ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

TÉCNICAS DE OPTIMIZACIÓN MODERNAS PARA PROBLEMAS COMPLEJOS

Realizado por

JOSÉ FRANCISCO CHICANO GARCÍA

Dirigido por

ENRIQUE ALBA TORRES

Departamento

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA

MÁLAGA, Mayo de 2003

UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente D^o/D^a. _____

Secretario D^o/D^a. _____

Vocal D^o/D^a. _____

para juzgar el Proyecto Fin de Carrera titulado:

TÉCNICAS DE OPTIMIZACIÓN MODERNAS PARA PROBLEMAS COMPLEJOS

del alumno D^o. José Francisco Chicano García

dirigido por D^o. Enrique Alba Torres

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN DE _____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga, a de de 2003

El Presidente	El Secretario	El Vocal
Fdo:	Fdo:	Fdo:

Índice general

Índice de figuras

Índice de tablas

Prólogo

Vivimos en un mundo en el que constantemente tenemos que resolver problemas. A lo largo de la historia la especie humana ha inventado máquinas para facilitarle el trabajo y transformar los problemas en otros más simples. La complejidad de las máquinas ha ido creciendo y con ella la de los problemas que son capaces de resolver. La aparición de los computadores permitió llevar a cabo un acercamiento entre el mundo de las matemáticas y el mundo real. Los algoritmos dejaban de ser un concepto en la mente de los matemáticos para convertirse en una realidad dentro de las nuevas máquinas. Los problemas que se podían resolver crecieron en dimensión. Los ordenadores eran una herramienta para los matemáticos y muchos de los problemas que resolvían habían sido sacados del mundo de las matemáticas.

Poco a poco se fueron usando los computadores para resolver problemas de la vida cotidiana relacionados con los dominios de la ingeniería, las ciencias y el mundo real en general. Con el descenso de su precio los computadores se han introducido en la mayoría de los hogares y los electrodomésticos contienen pequeños microprocesadores con algunos algoritmos implementados que controlan el aparato. No es de extrañar, por tanto, que exista tanto interés en resolver todo tipo de problemas con los computadores, una vez que se ha visto de los que son capaces.

Algunos de esos problemas pueden resolverse fácilmente con un algoritmo específico que hace el trabajo en un tiempo razonable. Para otros, en cambio, no es posible encontrar un algoritmo que lo resuelva en un tiempo aceptable.

Para tratar de resolver estos "problemas complejos" los algoritmos deterministas deben dar paso a los algoritmos estocásticos. Algunos de estos algoritmos llevan entre nosotros casi desde el comienzo de la computación. Otros acaban de surgir.

En este Proyecto Fin de Carrera presentamos algunas técnicas modernas empleadas en

la resolución de problemas del mundo real que tienen una gran complejidad. Describiremos algunos de estos problemas y los resolveremos usando dichas técnicas. Los objetivos del proyecto son:

- Estudiar un amplio abanico de algoritmos clásicos y heurísticos utilizando conocimiento de los problemas abordados. Los algoritmos estudiados han sido Algoritmos Genéticos, Estrategias Evolutivas, Retropropagación, Levenberg–Marquardt, Savings, λ–opt, λ–Intercambio y algoritmos voraces.

- Aplicar dichos algoritmos a un conjunto representativo de problemas en el dominio de las telecomunicaciones, la bioinformática, la ingeniería del software y la optimización combinatoria.

- Tratar de mejorar en la medida de lo posible la mejor marca para los problemas usados, definiendo marca como una mejora en la eficiencia y/o en la precisión o comprensibilidad de la solución final.

- Proponer mejoras y nuevos algoritmos para resolver estos problemas.

- Generar un código multiplataforma eficiente que permita ejecutar estos algoritmos en cualquier máquina. Incluso poder usar redes formadas por máquinas heterogénas para resolver los problemas. Para cumplir este objetivo se ha implementado el software en el lenguaje de programación Java.

- De forma paralela a la evolución del proyecto se han obtenido resultados secundarios de gran importancia. Se ha estudiado el uso en el problema VRP de una representación binaria para las permutaciones con interesantes propiedades y se ha resuelto un problema de aprendizaje consistente en obtener los parámetros de un modelo del crecimiento microbiano en alimentos envasados.

La memoria está estructurada en 7 Capítulos, 3 Apéndices y una Bibliografía.

El Capítulo 1 introduce al lector la motivación de este proyecto.

El Capítulo 2 describe los problemas y las instancias concretas de esos problemas resueltas en el proyecto. Cada problema es descrito en una sección separada.

El Capítulo 3 describe los algoritmos que se han usado para resolver los problemas propuestos. El capítulo está dividido en secciones donde se presentan los diferentes algoritmos.

El Capítulo 4 explica los detalles de la realización de las pruebas con el objetivo de que los resultados sean reproducibles.

El Capítulo 5 muestra los resultados obtenidos aplicando los algoritmos a las instancias de los problemas. Los resultados vienen acompañados de comentarios acerca de ellos.

El Capítulo 6 detalla una serie de resultados secundarios que obtuvimos durante la realización del proyecto y que no forman parte de los objetivos originales del mismo pero que por su importancia y relación merecen atención.

El Capítulo 7 presenta las conclusiones obtenidas de todo lo realizado en el proyecto y menciona la manera de ampliar el trabajo.

El Apéndice A describe el software implementado para trabajar con redes neuronales y los algoritmos de entrenamiento de estas redes. Este capítulo está dividido en secciones que se dedican a cada una de las partes del software de redes.

El Apéndice B describe el software implementado para trabajar con algoritmos evolutivos. Cada componente de un algoritmo evolutivo se analiza en una sección diferente.

El Apéndice C se dedica exclusivamente a estudiar los operadores de los algoritmos evolutivos que han sido implementados. Por la gran cantidad de estos operadores y su importancia en el software de algoritmos evolutivos merecían un Apéndice aparte. Los operadores son organizados en grupos que son analizados en las distintas secciones.

Capítulo 1

Introducción

Los estudios en optimización han sido de gran importancia en Informática a lo largo de toda su historia. Dondequiera que aparezca un problema de búsqueda, optimización o incluso aprendizaje tienen cabida los múltiples análisis existentes sobre el diseño eficiente de algoritmos. La optimización es una rama de trabajo muy dinámica debido a que nos enfrentamos constantemente a nuevos retos: nuevos problemas de ingeniería, nuevas situaciones de la industria, nuevos servicios que necesitan ser optimizados y un largo etcétera de situaciones que desafían constantemente a las técnicas de optimización existentes [RSORS96, MF98].

Dado que se puede demostrar que no existe un resolutor óptimo de problemas de optimización, o lo que es lo mismo, que ninguna técnica es superior a otra cuando se considera el abanico de todos los problemas posibles (*No Free Lunch Theorem* [WM97]) parece lógico intentar utilizar las mejores técnicas existentes en cada dominio como punto de partida o de comparación para el diseño de mecanismos de búsqueda basados en nuevas ideas. Por tanto, además de la gran importancia de la incorporación de conocimiento del problema en la técnica final de resolución [MF98] es de interés profundizar en los conocimientos sobre plantillas genéricas de búsqueda que permitan al menos la hibridación con técnicas de búsqueda local para así rentabilizar el conocimiento.

Un ejemplo distinguido de meta–heurísticos son los algoritmos evolutivos [BFM97], de cuyo formato genérico de búsqueda pueden extraerse numerosas familias de algoritmos parametrizados de flexible adaptación al problema objetivo. La hibridación de estos algoritmos con otros dependientes del dominio de trabajo es de vital importancia para

conseguir un buen *grado de penetración*, es decir, un buen equilibrio entre exploración y explotación que minimice el número de puntos visitados (esfuerzo computacional) y que a la vez no se estanque en óptimos locales y afine la solución final hasta soluciones aceptables para el problema.

Por otro lado, la gran difusión de las redes de ordenadores y de Internet hacen aconsejable abordar algoritmos paralelos tanto en red local (LAN) como en área extensa (WAN), ya que esto permite grandes reducciones en los tiempos de ejecución y llevan a dichos algoritmos hasta la posibilidad de ser ofertados on–line como servicio en Internet a empresas y otras organizaciones. Internet es rica en tecnologías para el diseño cooperativo de algoritmos (XML [Hol01], .NET [Pla01], J2EE [ZU01]) y también para ser explotada como conjunto de máquinas heterogéneas para cálculo (idea que se asemeja al concepto de *grid* [HAFF99]).

Pero el campo algorítmico no es el único que debe ser tenido en cuenta a la hora de realizar estudios de viabilidad o comparativas entre técnicas clásicas y modernas. Es muy importante que los estudios utilicen un conjunto de problemas representativos de interés actual, y además que tengan una complejidad considerable (no únicamente de laboratorio), de manera que el impacto en el conocimiento científico sea innegable.

