian muy pocos programas sintácticamente correctos y menos aún semánticamente correctos.

Una alternativa es diseñar nuevos operadors genéticos específicos del lenguaje para preservar por lo menos la integridad sintáctica y enfocarse a lenguajes con sintáxis sencilla (e.g., Lisp o Prolog "puros").

Sin embargo, algo como Lisp, que es procedural por naturaleza, hace que el cambio de dos lineas de código o pequeños cambios hagan todo el programa sin sentido.

Esto orillo (por lo menos en un principio) a enfocarse a sistemas de reglas de producción.

Ventajas:

- Se puede considerar el conocimiento como datos a ser manipulados.
- El conocimiento puede ejecutarse.

Surgieron dos formas de representar reglas:

- *The Pitt approach*: cada individuo de la población representa un programa completo de reglas.
- *The Michigan approach*: cada individuo de la población representa una regla y la población completa un programa.

9.8.1 The Pitt Approach

Hay que considerar la representación. Cruce nos da nuevas combinaciones de reglas y mutación nuevas reglas.

Como los genes puede asumir muchos valores podemos convergir prematuramente, por lo que hay que usar una representación binaria para que cruce también produsca nuevas reglas.

Pero ahora tenemos que asegurar que produzcan reglas válidas y potencialmente útiles. Una forma es asegurarse que todas las reglas tienen una longitud fija.

También se pueden hacer operadores sensibles a la representación para hacer cambios de acuerdo a la sintáxis.

El otro punto es el tamaño de la población. Difícil justificar que el sistema de reglas tenga una lognitud fija (aunque se podría justificar en términos de reglas redundantes).

Existen enfoques con poblaciones de longitud variable (Smith 80). Aqui se tiene que tener un balance entre el tamaño y su desempeño.

Ciclo de operación:

- Selección: se observa el desepeño de cada individuo (conjunto de reglas) de la población con respecto a los otros individuos en la población y se selecciona (posiblemente varias veces) asegurandose que su número de descendientes es proporcional a su desempeño con respecto a los otros individuos.
- Recombinación: se combinan pares de individuos por cruce, pero a diferentes niveles de granularidad (LS-1).
  - A nivel de listas de reglas (normal).
  - A nivel de reglas (con partes de pares de reglas). Se pueden ver como variantes de reglas de padres de alto desempeño.
  - Operador de inversión.

9.8.2 Michigan approach

También llamados sistemas clasificadores. Cada regla (clasificador) manipula un mensaje.

El sistema interactua con el ambiente el cual manda mensajes (en este sentido es parecido a un sistema experto).

Las reglas son del tipo:

> If condición (mensajes que satisface)
> Then acción (manda un mensaje)

En general, todos los mensajes son del mismo tamaño sobre un alfabeto predefinido.

Componentes:

- Interface de entrada: convierte el estado del ambiente en mensajes.
- Clasificadores o reglas: definen el proceso de mensajes.
- Lista de mensajes: mensajes de entrada y dados por las reglas disparadas.
- Interface de salida: traduce los mensajes en acciones para modificar el ambiente.

Ciclo de ejecución:

- Coloca todos los mensajes de la interface de entrada en la lista de mensajes.
- Itera:
  - Compara todos los mensajes con la lista de condiciones de los clasificadores.
  - Dispara todos los clasificadores que satisfacen sus condiciones (y tienen suficiente fuerza) formando con cada acción una lista de mensajes nuevos.
  - Reemplaza todos los mensajes por los mensaje nuevos.

La especificación de las condiciones de las reglas normalmente es por medio de {0,1,#} y permiten tener $\neg$ *Mensaje*.

El ambiente proporciona el refuerzo (premio/castigo) y se tiene que pensar en un método para asignar crédito a las reglas que intervinieron en la solución.

Se han propuesto varios modelos: (i) profit–sharing plan (ii) bucket brigade model (iii) rudi.

Uno de los métodos (por lo menos inicialmente) más usados es el *bucket brigade* el cual distribuye la ganancia entre las reglas que sirvieron para la solución.

A cada regla se le asigna una cantidad (fuerza) que se ajusta y se usa para competir.

La competencia se basa en oferta: las reglas más fuertes son las que ganan.

En el proceso de oferta se toman en cuenta dos factores:

- utilidad: fuerza de la regla. $s(C,T)$ = fuerza del clasificador C al tiempo t
- relevancia: especificidad. $R(C)$ = número de no "#"'s en la condición de la regla entre su longitud.

En el ciclo general se calcula la oferta (bid) de los clasificadores

$$B(C,t) = b * R(C) * s(C,t)$$

donde,
b = constante < 1 (e.g., 1/8, 1/16).

El tamaño de la oferta determina la probabilidad de que el clasificador ponga su mensaje (e.g., la probabilidad puede decrecer exponencialmente con la oferta).

Al usar probabilidad quiere decir que a veces la regla más probable no dispara.

Bucket brigade trata a cada regla como un intermediario el cual se entiende con sus proveedores (reglas que hacen que se satisfagan sus condiciones) y sus consumidores (reglas que satisfacen sus condiciones con sus acciones).

Cuando una regla gana una oferta, paga parte a sus proveedores (si no gana no paga nada) y puede recibir pago de sus consumidores.

Su fuerza aumenta cuando recibe más de lo que dá.

Un clasificador (C) al depositar un mensaje paga:

$$s(C,t+1) = s(C,t) - B(C,t)$$

Los clasificadores (C') que mandaron mensajes para que ganara el clasificador (proveedores) aumentan su fuerza:

$$s(C', t+1) = s(C', t) + a * B(C', t)$$

donde $a = 1/$elementos de C'.

Las reglas finales obtienen el pago del ambiente.

Las reglas malas son eliminadas por competencia.

Con el tiempo, las mejores reglas tienen más peso y son favorecidas a disparar por el mecanismo de resolución de conflictos.

Puntos en general de los GA para reglas de producción:

- Reglas no ordenadas.
- LHS: algunos patrones de longitud fija usando {0,1,#}, otros argumentan que es demasiado restrictivo.

  En general hay que establecer un balance entre simplicidad y poder expresivo.

  También se ha sugerido que los patrones del LHS nos regresen un valor de apareo.
- Memoria de trabajo: dejar o no que las reglas hagan cambios en la memoria de trabajo. Los que están en contra argumentan que las aplicaciones no ameritan la generalidad y complejidad adicional (se tendría que restringir el número de acciones sobre la memoria de trabajo) y la mayoría de los trabajos en aprendizaje de conceptos se han enfocado en reglas tipo estimulo–respuesta.
- Retroalimentación: A pesar de que uno considere la representación adecuada y la arquitectura de sistemas de producción adecuados, la efectividad también depende en buena medida de la retroalimentación.

  No todos lo esquemas de premio/castigo son adecuados (e.g., muy poco premio a menos que sea casi la solución no nos sirve).

  El bucket brigade distribuye la retroalimentación. Pero también se tiene que considerar el número de reglas y el tiempo.

Se puede pensar en dar retroalimentación tanto a cada regla como al conjunto de reglas.

9.8.3 XCS: sistema clasificador genético

Es un sistema clasificador reciente que difiere con los anteriores en lo siguiente:

- La aptitud de un clasificador se basa en la precisión de la predicción de pago más que en la predicción misma.
- El algoritmo genético actúa sobre los conjuntos de acciones en lugar de la población en su conjunto.
- XCS no tiene lista de mensajes y es sólo aplicable a ambientes Markovianos.

Al igual que otros clasificadores, el ambiente da una entrada sensorial del tipo $\sigma(t) \in \{0,1\}^L$. El sistema responde con una acción $\alpha(t) \in \{a_1, \ldots, a_n\}$. Cada acción recibe una recompensa escalar $\rho(t)$ asociada.

Los problemas pueden ser de un solo paso (las situaciones sucesivas no están relacionadoas entre si) o de varios pasos (sí están relacionadas).

XCS tiene una población de clasificadores cada uno siendo una regla del tipo condición-acción con la siguiente información:

- Condición $C \in \{0,1,\#\}^L$.
- Acción $A \in \{a_1, \ldots, a_n\}$.
- Predicción p que es la recompensa esperada si se usa este clasificador.

También se puede guardar, el error en la predicción (ϵ), la aptitud (F) del clasificador, la experiencia (exp), que es el número de veces que ha estado en el conjunto de acciones, la última vez que se usó, el tamaño promedio

de los conjuntos de acciones a los que ha pertenecido (as), el número de clasificadores que este clasificador representa (n).

Se consideran cuatro conjuntos en XCS:

- La población de clasificadores $[P]$.
- El conjunto de clasificadores $[M]$ que aparea la situación actual $\sigma(t)$.
- El conjunto de acciones $[A]$ con los clasificadores que proponen una acción.
- El conjunto de acciones anterior $[A]_{-1}$.