De esta forma, parece que tres de los campos de gran interés en las líneas de investigación regionales, españolas y europeas son las telecomunicaciones [COS00], la bioinformática [PRS00] y la ingeniería del software [CCZ01]. En este trabajo, además de abordar problemas en estos tres dominios somos conscientes de que en el fondo de muchos de estos problemas existe una base de optimización combinatoria; ésta será analizada aparte a través de la solución de problemas de dificultad notable en relación a la selección de rutas para vehículos (VRP) [LPS99], por tratarse de una extensión compleja y real de otros clásicos como TSP.

Estos cuatro dominios son notables fuentes de problemas de optimización para los que la mayoría de técnicas exactas no son siquiera aplicables, y muchas técnicas heurísticas son claramente mejorables. Existen numerosas comparativas, pero ninguna conjunta todos ellos y con problemas de complejidad elevada. Asimismo, la necesidad de una metodología de estudio y una caracterización de los algoritmos resultantes es muy elevada, ya que esto permitiría predecir su comportamiento y elegir sus parámetros a partir de modelos matemáticos formales.

Capítulo 2

Problemas

En este capítulo presentaremos los problemas y mencionaremos las instancias concretas resueltas en el proyecto. Los problemas abordados podemos clasificarlos dentro de cuatro dominios: bioinformática, telecomunicaciones, ingeniería del software y optimización combinatoria. Al primer dominio pertenecen las instancias que hemos usado del problema de Entrenamiento de Redes de Neuronas Artificiales; al segundo, el diseño de códigos correctores y la asignación de terminales a concentradores; al tercero, la planificacón de tareas en ingeniería del software, y al último el problema de guiado de vehículos. A continuación pasamos a describirlos uno por uno.

2.1 Entrenamiento de Redes de Neuronas Artificiales (ANN)

Una red de neuronas artificiales (o red neuronal) es un procesador masivamente paralelo distribuido que es propenso por naturaleza a almacenar conocimiento experimental y hacerlo disponible para su uso. Este mecanismo se parece al cerebro en dos aspectos: el conocimiento es adquirido por la red a través de un proceso que se denomina aprendizaje y se almacena mediante la modificación de la fuerza o peso sináptico de las distintas uniones entre neuronas [Hay94].

Una red neuronal queda caracterizada por su:

- **Arquitectura**: Estructura o patrón de conexiones entre las unidades de proceso. Se puede caracterizar mediante un grafo.

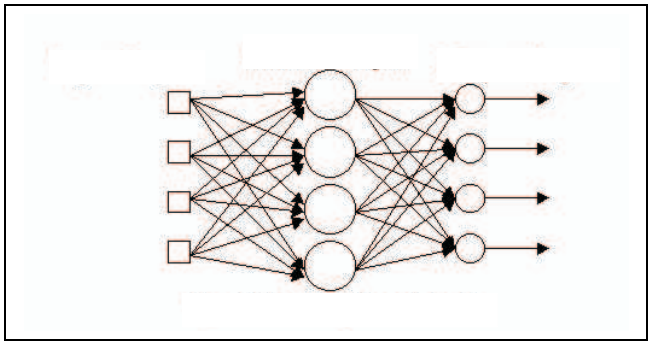

Figura 2.1: Una Red Neuronal.

- **Algoritmo de entrenamiento o aprendizaje**: Procedimiento para determinar los pesos de las conexiones. Podemos destacar dos tipos: supervisado y no supervisado. El primero requiere la existencia de una fuente externa que indique si la salida de la red es correcta o no, y en qué medida se aproxima a la salida deseada. En el segundo tipo la red evoluciona sin ayuda externa.

- **Función de Activación (o de transferencia)**: Función que especifica cómo se transforma la entrada de la unidad de proceso en la señal de salida.

Las unidades de proceso o neuronas toman una serie de entradas procedentes de una fuente externa o de la salida de otras neuronas y la transforman aplicando la función de activación para obtener una salida que será a la vez entrada de otras neuronas o salida de la red. Antes de aplicar la función de activación, se calcula la suma ponderada de las entradas de la neurona y se le resta un valor llamado *umbral* (Figura 2.2). Los factores de ponderación son los llamados *pesos sinápticos*. Durante el proceso de entrenamiento estos pesos serán modificados. Las redes neuronales presentan una serie de entradas y una serie de salidas. Cuando se le aplican valores a las entradas obtenemos valores a la salida. El conjunto de valores de entrada se conoce como vector de entrada y el conjunto de valores de la salida es el vector de salida que proporciona la red para ese valor de entrada.

Nuestro objetivo es entrenar redes neuronales para usarlas en problemas de clasificación. Los problemas de clasificación consisten en determinar la clase a la que pertenece un vector de valores característicos determinado. Numerosos estudios han comparado la capacidad de clasificación de las Redes de Neuronas Artificiales (ANNs) con las técnicas

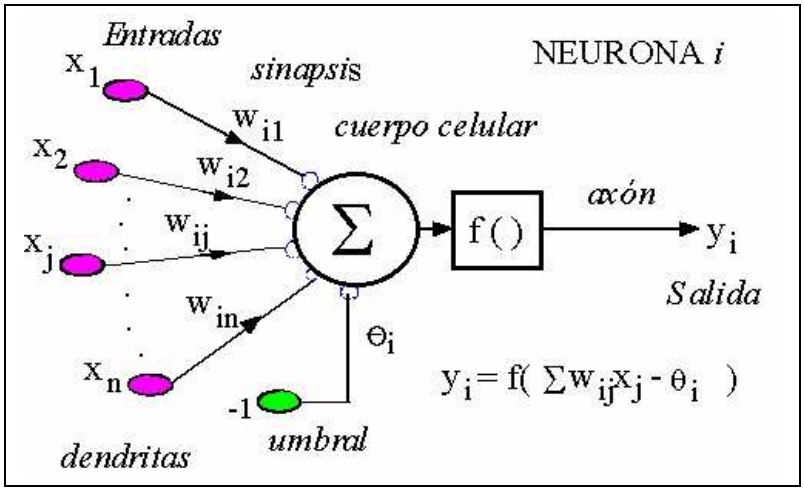

Figura 2.2: Modelo de una neurona.

estadísticas y han llegado a la conclusión de que las ANNs son mejores que las técnicas tradicionales en muchos casos [AW92, HMS92, KWR93, PHH93, SCL92, SHH93, TK92, YSM93].

Usamos aprendizaje supervisado para entrenar ANNs y que aprendan así un conjunto de patrones. Cada patrón está formado por un vector de entrada y un vector deseado de salida. Estos vectores están formados por números reales. Sin embargo, en los problemas de clasificación la salida de la red debe interpretarse como una clase y el número de clases es discreto y finito. Las formas de realizar esta interpretación pueden ser muy diversas [Pre94]. Una de ellas consiste en poner una neurona de salida por cada clase que haya. Cuando la red arroja un vector de salida se busca la neurona de salida con mayor valor y su clase asociada es la que la red asigna al vector de entrada. Para que esto funcione cada patrón debe tener un vector de salida donde sólo hay un elemento con valor máximo (que indica la clase del patrón) y el resto tiene valor mínimo. Esta técnica es conocida como "el ganador se lleva todo" y es la que hemos seguido aquí. Otra técnica, la técnica de los umbrales, consiste en considerar que un patrón está bien clasificado cuando la diferencia entre las salidas de la red y las salidas deseadas en valor absoluto no superan un cierto valor umbral. La codificación del vector deseado de salida de los patrones debe ser igual que en la técnica anterior.

Las instancias usadas pueden obtenerse del *Repositorio UCI de Aprendizaje Máquina*

(`http://www.ics.uci.edu/ mlearn/MLRepository.html`) [SD00]. Son seis que vamos a describir a continuación.

- **Cancer** (BC): Consiste en diagnosticar el cancer de mama. Hay que clasificar un tumor como benigno o maligno a partir de descripciones de las células obtenidas mediante observación microscópica. El vector de entrada está formado por nueve parámetros de las células. Hay 699 patrones que fueron obtenidos por el Doctor William H. Wolberg en los Hospitales de la Universidad de Wisconsin, Madison [Wol90, WM90, MSW90, BM92].

- **Diabetes** (DI): Consiste en diagnosticar la diabetes en los Indios Pima. Hay que dar un diagnóstico positivo o negativo en función de ocho parámetros de los individuos. Hay 768 patrones pertenecientes al *National Institute of Diabetes and Digestive and Kidney Diseases* y donadas al repositorio por Vincent Sigillito.

- **Heart** (HE): Consiste en predecir una dolencia de corazón. Hay que decidir si al menos una de las cuatro cámaras del corazón se ha reducido en diámetro más del 50%. Esta decisión se hace basada en trece datos pertenecientes a la persona. Hay 920 patrones que fueron recogidos de cuatro fuentes distintas. Las instituciones y los investigadores responsables de tal recolección son:

  - Hungarian Institute of Cardioligy, Budapest: Andras Janosi, M. D.
  - University Hospital, Zurich, Switzerland: William Steinbrunn, M. D.
  - University Hospital, Basel, Switzerland: Matthias Pfisterer, M. D.
  - V.A. Medical Center, Long Beach and Cleveland Clinic Foundation: Robert Detrano, M. D., Ph.D.

- **Gene** (GE): Consiste en detectar las lindes entre intrones y exones en secuencias de nucleótidos. Dada una ventana de 60 nucleótidos sacados de una secuencia de ADN hay que decidir si la mitad representa un paso de intrón a exón, de exón a intrón o ninguna de ellas. En este problema la entrada es un vector de 60 nucleótidos (hay cuatro tipo de nucleótidos en el ADN: Adenina, Citosina, Guanina y Timina). Hay 3175 patrones donados por G. Towell, M. Noordewier y J. Shavlik y tomados de *GenBank 64.1*(`ftp://genbank.bio.net`).

- **Soybean** (SO): Consiste en reconocer 19 enfermedades diferentes de las semillas de soja. La decisión debe tomarse basada en la descripción de la semilla y la planta y en el historial de la vida de la planta. El número total de atributos es de 35. Los donadores de este conjunto de 683 patrones son Ming Tan y Jeff Schlimmer.

- **Thyroid** (TH): Consiste en diagnosticar la hiper e hipofunción del tiroides. Basado en el examen y consulta del paciente hay que decidir si el tiroides del paciente es hiperfuncionante, normal o hipofuncionante. Se usan 21 atributos del paciente. Hay 7200 patrones donados por Randolf Werner y obtenidos de Daimler–Benz.

Estos son los problemas de clasificación abordados. Todos pueden encontrarse en el repositorio mencionado. Sin embargo, a la hora de usar los conjuntos de datos nos encontramos con que debemos preprocesarlos antes de dárselos a la red. Esto es debido a que muchos de los atributos de entrada son nominales (no son números) o se encuentran en un rango no adecuado. Este preprocesamiento puede ser llevado a cabo de muy diversas formas y puede influir en el resultado final de la clasificación, de modo que resultados de distintos investigadores no podrían compararse. Para evitar esto Prechelt en [Pre94] propone una serie de reglas que deben ser seguidas para poder comparar la bondad de distintas técnicas de clasificación. Algunas de esas reglas son: tener los patrones en el mismo orden, codificar los atributos de cierta forma, especificar claramente los patrones usados para el entrenamiento y para los tests, etc. Además, en el mismo documento, propone una serie de problemas para realizar las comparativas. A esos problemas Prechelt les aplica sus reglas y como resultado procesa cada conjunto de patrones para obtener un nuevo conjunto que puede ser usado directamente para entrenar y testear una red neuronal. El benchamrk llamado PROBEN1 y formado por diez problemas de clasificación (con cuatro variantes para el problema **heart**) y tres de aproximación funcional puede obtenerse via **ftp** anónimo en `ftp://ftp.ira.uka.de/pub/neuron/proben1.tar.gz`. Los seis problemas anteriores pertenecen a ese benchmark. En el proyecto no hemos usado los conjuntos de datos que pueden obtenerse del repositorio directamente, sino que hemos usado los conjuntos preprocesados de Prechelt. Los detalles del preprocesamiento de los datos de estos problemas se encuentran en el Capítulo 4.

2.2 Diseño de Códigos Correctores de Errores (ECC)

Entre las primeras labores del diseño de un sistema de comunicación se encuentra la confección de un código capaz de transmitir los mensajes de forma fiable y lo más rápido posible. Por un lado, interesa reducir la longitud de las palabras del código para poder transmitir los mensajes rápidamente. Por otro, la distancia de Hamming entre las palabras debe ser lo más alta posible para asegurar la corrección en el receptor. Como suele ocurrir en optimización, estos dos objetivos se contraponen y tendremos que llegar a un resultado de compromiso.

Hay muchos tipos de códigos correctores de errores; nosotros nos centraremos en los códigos binarios lineales de bloques. Estos códigos pueden ser representados con un vector de tres elementos (n, M, d), donde n es el número de bits de cada palabra del código, M es el número de palabras en el código y d la mínima distancia de Hamming entre un par cualquiera de palabras distintas del código.

La idea subyacente en la corrección de una palabra es la siguiente: si todas las palabras están separadas como mínimo por d bits, cualquier modificación de hasta $(d-1)/2$ bits en una palabra válida puede ser corregida tomando la palabra más parecida. En la Figura 2.3 vemos una interpretación gráfica del proceso. La palabra de código W es recibida por un canal de comunicación. Esa palabra no se corresponde con ninguna del código, sin embargo, la distancia de Hamming entre ella y la palabra $C(i)$, que sí pertenece al código, es menor que $(d-1)/2$ bits y se asume que $C(i)$ era la palabra que se envió originalmente por el canal. Por tanto, si se recibe W se sustituye por $C(i)$. Así es como se utilizan estos códigos correctores de errores. Si una palabra en su camino hacia el destino es transformada y más de $(d-1)/2$ bits resultan alterados el mecanismo no asegura que se tome la palabra correcta al corregir el error. Dependiendo del ruido que haya en el canal tendremos que escoger un código con mayor o menor distancia de Hamming entre palabras.

El problema que aquí nos planteamos es el siguiente. Dados el número de bits y de palabras (n y M) deseamos encontrar un código cuya distancia mínima de Hamming d sea la más alta posible. Hay estudios teóricos que establecen las relaciones numéricas existentes entre n, M y d [AVZ01], sin embargo, no existe una forma eficiente de encontrar esos códigos en el caso general. En el proyecto abordamos el problema de encontrar un

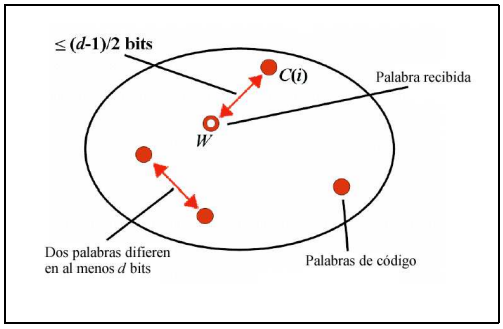

Figura 2.3: Interpretación gráfica de un código corrector.

código con estas características para $n = 12$ y $M = 24$ [CFW98]. Se sabe que la distancia máxima para un código con esos parámetros es $d = 6$ [AVZ01].

2.3 Asignación de Terminales a Concentradores (TA)

Este problema consiste en minimizar el coste de conectar un conjunto de terminales a un conjunto de concentradores para crear una red de comunicación. Cada terminal debe estar conectado a uno y sólo un concentrador. Asociado con cada terminal existe una capacidad requerida por ese terminal y asociado a cada concentrador existe una capacidad soportada por dicho concentrador. La suma de las capacidades requeridas por los terminales conectados a un concentrador no puede superar la capacidad soportada por el concentrador. Asociado a cada par terminal–concentrador existe un coste, que es el correspondiente a asignar el terminal a dicho concentrador. El coste de una solución es la suma de todos los costes de las asignaciones de terminales a concentradores [ASW94]. El problema puede formalizarse como sigue:

Encontrar x_{ij} para minimizar la expresión

$$\sum_{j=1}^{C} \sum_{i=1}^{T} c_{ij} x_{ij} \tag{2.1}$$

sujeto a:

$$\forall i \in \{1, 2, \ldots, T\} \sum_{j=1}^{C} x_{ij} = 1 \tag{2.2}$$

$$\forall j \in \{1, 2, \ldots, C\} \sum_{i=1}^{T} w_i x_{ij} \leq v_j \tag{2.3}$$

donde C es el número de concentradores, T el número de terminales, c_{ij} es el coste de asignar el terminal i al concentrador j, w_i es la capacidad requerida por el terminal i, v_j es la capacidad soportada por el concentrador j, x_{ij} es 1 si el terminal i está conectado al concentrador j y 0 en otro caso. La primera restricción (Ecuación 2.2) asegura que cada terminal está conectado a un solo concentrador. La segunda (Ecuación 2.3) asegura que la suma de las capacidades requeridas por los terminales conectados a un concentrador no supera la capacidad soportada por el concentrador.

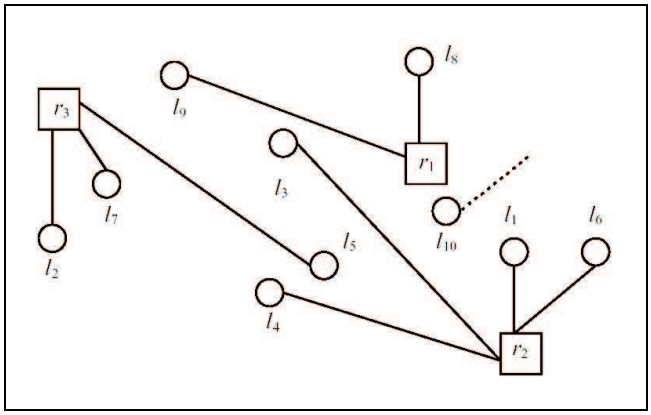

Figura 2.4: Un ejemplo de asignación.

En nuestro caso, hemos resuelto el problema para las 10 instancias de 100 terminales que son usadas en [ASW94]. En ellas los terminales y concentradores se encuentran situados en un plano bidimensional y el coste asociado a cada par terminal–concentrador es la parte entera de la distancia euclídea entre el terminal y el concentrador. Los detalles de estas instancias se pueden encontrar en el Capítulo 4.