La población inicial puede estar vacía o llenarse con N clasificadores generados aleatoriamente. La diferencia entre los dos enfoques es poca, por lo que en general se empieza con un conjunto vacío.

Cuando se recibe una señal de entrada se seleccionan todos los clasificadores que aparean la situación ($[M]$). Se ve para cada posible acción cual es su predicción de pago. Se selecciona una acción a_i tomando en cuenta esta predicción y se seleccionan todos los clasificadores que proponen dicha acción $[A]$.

La acción ganadora se ejecuta y el conjunto de acciones previo $[A]_{-1}$ (en caso de problemas de multiples pasos) se modifica usando un esquema parecido a Q-learning. El algoritmo viene descrito en la tabla 9.2.

Al momento de generar el conjunto de clasificadores que aparea la situación (MATCH SET), si el número de acciones presentes en $[M] < \theta$ (un cierto umbral que se tiene que definir), entonces se crea un nuevo clasificador que aparea las condiciones del ambiente ($\sigma(t)$) y que contiene *don't cares* (#) con cierta probabilidad ($P_{\#}$). La acción del clasificador se selecciona aleatoriamente entre las que no están presentes en $[M]$.

Para generar el PREDICTION ARRAY ($[PA]$), se suma la multiplicación de la predicción de cada acción que aparece en $[M]$ por su aptitud, y el total se divide entre el total de las aptitudes de las $[M]$ que tienen esa acción. Osea que todas las predicciones y aptitud de las acciones iguales que aparecen en $[M]$ se consideran para regresar la predicción de cada acción.

Tabla 9.2: Algoritmo de XCS.

$\rho_{-1} \leftarrow 1$
do
    $\sigma \leftarrow$ situación del ambiente
    $[M] \leftarrow$ genera MATCH ARRAY a partir de $[P]$ usando σ
    $PA \leftarrow$ genera PREDICTION ARRAY a partir de $[M]$
    $act \leftarrow$ selecciona acción de acuerdo a PA
    $[A] \leftarrow$ genera ACTION SET a partir de $[M]$ de acuerdo a act
    realiza acción act y recibe recompensa ρ
    if($[A]_{-1} \neq \emptyset$)
        P $\leftarrow \rho_{-1} + \gamma * max(PA)$
        Actualiza $[A]_{-1}$ usando P
        corre GA en $[A]_{-1}$ considerando σ_{-1}
    if(fin de problema)
        P $\leftarrow \rho$
        Actualiza $[A]$ usando P
        corre GA en $[A]$ considerando σ
        vacía $[A]_{-1}$
    else
        $[A]_{-1} \leftarrow [A], \rho_{-1} \leftarrow \rho, \sigma_{-1} \leftarrow \sigma$
until criterio de paro

Tabla 9.3: Actualización de parámetros en XCS.

$\forall cl \in [A]$
$cl.epx++$
if $(cl.epx < 1/\beta)$
$cl.p \leftarrow cl.p + (P - cl.p)/cl.exp$
$cl.\epsilon \leftarrow cl.\epsilon + (|P - cl.p| - cl.\epsilon)/cl.exp$
$cl.as \leftarrow cl.as + (\sum_{c\in[A]} cl.n - cl.as)/cl.exp$
else
$cl.p \leftarrow cl.p + \beta * (P - cl.p)$
$cl.\epsilon \leftarrow cl.\epsilon + \beta * (|P - cl.p| - cl.\epsilon)$
$cl.as \leftarrow cl.as + \beta * (\sum_{c\in[A]} cl.n - cl.as)$
if $(cl.\epsilon < \epsilon_0)$
$\kappa(cl) \leftarrow 1$
else
$\kappa(cl) \leftarrow \alpha * (cl.\epsilon/\epsilon_0)^{-\nu}$
$SumAcc \leftarrow SumAcc + \kappa(cl) * cl.n$
$\forall cl \in [A]$
$cl.F \leftarrow cl.F + \beta * (\kappa(cl) * cl.n/SumAcc - cl.F)$

Tomando en cuenta $[PA]$, la selección de la acción (acc) puede hacerse con $\epsilon-$greedy (aunque puede hacerse de otras formas).

El ACTION SET son todos los clasificadores en $[M]$ cuya acción es acc.

El pago de cada clasificador es una combinación de recompensa inmediata y del estimado de la máxima recompensa en el siguiente estado.

La actualización de los parámetros de los clasificadores se hace en el siguiente orden: experiencia (exp), estimación de predicción (p), predicción de error (ϵ), tamaño promedio del conjunto de acciones al que pertenece el clasificador (as), la aptitud (F). Esta actualización se hace como se ilustra en la tabla 9.3.

El algoritmo genético para generar nuevos individuos, se corre si se rebasa también un cierto umbral, que determina cuándo se uso la última vez el algoritmo genético. En tal caso, se seleccionan dos clasificadores tomados de $[A]$ usando ruleta, basados en su aptitud. Si se cruzan, su predicción, error

y aptitud se toman como el promedio de sus padres.

Cruza sólo se aplica sobre las condiciones de los clasificadores, mientras que mutación se aplica sobre todo el clasificador. Si se muta una condición del clasificador, ésta debe de ser ó # ó un valor que aparea la información del ambiente.

Si se verifica por subsumsión, los hijos subsumidos por sus padres no se añaden y se aumenta la numerosidad al padre ($cl.n++$).

También se eliminan individuos seleccionados por ruleta y revisando que la numerosidad de los clasificadores sea mayor que N ($cl.n > N$).

Si el clasificador seleccionado tiene una numerosidad mayor que 1, ésta se decrementa en 1, si es 1, se elimina el clasificador.

El criterio para eliminar clasificadores se hace si el clasificador tiene suficiente experiencia ($cl.exp > \theta_{elim}$) y si su aptitud es menor que el promedio de aptitud de los clasificadores.

Debido a su conección con aprendizaje por refuerzo y análisis más teórico, XCS tiende a producir mejores y más compactos clasificadores.

9.8.4 Otros aspectos a considerar

Dentro de la representación, los clasificadores tienen una representación binaria que no es útil en problemas continuos. Aquí se ha experimentado con la definición de rangos de variables.

También se ha buscado quitar la restricción en la sintáxis de tener una conjunción y se ha experimentado con expresiones en Lisp (muy conectado con programación genética).

Esto crea clasificadores de longitrud variable (e.g., messy genetic algorithms).

XCS se puede utilizar para hacer planificación, fijandose en el siguiente estado que nos de la máxima recompensa.

Para la generalización de los clasificadores se usa un esquema de nichos (niche

GA), en donde, si dos clasificadores son igualmente precisos, se conserva el más general.

El otro proceso es el de eliminación por subsumsión, que verifica si un nuevo clasificador generado por GA es subsumido por otro que tiene un alto desempeño, en cuyo caso se desecha.

Esto provoca poblaciones pequeñas, con conjuntos que no se traslapan, sin embargo, puede crear clasificadores sobre-generalizados.

Esto es un problema en poblaciones de clasificadores pequeñas o cuando es difícil distinguir entre la aptitud de dos clasificadores.

Aunque se tengan poblaciones mayores y mejores funciones de desempeño, se puede sobre-generalizar.

Para esto se compara si su error relativo comparado con el del promedio de la población es mayor a un cierto umbral, en cuyo caso se crea un nuevo clasificador para cubrir el caso actual con una cierta probabilidad fija de generar *don't cares*.

9.8.5 Programación Genética

Computación evolutiva que genera programas (Koza, '92). Osea que lo que se tiene es una población de programas.

Los programas típicamente se representan como árboles de "parseo" de un programa. Donde cada función está representada en un nodo y los argumentos de la función en sus ramas.

Para aplicar programación genética a un dominio se tiene que especificar:

- Las funciones primitivas a considerar (e.g., cos, $\sqrt{}$, $+$, $-$, etc.).
- Los nodos terminales (e.g., $x, y, 2$, etc.).

Se siguen los siguientes pasos:

- Genera una población inicial mediante una composición (generalmente aleatoria) de funciones y símbolos terminales relacioandos al dominio.
- Realiza en forma iterativa los siguientes pasos, hasta llegar a un criterio de terminación:
  - Ejecuta cada programa y asignales un valor de acuerdo a tu función de aptitud.
  - Crea nuevos programas haciendo: (i) Reproducción (copia), (ii) Cruza (intercambio aleatorio de dos partes de un progrma, (iii) Mutación (cambia una parte aleatoria de un programa), y (iv) Operaciones que cambian la arquitectura (ver abajo),
- Selecciona el mejor programa.

Cruza selecciona aleatoriamente nodos padres de dos árboles y los intercambia.

Las operaciones que alteran la "arquitectura" se usan para determinar: (i) el número de funciones a utilizar, (ii) el número de argumentos por función, y (iii) el tipo de jerarquía (quien llama a quien).

9.8.6 Conocimiento del dominio

En principio se ven a los GA como métodos de búsqueda independientes del dominio.