2.4 Ingeniería del Software (SWENG)

La gestión de un proyecto software implica la programación, planificación, monitorización, y control de personal, procesos, y recursos para conseguir objetivos específicos, a la vez que hay que satisfacer una serie de restricciones. De forma general, el problema de

la programación de tareas con restricciones de recursos consiste en, dado un conjunto de tareas, recursos, y el modo de evaluar el rendimiento, obtener el mejor modo de programar la asignación de recursos a las actividades de forma que el rendimiento sea máximo [CCZ01]. Las tareas pueden ser cualquier cosa, desde mantener documentos hasta escribir una clase en C++. Los recursos incluyen al personal, tiempo, habilidades, equipo, e instalaciones. Los objetivos típicos de la gestión de proyectos incluyen minimizar la duración del proyecto y el coste del mismo y maximizar la calidad del producto. La programación de tareas describe cómo y cuándo se harán las tareas.

En general, el problema es NP–completo y su resolución mediante técnicas exactas necesita una cantidad de tiempo que puede ser prohibitiva. Para especificar una instancia del problema es necesario que en ella se incluya la siguiente información:

- Una descripción de las tareas y sus requisitos.
- Las relaciones existentes entre las tareas.
- Una descripción de los recursos disponibles para realizar las tareas.
- Un conjunto de objetivos que se usarán para evaluar la programación resultante.
- Una especificación de cualesquiera restricciones que el proyecto deba satisfacer.

La relación entre tareas se puede representar mediante un Grafo de Precedencia de Tareas (Task Precedence Graph, TPG). Un TPG es un grafo acíclico dirigido que consiste en un conjunto de tareas $V = \{T_1, T_2, \ldots, T_n\}$ y un conjunto de precedencias $P = \{(P_{ij}), i \neq j, 1 \leq i \leq n, 1 \leq j \leq n\}$, donde $P_{ij} = 1$ si la tarea i debe completarse, sin que intervenga ninguna otra tarea entre medio, para que pueda comenzar la tarea j y $P_{ij} = 0$ en caso contrario (ver Figura 2.5). Asociado con cada tarea hay un esfuerzo estimado y una serie de habilidades necesarias para realizar la tarea. Los recursos incluyen personal, normalmente descrito como número de horas, equipo e instalaciones. La relación entre los recursos y las tareas es de muchos a muchos. Un recurso puede asignarse a varias tareas y una tarea puede ser realizada por mucho recursos. En nuestro caso, el único recurso que consideramos es el personal. Cada empleado tiene una serie de habilidades, un salario y una dedicación máxima. Cada tarea tiene unas habilidades

requeridas y un esfuerzo que viene medido en personas–mes. Además, las tareas están relacionadas entre sí por el TPG. Los objetivos que considerados en la búsqueda de una solución son los siguientes:

1. Validez de la asignación de trabajos. Para que una asignación sea válida debe cumplir:

   - *Posesión de habilidades*: La unión de las habilidades que poseen los empleados asignados a una tarea contiene al conjunto de habilidades requeridas por esa tarea.
   - *Completitud*: Todas las tareas tienen asignadas al menos un empleado.

2. Nivel mínimo de sobrecarga. El trabajo extra de los empleados es sumado para todos los empleados y debe ser minimizado. El trabajo extra de un empleado es el tiempo que dedica al proyecto por encima de su dedicación máxima.

3. Mínimo coste. El coste total del proyecto debe ser mínimo.

4. Mínima duración del proyecto. El tiempo total requerido para llevar a cabo el proyecto debe ser mínimo.

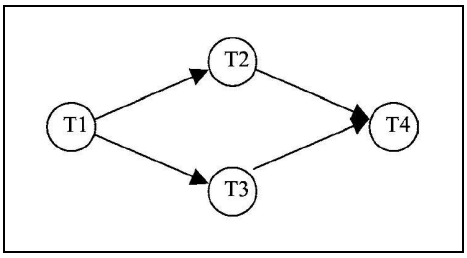

Figura 2.5: Un ejemplo de grafo de precedencia de tareas (TPG).

Los detalles sobre cómo calcular cada uno de los elementos necesarios para evaluar una solución se encuentran en el Capítulo 4. La instancia que hemos usado se corresponde con la primera instancia empleada en [CCZ01] (pág. 121) compuesta por 18 tareas y con 10 empleados.

2.5 Guiado de Vehículos (VRP)

Las empresas que ofrecen servicios de reparto deben plantearse cómo van a realizarlo usando los recursos de que disponen y tratando de minimizar costes. Este problema puede ser descrito como un problema de guiado de vehículos (Vehicle Routing Problem, VRP). En VRP tenemos un conjunto de clientes que hay que visitar para suministrarles mercancía procedente de un almacén. Para hacerlo se emplean vehículos que, en el caso más general, poseen restricciones de capacidad (existe un límite en la cantidad de mercancía que pueden llevar), distancia (la distancia recorrida por el vehículo entre dos paradas en el almacén es limitada) y tiempo (el tiempo transcurrido entre la salida y la llegada del vehículo al almacén está acotado). La solución al problema es un conjunto de rutas que parten del almacén y llegan a él. Cada ruta será recorrida por un vehículo y cada cliente formará parte de una ruta y sólo una (puesto que debe ser visitado por un vehículo exactamente).

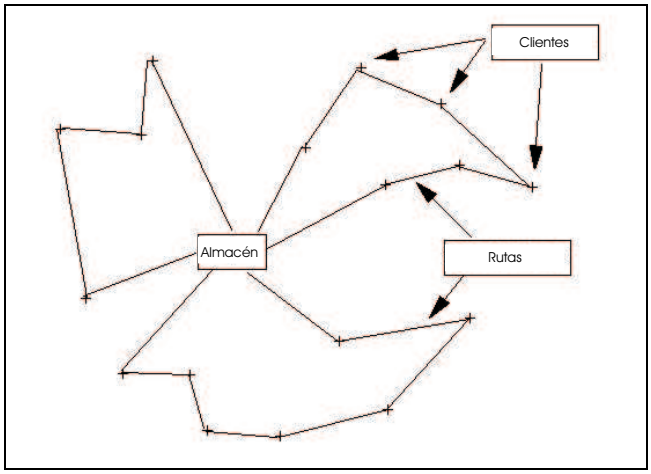

Figura 2.6: Una solución del VRP.

La versión del problema con la que se ha trabajado se puede formalizar como sigue. Tenemos un grafo $G = \{V, E\}$, donde el conjunto de vértices V es de la forma $V = \{0, 1, 2, ..., n\}$. Cada vértice del grafo representa un cliente, a excepción del vértice 0 que representa el almacén de donde parten los vehículos. Asociado a cada arco $(i, j) \in E$ tenemos un coste c_{ij} y asociado a cada vértice i tenemos una demanda m_i y un coste de servicio f_i. Cada vehículo k tiene una capacidad máxima q_k y un coste máximo de viaje r_k. Una ruta es una secuencia alternante de vértices y arcos que comienza y acaba en

vértice y donde cada arco tiene por extremos los vértices adyacentes en la secuencia. El coste de una ruta es la suma de los costes c_{ij} de los arcos de la ruta más los costes de servicio f_i de sus vértices. La demanda de una ruta es la suma de las demandas m_i de los vértices que aparecen en la ruta (el almacén por definición tiene demanda 0). El coste de una solución es la suma de los costes de las rutas que forman parte de la solución. El problema consiste en encontrar un conjunto de rutas que partan y lleguen al almacén (el vértice inicial y final de cada ruta es 0), minimicen el coste total y cumplan las siguientes restricciones:

- La demanda de cada ruta no excede la capacidad (q_k) del vehículo que la recorre.
- El coste de cada ruta es menor que el coste máximo de viaje r_k del vehículo que la recorre.

Este problema está a la orden del día en las empresas que ofrecen un servicio de reparto. Normalmente, estas empresas están interesadas en minimizar el gasto de combustible de sus vehículos. Desafortunadamente, el problema es NP–completo, lo cual significa que los algoritmos para encontrar la solución óptima tienen complejidad exponencial. Debido a esto hay que recurrir a algoritmos heurísticos para encontrar el óptimo o a veces simplemente un valor sub–óptimo pero aceptable. En el proyecto hemos resuelto las 14 instancias de Christofides [CMT79]. En el Capítulo 4 daremos más detalles sobre ellas.

Capítulo 3

Algoritmos

En este capítulo presentaremos los algoritmos que han sido usados para resolver los problemas propuestos. Los problemas ECC y SWENG fueron resueltos exclusivamente mediante algoritmos evolutivos. Para el problema de entrenamiento de redes neuronales fueron empleados además los algoritmos de retropropagación (Backpropagation) y Levenberg–Marquardt [HM94]. Para el caso del VRP se empleó además el algoritmo Savings junto con λ–opt [LGPS99]. Para el TA se usó también el algoritmo greedy propuesto en [KC97]. Uno de los objetivos del proyecto es crear nuevos algoritmos para mejorar los resultados obtenidos con estos.

3.1 Algoritmo de Retropropagación

Podemos clasificar las redes neuronales de acuerdo a su arquitectura según dos tipos: redes neuronales con *alimentación directa* y redes neuronales *recurrentes*. En las primeras, la salida de las neuronas no influye en ninguna de sus entradas. En las segundas hay al menos una neurona cuya salida afecta a alguna de sus entradas. La forma habitual de representar la arquitectura de la red es mediante un grafo dirigido donde los vértices representan a las neuronas y un arco (n_i, n_j) indica que la salida de la neurona n_i es una entrada de la neurona n_j. Las redes con alimentación directa son aquellas cuyo grafo es acíclico mientras que en las recurrentes hay al menos un ciclo. Nosotros nos centraremos en el primer tipo de redes, en concreto en el *perceptrón multicapa generalizado* [YL97].

En el aprendizaje supervisado tenemos un conjunto de patrones. Cada patrón es un

par entrada/salida que la red debe aprender. El objetivo del aprendizaje supervisado es ajustar los pesos sinápticos de la red para que la salida de ésta coincida con la del patrón cuando se le presenta la entrada. Para medir el error que comete la red se usa la suma del error cuadrático que se calcula sumando los cuadrados de las diferencias entre la salida deseada y la obtenida.

El algoritmo de Retropropagación (Backpropagation, BP) es un algoritmo de aprendizaje supervisado para redes neuronales basado en el gradiente del error de la red con respecto a los pesos (y umbrales). Para poder escribir las ecuaciones del algoritmo vamos a introducir antes algo de notación.

Un perceptrón multicapa generalizado está formado por un conjunto de neuronas conectadas mediante arcos, formando un grafo que vamos a llamar $R = (N, A)$. El conjunto de neuronas N son los vértices del grafo y el conjunto de arcos A indican las conexiones entre las neuronas. Notaremos la neurona i-ésima mediante n_i. Existe en subconjunto de neuronas en N que son neuronas de entrada y que tienen un comportamiento especial que más adelante describiremos. Existe también un subconjunto de neuronas que son de salida. Asociado a cada neurona n_i que no sea de entrada existe un valor *umbral* que denotaremos por θ_i y una función de activación que denotaremos por f_i. Asociado a cada arco de A existe un valor real llamado *peso sináptico*. Con w_{ij} denotamos el peso sináptico asociado al arco que va de la neurona n_j a la n_i si tal arco existe. En las redes con alimentación directa (como la que nos ocupa) podemos encontrar una numeración de las neuronas tal que no existan arcos (n_j, n_i) con $j \geq i$. Asumimos que nuestra numeración de las neuronas es de ese tipo. A las neuronas de entrada les asignaremos el índice más bajo. Así, si decimos que una red tiene dos neuronas de entrada, sabemos que son las neuronas n_1 y n_2. Entre las neuronas de entrada no puede haber ninguna conexión. A las neuronas de salida, por el contrario, se les asignará el índice más alto y puede haber conexión entre ellas. La red toma un vector de entrada y realiza cálculos para obtener un vector de salida. La forma en que lo hace la describimos a continuación. Cada neurona n_i tiene una salida x_i cuyo valor se calcula de forma distinta si la neurona es de entrada o no. Para la neurona de entrada n_i la salida x_i es la componente i-ésima del vector de entrada que toma la red. Para el resto de las neuronas la salida se calcula mediante la expresión:

$$x_i = f_i(h_i) \tag{3.1}$$

con

$$h_i = \sum_{j \in Pred(i)} w_{ij} x_j - \theta_i \tag{3.2}$$

donde $Pred(i)$ denota al conjunto de los índices de las neuronas predecesoras a n_i en el grafo R. El valor h_i es la suma ponderada de las entradas menos el umbral y recibe el nombre de potencial sináptico de la neurona. Una vez calculadas las x_i de todas las neuronas de la red se forma el vector de salida usando los valores de las neuronas de salida. Llamaremos $\mathbf{o}$ al vector de salida de la red y denotaremos sus componentes con subíndices. La expresión del error cuadrático de la red para un solo patrón es:

$$E = \frac{1}{2} \sum_{i=1}^{S} (t_i - o_i)^2 \tag{3.3}$$

donde el vector $\mathbf{t}$ es el vector deseado de salida y S es el número de neuronas de salida.

Normalmente trabajamos con conjuntos de patrones y nos interesa minimizar el error para el conjunto completo de patrones (entrenamiento por lotes). En tal caso el error de la red para el conjunto de patrones se mide mediante la expresión:

$$E = \frac{1}{2} \sum_{p=1}^{P} \sum_{i=1}^{S} (t_i^p - o_i^p)^2 \tag{3.4}$$

donde los vectores $\mathbf{o}^p$ y $\mathbf{t}^p$ denotan la salida de la red y la salida deseada para el patrón p y P es el número de patrones.

Podemos destacar dos modos de funcionamiento del algoritmo: entrenamiento individualizado y entrenamiento por lotes. En el primero se trabaja patrón a patrón modificando los pesos y umbrales de la red para tratar de reducir su error (Ecuación 3.3). En el segundo se modifican los pesos y umbrales para reducir el error de todos los patrones (Ecuación 3.4). En este último caso los pesos y umbrales se actualizan de acuerdo a las siguientes ecuaciones:

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \tag{3.5}$$

$$\theta_i \leftarrow \theta_i + \Delta\theta_i \tag{3.6}$$

con

$$\Delta w_{ij} = \eta \sum_{p=1}^{P} \delta_i^p x_j^p \tag{3.7}$$

$$\Delta\theta_i = \eta \sum_{p=1}^{P} \delta_i^p \tag{3.8}$$

donde para las neuronas de salida

$$\delta_i^p = (t_i^p - o_i^p) f_i'(h_i^p) \tag{3.9}$$

y para el resto de neuronas

$$\delta_i^p = f_i'(h_i^p) \sum_{k \in Suc(i)} w_{ki} \delta_k^p \tag{3.10}$$

El valor h_i^p es el potencial sináptico de la neurona n_i para el patrón p y $Suc(i)$ denota al conjunto de los índices de las neuronas sucesoras a n_i en el grafo R. El valor η es la denominada *tasa de aprendizaje* y regula cuánto avanzamos en el sentido opuesto al gradiente. Un valor alto hace que el aprendizaje sea más rápido: el cambio en los pesos es mayor. Un valor pequeño hace que el aprendizaje sea más lento: el cambio en los pesos es menor. A este algoritmo se le conoce como *regla delta*.

Como puede comprobarse, el cálculo de δ_i^p para neuronas que no son de salida requiere conocer δ_j^p, siendo n_j una neurona sucesora de n_i. Por este motivo, para calcular las δ hay que comenzar por las neuronas de salida y seguir hacia atrás (de ahí que el algoritmo se llame retropropagación). Para las neuronas de entrada no hay que calcular δ.

La regla delta se encuentra con el problema de que a veces un valor de η muy alto puede hacer que la red se vuelva inestable (oscilatoria), mientras que un valor demasiado bajo incrementa en exceso el número de épocas de entrenamiento. Para aliviar este problema surgió el *aprendizaje con momentos* o *regla delta generalizada* que consiste en añadir un término con efecto estabilizador en las Ecuaciones 3.7 y 3.8. Las nuevas Ecuaciones son:

$$\Delta w_{ij}(t+1) = \alpha \Delta w_{ij}(t) + \eta \sum_{p=1}^{P} \delta_i^p x_j^p \tag{3.11}$$

$$\Delta \theta_i(t+1) = \alpha \Delta \theta_i(t) + \eta \sum_{p=1}^{P} \delta_i^p \tag{3.12}$$

donde α es la *constante de momentos*. Debe cumplirse $0 \leq |\alpha| < 1$. Puede verse que ahora el cambio en los pesos (y los umbrales) no sólo depende del gradiente de la función de error, sino que también depende del cambio que se produjo en la época anterior. Con la inclusión del término momento el algoritmo de retropropagación tiende a acelerar la bajada en las direcciones de descenso constante (cuando la función de error no cambia mucho), mientras que si el gradiente cambia en iteraciones sucesivas, las modificaciones en los pesos se hacen pequeñas.

El pseudocódigo del algoritmo de retropropagación se puede ver en la Figura 3.1. Hemos llamado s_i^p al elemento i-ésimo del vector de entrada del patrón p-ésimo. Debemos tener en cuenta que o_i es la salida de la i-ésima neurona de salida, que también puede denotarse con $x_{neurons-S+i}$. Lo primero que hace el algoritmo es inicializar los pesos de la red y poner a cero los incrementos de los pesos y los umbrales. Dentro del bucle, lo primero que hace es inicializar con cero los elementos de los arrays donde almacenará el gradiente del error. Luego entra en un bucle cuyo cuerpo se ejecuta para cada patrón. Calcula la salida de la red y los valores de delta, para después actualizar el array del gradiente sumándole el gradiente del error para el patrón en consideración. Tras salir del bucle actualiza los incrementos de los pesos y umbrales para después hacer lo mismo con los propios pesos y umbrales. Esto se repite hasta que se da cierta condición de parada. Las condiciones de parada más habituales son: alcanzar un número de épocas, reducir el error en el conjunto de patrones de entrenamiento hasta un cierto valor o reducir dicho error para un conjunto distinto de patrones llamado conjunto de *validación*.