Sin embargo conocimiento del dominio puede incorporarse en:

- Los operadores genéticos a utilizar.
- La representación usada.
- Poblaciones iniciales.
- Retroalimentación.

En la mayoría de los casos, el desempeño de un algoritmo genético depende fuertemente de la representación utilizada y de la selección de la función de aptitud. Un tema de investigación actual es tratar de descubrir automáticamente las primitivas que mejoren el comportamiento de las primitivas originales.

9.8.7 Otros temas

- Evolución Lamarckiana: Lamarck (científico del s XIX) propuso que las expeciencias de un individuo durante su vida pueden afectar genéticamente a sus descendientes. Aunque científicamente esto ha sido rechazado, algunos autores han usado estas ideas dentro de sus algoritmos.

- Efecto Baldwin: Baldwin (1896) propuso que si existen cambios en el ambiente, la evolución tiende a favorecer individuos con la capacidad de aprender a adaptarse al nuevo medio ambiente acelerando cambios genéticos entre individuos parecidos.

- *Crowding*: es el fenómeno cuando algún individuo es mucho más apto que los demás y se reproduce rápidamente llenando la problación con sus copias. Esto reduce diversidad y desacelera rápidamente el proceso evolutivo.

  Se han propuesto varias posibles soluciones cambiando el criterio de selección:

  - Selección por torneo.
  - Selección por ranqueo.
  - "fitness sharing": reducir la aptitud de un individuo por la prescencia de otros individuos parecidos.
  - Sólo combinar individuos muy parecidos entre si, creado "subespecies".

- Paralelización: Se han propuesto (por razones naturales) varios esquemas de paralelización: (i) Grano grueso: varias poblaciones (*demes*) al mismo tiempo con *migración* entre ellas y (ii) Grano fino: individuo por procesador.

9.9 Algoritmos Meméticos

Es un algoritmo poblacional, que puede verse como una variante de los algoritmos genéticos.

La idea básica del algoritmo es la de incorporar la mayor cantidad de conocimiento del dominio que sea posible durante el proceso de generación de una nueva población.

Así como en búsqueda local teniamos definida una vecindad para un solo individuo, la vecindad de una población de individuos se puede obtener mediante la composición de los individuos.

La idea es crear un conjunto de soluciones nuevas a partir de las actuales. Esto se puede hacer identificando y combinano los atributos de las soluciones actuales.

Los operadores de recombinación ciegos, como los usados tradicionalmente por los algoritmos genéticos, no incorporan ningún conocimiento del dominio al momento de generar nuevos individuos, por ejemplo, cruza uniforme.

El argumento principal para no introducir conocimiento es para no sesgar la búsqueda, y evitar convergencias prematuras a soluciones subóptimas, sin embargo, esto último ha sido cuestionado últimamente.

Los operadores que introducen conocimiento se llaman *heuristic* o *hybrid*.

El conocimiento se puede incorporar en:

- La selección de los atributos de los padres que van a ser transmitidos a los hijos.
- La selección de los atributos que no son de los padres que van a ser transmitidos a los hijos.

La tabla 9.4 describe una estructura genérica de un algoritmo poblacional. Se construye inicialmente una población, a partir de ésta, se crea una nueva población, y finalmente de decide que parte se reemplaza de la población

Tabla 9.4: Algoritmo Poblacional Genérico

```
Búsqueda Poblacional
begin
   POBL ← GeneraPoblInic()
   repeat
      POBLNVA ← GenNvaPobl(POBL)
      POBL ← ActualizaPobl(POBL,POBLNVA)
      If POBL convergió
      Then POBL ← reinicia(POBL)
   until criterio de paro
```

anterior para quedarse con una nueva población y repetir el proceso. En caso de converger se puede reiniciar el proceso con otra población.

Para generar una población inicial, se puede hacer en forma aleatoria (mientras sean soluciones factibles), puede aplicarse a ésta búsqueda local, o de plano usar desde el principio GRASP.

Para generar una nueva población (reinicio), se pueden mantener algunos de los mejores individuos (inclusive uno solo) o se puede hacer una mutación muy fuerte.

Los dos tienen sus ventajas y desventajas. Cuando se deja uno o más anteriores, se tiene que tener cuidado de que no invadan rápidamente la población. Cuando se hace por mutción, se puede perder información valiosa con mucha mutación o se puede regresar al punto anterior con poca mutación.

Lo diferente viene en la generación de una nueva población a partir de operadores genéticos. Los algoritmos meméticos utilizan conocimiento del dominio para guiar mejor la búsqueda y decidir qué elementos incorporar y cuáles desechar al momento de crear nuevos individuos.

Los algoritmos meméticos se pueden ver como algoritmos genéticos en donde se introduce conocimiento del dominio para crear una nueva generación de individuos.

Son muy parecidos que scatter-search, pero las formas de crear nuevos algoritmos se basa en operadores genéticos más que en combinaciones lineales de soluciones.

9.10 Poblaciones Estadísticas

La mayor parte de la teoría de los GA se basa en los llamados "bloques constructores".

Opciones para evitar la ruptura de los bloques:

1. Manipular la representación de las soluciones para disminuir las posibles rupturas usando operadores de recombinación
2. Generar nuevas soluciones usando información extraida de todas las soluciones

La información global se puede usar para estimar una distribución y usar esa estimación para generar nuevas soluciones.

Como en otras ocasiones, existe un balance entre la exactitud de la estimación y el costo computacional para realizarla.

Lo más fácil es considerar cada variable independiente de las demás (PBIL).

9.10.1 PBIL (Population-based Incremental Learning)

Usa un vector de probabilidad que al muestrearse produce con alta probabilidad vectores con evaluaciones altas.

Inicialmente todos los lugares toman el valor de 0.5 (prob. de producir un 1).

Se genera un conjunto de individuos tomando en cuenta las probabilidades del vector.

Tabla 9.5: Algoritmo de PBIL.

Algoritmo
for $i := 1$ *to* LONG. *do* $P[i] = 0.5$
while NOT criterio de paro
    for $i := 1$ *to* NUM-MUEST. *do*
        vectores_soluc[i] := genera una muestra con prob. P
        evalaciones[i] := *evalúa*(vectores_soluc[i])
    ordena vectores_soluc
    for $j := 1$ *to* NUM-VECT-ACT. *do*
        for $i := 1$ *to* LONG. *do*
            $P[i] := P[i] * (1 - LR) + vectores\_soluc[j][i] * LR$

Se cambia el vector de probabilidades hacia valores que produjeron mejores resultados (la magnitud del cambio depende de un parámetro).

Se continua con el proceso hasta llegar a un criterio de paro (todos los lugares cercanos a 1 o 0).

Parámetros: (i) número de muestras a generar (e.g., 200), (ii) razón de aprendizaje (LR, e.g., 0.005), y (iii) número de vectores a condiderar para actualizar el vector de probabilidad (e.g., 2). El algoritmo se muestra en la tabla 9.5.

Funciona bien si no hay interacciones significativas entre variables y se aproxima (por lo menos UMDA, una variante de PBIL) al comportamiento de un GA con cruza uniforme.

Variantes:

- Incluir "mutaciones" (alteraciones estocásticas)
- Mover el vector de probabilidad también con ejemplos "negativos"
- Mover el vector de probabilidad de acuerdo a la fuerza de cada individuo
- Utilizar varios vectores y combinarlos

9.10.2 BOA: The Bayesian Optimization Algorithm

Para tratar de capturar interacciones entre variables se han propuesto las siguientes mejoras:

- Interacciones por pares y construcción de un árbol de dependencias (relacionado a Chow y Liu)
- Factorización y descomposición de problemas (se requiere conocimiento del dominio)
- Usar redes bayesianas (BOA) y descubrir la estructura intrínseca

BOA genera genera una población aleatoria, selecciona los mejores individuos, construye una red bayesiana que ajuste esos individuos bajos ciertos criterios, genera nuevos individuos usando la distribución codificada en la red bayesiana y se reemplaza (parte de) la población original. Este proceso se repite hasta cumplir el criterio de terminación.

Algunas de las restricciones que se usan son:

- número de padres máximo
- árbol o poliárbol
- algoritmo tipo hill-climbing
- calidad de la red (usando la métrica Dirichlet Bayesiana)

El algoritmo viene descrito en la tabla 9.6.

Tabla 9.6: Algoritmo de BOA.

$t \leftarrow 0$
Genera aleatoriamente problación inicial $P(0)$
Hasta llegar a criterio de paro:
    Evalúa y selecciona los mejores individuos $S(t)$ de $P(t)$
    Construye una BN (B) bajo ciertas métricas y restricciones
    Genera nuevos individuos ($O(t)$) usando la distrib. de B
    Crea nueva población $P(t+1)$ reemplanzando algunos individuos
        de $P(t)$ con los de $O(t)$
    $t \leftarrow t+1$