3.2 Algoritmo Levenberg–Marquardt

Una vez fijo el conjunto de patrones con el que vamos a entrenar la red la fórmula del error de la red E (Ecuación 3.4) tan sólo depende de los pesos y los umbrales. A partir de

```
inicializa_pesos;
Δw_ij := 0 // Se inicializan los incrementos
Δθ_i := 0
MIENTRAS NO condición_parada HACER
    PARA i := 1..neurons HACER
        PARA j := 1..i HACER
            wght[i][j] := 0;
        FINPARA;
        thr[i] := 0;
    FINPARA;

    PARA p := 1..P HACER
        PARA i := 1..neurons HACER // Se aplica el patrón p a la entrada
            SI n_i es de entrada ENTONCES
                x_i := s_i^p;
            SINO
                h_i := Σ_{j∈Pred(i)} w_ij x_j − θ_i;
                x_i := f_i(h_i);
            FINSI;
        FINPARA;

        PARA i := neurons..1 HACER // Se calculan los δ
            SI n_i es de salida ENTONCES
                δ_i := (t_i^p − o_i) f_i'(h_i);
            SINO SI n_i no es de entrada ENTONCES
                δ_i := f_i'(h_i) Σ_{k∈Suc(i)} w_ki δ_k;
            FINSI;
        FINPARA;

        PARA i := 1..  neurons HACER // Se calcula actualiza gradiente
            PARA j := 1..i HACER
                wght[i][j] := wght[i][j] + δ_i x_j;
            FINPARA;
            thr[i] := thr[i] + δ_i;
        FINPARA;
    FINPARA;

    PARA i := 1..neurons HACER // Actualiza los pesos e incrementos
        PARA j := 1..i HACER
            Δw_ij := α · Δw_ij + η·wght[i][j];
            w_ij := w_ij + Δw_ij;
        FINPARA;
        Δθ_i := α · Δθ_i + η·thr[i]
        θ_i := θ_i + Δθ_i;
    FINPARA;
FINMIENTRAS;
```

Figura 3.1: Algoritmo de Retropropagación (BP).

ahora llamaremos $\mathbf{w}$ al vector que contiene a los pesos y umbrales de la red[1]. El algoritmo de retropropagación calcula el gradiente del error con respecto a los pesos y umbrales, es decir, $\nabla E(\mathbf{w})$ y actualiza los pesos e incrementos en función de ese vector. Con la nueva

[1]Consideramos que los vectores son de tipo columna

notación las Ecuaciones 3.5, 3.6, 3.11 y 3.12, pueden resumirse en las dos siguientes:

$$\Delta \mathbf{w}(t+1) = \alpha \Delta \mathbf{w}(t) - \eta \nabla E(\mathbf{w}(t)) \tag{3.13}$$

$$\mathbf{w}(t+1) \leftarrow \mathbf{w}(t) + \Delta \mathbf{w}(t+1) \tag{3.14}$$

Hagamos por un momento $\alpha = 0$ en la Ecuación 3.13. Entonces tenemos la regla delta y el incremento en el vector de pesos se calcula mediante la expresión $\Delta \mathbf{w} = -\eta \nabla E(\mathbf{w})$. Esta expresión procede de una aproximación de la función de error mediante el primer término del desarrollo en series de Taylor esto es:

$$E(\mathbf{w} + \Delta \mathbf{w}) \simeq E(\mathbf{w}) + \nabla E(\mathbf{w})^T \Delta \mathbf{w} \tag{3.15}$$

A partir de esa ecuación se elige un valor para $\Delta \mathbf{w}$ que tenga la misma dirección y sentido opuesto a $\nabla E(\mathbf{w})$ para que el error disminuya. Si en lugar de realizar una aproximación lineal de la función de error realizamos una cuadrática usando los dos primeros términos del desarrollo en series de Taylor obtenemos la expresión:

$$\Delta E(\mathbf{w}) \simeq \nabla E(\mathbf{w})^T \Delta \mathbf{w} + \frac{1}{2} \Delta \mathbf{w}^T H(\mathbf{w}) \Delta \mathbf{w} \tag{3.16}$$

donde $H(\mathbf{w})$ es la matriz Hessiana de E con respecto a $\mathbf{w}$ y $\Delta E(\mathbf{w}) = E(\mathbf{w}+\Delta \mathbf{w}) - E(\mathbf{w})$. Diferenciando con respecto a $\Delta \mathbf{w}$, igualando a cero y despejando tenemos:

$$\Delta \mathbf{w} = -H^{-1} \nabla E(\mathbf{w}) \tag{3.17}$$

La Ecuación anterior nos permite calcular el valor de $\Delta \mathbf{w}$ que minimiza el cambio $\Delta E(\mathbf{w})$. Este es el método de Newton, que consigue en muchos casos mejores resultados que la retropropagación pero requiere calcular derivadas de segundo orden. Existe una aproximación al método de Newton que no requiere dicho cálculo y que es conocido como Levenberg–Marquardt [HM94]. Es más potente que el método del gradiente descendiente pero requiere más memoria. Llamaremos $\mathbf{e}^p(\mathbf{w})$ al vector de error de la red para el patrón p cuando los pesos tienen valor $\mathbf{w}$, esto es:

$$e_i^p(\mathbf{w}) = t_i^p - o_i^p(\mathbf{w}) \tag{3.18}$$

Nótese que ahora indicamos explícitamente que el valor de la salida de la red depende del valor de los pesos sinápticos (el hecho de que antes no se indicara esta dependencia no significa que no existiera). Podemos escribir lo siguiente:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{p=1}^{P} \mathbf{e}^p(\mathbf{w})^T \mathbf{e}^p(\mathbf{w}) \tag{3.19}$$

$$\nabla E(\mathbf{w}) = \sum_{p=1}^{P} J^p(\mathbf{w})^T \mathbf{e}^p(\mathbf{w}) \tag{3.20}$$

$$H(\mathbf{w}) = \sum_{p=1}^{P} \Big(J^p(\mathbf{w})^T J^p(\mathbf{w}) + S^p(\mathbf{w}) \Big) \tag{3.21}$$

donde $H(\mathbf{w})$ es la matriz Hessiana de E evaluada en $\mathbf{w}$, $S^p(\mathbf{w})$ es una matriz dependiente de las segundas derivadas de $\mathbf{e}(\mathbf{w})$ y $J^p(\mathbf{w})$ es la matriz Jacobiana de $\mathbf{e}^p$ evaluada en $\mathbf{w}$, la cual puede expresarse como:

$$J^p(\mathbf{w}) = \begin{bmatrix} \nabla e_1^p(\mathbf{w}) \\ \vdots \\ \nabla e_S^p(\mathbf{w}) \end{bmatrix} = \begin{bmatrix} \frac{\partial e_1^p}{\partial w_1}(\mathbf{w}) & \cdots & \frac{\partial e_1^p}{\partial w_Q}(\mathbf{w}) \\ \vdots & \ddots & \vdots \\ \frac{\partial e_S^p}{\partial w_1}(\mathbf{w}) & \cdots & \frac{\partial e_S^p}{\partial w_Q}(\mathbf{w}) \end{bmatrix} \tag{3.22}$$

donde Q es el número de pesos más el de umbrales. Ahora tomamos

$$\Delta\mathbf{w} = -\alpha M(\mathbf{w}) \nabla E(\mathbf{w})$$

para modificar los pesos. El valor de la matriz M usado por el método es $M(\mathbf{w}) = [\mu I + H(\mathbf{w})]^{-1}$ donde μ es algún valor no negativo. Además el valor de la Hessiana es sustituido por $\sum_{p=1}^{P} J^p(\mathbf{w})^T J^p(\mathbf{w})$, asumiendo que $S^p(\mathbf{w}) \simeq 0$. Con todo esto llegamos al método de Levenberg–Marquardt (LM) que queda resumido en la siguiente expresión:

$$\Delta\mathbf{w} = -\alpha \left[\mu I + \sum_{p=1}^{P} J^p(\mathbf{w})^T J^p(\mathbf{w}) \right]^{-1} \nabla E(\mathbf{w}) \tag{3.23}$$

El parámetro μ se incrementa o disminuye en cada paso. Si $E(\mathbf{w}(t+1)) \leq E(\mathbf{w}(t))$ entonces en la siguiente iteración μ se divide por un factor β. En caso contrario en la siguiente iteración el parámetro μ se multiplica por β. En [HM94] se sugiere tomar $\beta = 10$

y $\mu = 0.01$ al comienzo, sin embargo, nosotros hemos usado $\mu = 0.001$ tal y como hace MATLAB.

El algoritmo Levenberg–Marquardt actúa sobre los P patrones a la vez como sigue:

1. Calcula la salida de la red y los vectores de error $\mathbf{e}^p(\mathbf{w})$ para cada uno de los patrones.

2. Calcula la matriz Jacobiana $J^p(\mathbf{w})$ para cada patrón usando entre otras cosas el vector de error $\mathbf{e}^p(\mathbf{w})$.

3. Calcula $\Delta\mathbf{w}$ usando la Ecuación 3.23 y los resultados anteriores.

4. Vuelve a calcular el error usando $\mathbf{w} + \Delta\mathbf{w}$ como pesos de la red. Si el error ha disminuido divide μ por β, hace la asignación $\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$ y vuelve al paso 1. Si el error no disminuye multiplica μ por β y vuelve al paso 3.

5. El algoritmo acaba cuando la norma del gradiente es menor que un valor predeterminado o el error se ha reducido por debajo de un objetivo.

El pseudocódigo podemos verlo en la Figura 3.2.

```
inicializa_pesos;
MIENTRAS NO condición_parada HACER
    calcula e^p(w) para cada patrón;
    e1 := 1/2 Σ_{p=1}^{P} e^p(w)^T e^p(w);
    calcula el Jacobiano para cada patrón;
    REPITE
        calcula Δw;
        e2 := 1/2 Σ_{p=1}^{P} e^p(w + Δw)^T e^p(w + Δw);
        SI (e1 <= e2) ENTONCES
            μ := μ * β;
        FINSI;
    HASTA (e2 < e1);
    μ := μ/β;
    w := w + Δw;
FINMIENTRAS;
```

Figura 3.2: Algoritmo Levenberg–Marquardt (LM).

Cuando se pasa el algoritmo LM de la teoría a la práctica hay que tratar con ciertos problemas como puede ser la singularidad de la matriz $\left[\mu I + \sum_{p=1}^{P} J^p(\mathbf{w})^T J^p(\mathbf{w})\right]$. Nuestra solución en ese caso ha sido incrementar el valor de μ multiplicándolo por β y volver a realizar los cálculos. Además de este, aparecen nuevos problemas debido a que los ordenadores sólo pueden trabajar con valores discretos. Si observamos el algoritmo Levenberg–Marquardt de la Figura 3.2 podemos ver que si nunca se reduce el error en el bucle interno, el valor de μ crece indefinidamente. Un valor muy alto de μ puede traer graves problemas en la resolución de las ecuaciones, que pueden arrojar resultados con grandes márgenes de error. Además, en una situación así el algoritmo nunca acaba. Para resolver este problema se añade un valor máximo permitido de μ. Cuando dicho valor se alcanza, se sale del bucle interno. También puede ocurrir que se reduzca el error durante muchas etapas seguidas y μ alcance el valor cero (debido a la discretización que hacen los computadores de los números reales). Si llega a ese valor no cambiará en lo que queda de ejecución. Para evitar esto último, se inicializa el valor de μ al que tenía cuando comenzó la ejecución del algoritmo cada vez que llega a cero.

3.3 Algoritmos Evolutivos

Los algoritmos evolutivos son un conjunto de metaheurísticos modernos utilizados con éxito en un número elevado de aplicaciones reales de gran complejidad. Su éxito resolviendo problemas difíciles ha sido el motor de un campo conocido como *Computación Evolutiva* (EC).

El origen de la Computación Evolutiva se sitúa a finales de los cincuenta, pero no es hasta los ochenta cuando la comunidad científica comenzó a beneficiarse del uso de estas técnicas. A raíz del desarrollo de los Algoritmos Genéticos por parte de Holland [Hol75], de la Programación Evolutiva por parte de Fogel [FOW66] y de los trabajos de Rechenberg [Rec73] acerca de las Estrategias Evolutivas, el uso de estas técnicas proliferaron en la comunidad científica.

Los beneficios de las técnicas de computación evolutiva provienen en su mayoría de las ganancias en flexibilidad y adaptación a la tarea objetivo en combinación con su comportamiento robusto y características globales de la búsqueda que llevan a cabo. En la actualidad, se entiende la Compautación Evolutiva como un concepto adaptable para

la resolución de problemas, especialmente apropiados para problemas de optimización complejos.

3.3.1 Algoritmo Evolutivo Secuencial

Un *Algoritmo Evolutivo* (EA) es un proceso iterativo y estocástico que opera sobre un conjunto de individuos (población). Cada individuo representa una solución potencial al problema que se está resolviendo. Inicialmente, la población es generada aleatoriamente (quizás con ayuda de un heurístico de construcción). A cada individuo de la población se le asigna, por medio de una función de aptitud, una medida de su bondad con respecto al problema bajo consideración. Este valor es la información cuantitativa que el algoritmo usa para guiar su búsqueda. El proceso completo es esbozado en la Figura 3.3.

```
Genera (P(0));
t := 0;
MIENTRAS NO Criterio_Terminación(P(t)) HACER
    Evalúa (P(t));
    P'(t) := Selecciona(P(t));
    P''(t) := Aplica_Operadores_Reproducción(P'(t));
    P(t+1) := Reemplaza(P(t), P''(t));
    t := t+1;
FINMIENTRAS;
DEVUELVE Mejor_Solución;
```

Figura 3.3: Algoritmo Evolutivo Secuencial.

Pueden verse en el algoritmo tres pasos principales: selección, reproducción y reemplazo. El proceso completo es repetido hasta que se cumpla un cierto criterio de terminación (normalmente después de un número dado de iteraciones). Estudiemos los tres pasos principales por separado.

- **Selección:** Partiendo de la población inicial `P(t)` de μ individuos, durante esa fase, se crea una nueva población temporal `P'(t)` de λ individuos. Generalmente los individuos más aptos (aquellos correspondientes a las mejores soluciones contenidas en la población) tienen un mayor número de instancias que aquellos que tienen

menos aptitud (selección natural). De acuerdo con los valores de μ y λ podemos definir distintos esquemas de selección [AT99] (ver Figura 3.4):

1. **Selección por Estado Estacionario**. Cuando $\lambda = 1$ tenemos una selección por estado estacionario (*steady–state*) en la que únicamente se genera un hijo en cada paso de la evolución.
2. **Selección Generacional**. Cuando $\lambda = \mu$ tenemos una selección por generaciones en la que se genera una nueva población completa de individuos en cada paso.
3. **Selección Ajustable**. Cuando $1 \leq \lambda \leq \mu$ tenemos una selección intermedia en la que se calcula un número ajustable (*generation gap*) de individuos en cada paso de la evolución. Las anteriores casos son particulares de esta.
4. **Selección por exceso**. Cuando $\lambda > \mu$ tenemos una selección por exceso típica de los procesos naturales reales.

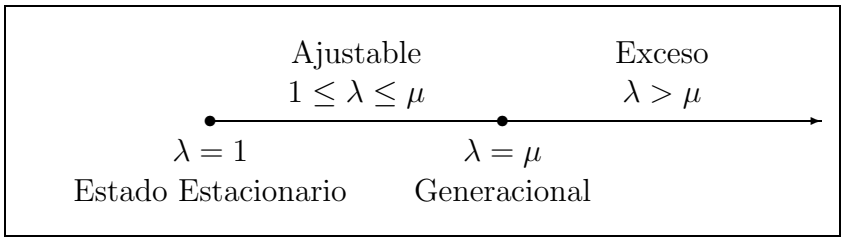

Figura 3.4: Esquemas de selección.

- **Reproducción:** En esta fase los operadores reproductivos son aplicados a los individuos de la población. Típicamente, esos operadores se corresponden con la recombinación de parejas junto con la mutación de los nuevos individuos generados. Estos operadores de variación son, en general, no deterministas, es decir, no siempre se tienen que aplicar en todas las generaciones del algoritmo, sino que su comportamiento viene determinado por su probabilidad asociada.
- **Reemplazo:** Finalmente, los individuos de la población original son sustituidos por los individuos recién creados. Este reemplazo usualmente intenta mantener

los mejores individuos eliminando los peores. Dependiendo de si para realizar el reemplazo se tiene en cuenta la antigua población `P(t)` o no podemos obtener dos tipos de estrategias de reemplazo:

1. $(\mu,\ \lambda)$ si el reemplazo se realiza utilizando únicamente los individuos de la nueva población `P'(t)`. Se debe cumnplir que $\mu \leq \lambda$.
2. $(\mu + \lambda)$ si el reemplazo se realiza seleccionando μ individuos de la unión de `P(t)` y `P'(t)`.

Estos algoritmos establecen un equilibrio entre la explotación de buenas soluciones (fase de selección) y la exploración de nuevas zonas del espacio de búsqueda (fase de reproducción), basados sobre el hecho de que la política de reemplazo permite la aceptación de nuevas soluciones que no mejoran necesariamente las existentes.

Los EAs son heurísticos, por lo que no aseguran la solución óptima. La conducta de estos algoritmos es estocástica así que lo más seguro es que encuentren diferentes soluciones en diferentes ejecuciones de un mismo algoritmo. Por eso se suelen lanzar bastantes ejecuciones del problema y se aplican estudios de significación estadística.

3.3.2 Algoritmo Evolutivo Paralelo

Para problemas no triviales, los EAs consumen muchos recursos computacionales (tanto en memoria como en CPU); por ese motivo se están estudiando una gran variedad de características de los EAs para abordar mejor el diseño de EAs eficientes. Una de estas mejoras es paralelizar estos algoritmos dando lugar a los *Algoritmos Evolutivos Paralelos* (PEAs). Los PEAs no son versiones paralelas de los algoritmos secuenciales, son algoritmos distintos que trabajan con múltiples subpoblaciones. El comportamiento de estos algoritmos paralelos es generalmente mejor que la de los EAs en muchos problemas. Estos algoritmos incluyen, tras la fase de reemplazo, una fase de comunicación entre los diferentes subalgoritmos que los componen (ver Figura 3.5).

Para terminar de describir el funcionamiento de los algoritmos paralelos necesitamos explicar la fase de comunicaciones (las demás ya fueron descritas en el apartado anterior). La comunicación viene determinada por la política de migración, que podemos describir con una tupla de cinco valores [AT00]:

```
Genera (P(0));
t := 0;
MIENTRAS NO Criterio_Terminación(P(t)) HACER
    Evalúa (P(t));
    P'(t) := Selecccciona(P(t));
    P''(t) := Aplica_Operadores_Reproducción(P'(t));
    P(t+1) := Reemplaza(P(t), P''(t));
COMUNICACIÓN;
t := t+1;
FINMIENTRAS;
DEVUELVE Mejor_Solución;
```

Figura 3.5: Algoritmo Evolutivo Paralelo.

$$M = (m, \zeta, \omega_S, \omega_R, sync) \tag{3.24}$$

donde:

- m: es el número de individuos que son migrados (*migration rate*). Alternativamente, en vez de dar el número de individuos se puede dar el porcentaje de la población a migrar.

- ζ: indica la frecuencia de la migración en número de evaluaciones.

- ω_S: define la política de selección, indicando tanto qué elementos migrar como a qué subalgoritmo.

- ω_R: define la política de reemplazo, indicando qué elementos de la población deben ser reemplazados por los individuos que han llegado.

- *sync*: este flag indica si las comunicaciones serán bloqueantes (cuando envía, espera a recibir) o no bloqueantes (los individuos de los otros subalgoritmos pueden llegar en cualquier momento).

3.3.3 Operadores

Existen muchos operadores en la literatura. En este apartado vamos a hablar de aquellos que hemos implementado. Los detalles sobre la implementación de estos operadores se encuentran en el Apéndice C. Hemos clasificado los operadores en cinco grupos que describiremos a continuación.

Operadores de Selección

Los operadores de selección implementados son: el **Torneo** y la **Ruleta**. La selección de un individuo por torneo consiste en elegir q individuos aleatoriamente y escoger el mejor de ellos. La selección por ruleta consiste en elegir al individuo de acuerdo a su aptitud. Cuanto mayor es la aptitud más probabilidades tiene de salir elegido. La probabilidad de elección de un individuo se calcula como su aptitud dividida por la suma de las aptitudes de todos los individuos de la población [Bac96].

Operadores de Recombinación

La **Recombinación de z Puntos** se aplica a dos cromosomas binarios. Divide los cromosomas por z puntos e intercambia los segmentos formados de manera que no haya en el resultado dos segmentos adyacentes procedentes del mismo individuo (Figura 3.6). Este operador devuelve dos individuos como resultado de su operación.

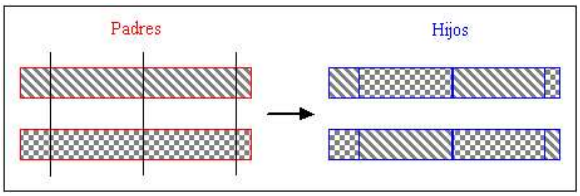

Figura 3.6: Recombinación de 3 Puntos.

La **Recombinación Uniforme** se aplica sobre dos cromosomas binarios. Para cada bit del cromosoma hijo decide de qué padre lo va a tomar. Tiene un parámetro llamado *bias* que es la probabilidad de tomar el bit del mejor padre (el de mayor aptitud). Como resultado de su operación devuelve un individuo.

La **Recombinación de 1 Punto para 2–D** se aplica sobre tablas. La recombinación de un punto para una estructura de datos bidimensional consiste en escoger aleatoriamente una fila y una columna de corte y se intercambian los elementos cuya fila y columna son menores que las escogidas y aquellos cuya fila y columna son mayores (Figura 3.7). Como resultado de esta recombinación se crean dos hijos.

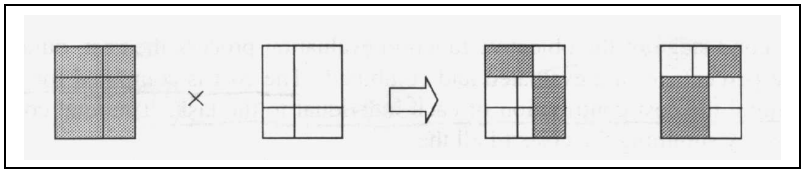

Figura 3.7: Recombinación de 1 Punto para 2–D.

El operador de **Recombinación de Aristas (ERX) para VRP** se aplica sobre soluciones del VRP. El operador funciona del siguiente modo. Dadas dos soluciones del VRP, forma una tabla de adyacencias. En dicha tabla existe una entrada por cada cliente. Cada entrada contiene los clientes adyacentes según ambas soluciones. Para generar una ruta se elige aleatoriamente el primer cliente. Se elimina el cliente elegido de las entradas de los otros clientes para que no pueda volver a visitarse. Consultamos su entrada en la tabla y escogemos como próximo cliente aquel que tenga un menor número de clientes adyacentes. Así procedemos hasta que se incumpla alguna de las restricciones de capacidad o coste o encontremos una entrada sin clientes adyacentes. En cualquiera de estos casos comenzamos una nueva ruta eligiendo de nuevo un cliente aleatoriamente de entre los restantes. El operador crea un único individuo a partir de los dos padres. Este operador se ha usado tradicionalmente para cruzar permutaciones [Whi00] y en particular, para resolver el problema del viajante de comercio (TSP). Nosotros lo hemos adaptado para trabajar con VRP, que es una generalización de TSP.

La **Recombinación Parcialmente Mapeada (PMX)** se aplica sobre dos permutaciones. Su funcionamiento es como sigue. Se eligen dos puntos de cruce y se copian los elementos entre esos puntos de uno de los padres en el hijo. El resto de los elementos se rellenan basándose en el primer padre de forma que no se repita ningún elemento [Whi00]. Este operador sólo genera un hijo (Figura 3.8).

La **Recombinación Cíclica** al igual que la anterior, actúa sobre permutaciones. Crea una partición del conjunto de posiciones de la permutación basándose en la idea de *ciclo*

Padre 1:	a	b	c	d	e	f	g	h	i	j	k	l
Padre 2:	h	k	c	e	f	d	b	l	a	i	g	j
Hijo :	i	g	c	d	e	f	b	l	a	j	k	h

Figura 3.8: Operador PMX.

[Whi00]. Luego, por cada clase de equivalencia creada elige de qué padre va a tomar los elementos. Cada clase de equivalencia es un conjunto de posiciones de la permutación. Este operador devuelve un solo hijo.

En las Estrategias Evolutivas un individuo está formado por un vector de variables y dos vectores de valores autoadaptativos. Estos vectores son desviaciones estándar y ángulos que son usados en el operador de mutación para generar el nuevo valor del vector de variables. Estos vectores autoadaptativos son recombinados y mutados como las variables. La forma en que son recombinados los diferentes vectores puede ser distinta. El operador de **Recombinación para Estrategias Evolutivas** que hemos implementado recombina el vector de variables usando la recombinación discreta uniforme y los vectores de ángulos y desviaciones usando recombinación intermedia [Bac96].

Operadores de Mutación

La **Mutación por Inversión de Bits** (Bit–Flip) se aplica a cromosomas binarios. Recorre la cadena de bits invirtiéndolos con cierta probabilidad. Esta probabilidad es un parámetro del operador.

La **Mutación por Intercambio** (Swap) se aplica a permutaciones. Su labor consiste en intercambiar los elementos de dos posiciones distintas escogidas aleatoriamente.

El operador de **Mutación para Estrategias Evolutivas** toma protagonismo frente al de recombinación en dichos EAs. A veces, ni siquiera se emplea la recombinación. El operador de mutación transforma el vector de variables y los vectores de parámetros estratégicos. Los parámetros estratégicos son desviaciones estándar y ángulos que definen la matriz de covarianza de la distribución normal multidimensional usada para generar un vector aleatorio. El vector de variables es modificado añadiéndole el vector generado. Los parámetros estratégicos por tanto, controlan la mutación del vector de variables. Estos parámetros también son mutados.

Operadores de Reemplazo

El **Reemplazo** (μ, λ) escoge μ individuos de entre los λ que han sido creados para formar la nueva población.

El **Reemplazo** $(\mu + \lambda)$ escoge μ individuos de entre los λ que han sido creados y la antigua población para formar la nueva.

Operadores de Búsqueda Dirigida

En ocasiones, cuando nos encontramos ante problemas complejos los algoritmos evolutivos usando los operadores habituales no consiguen buenos resultados. Una forma de ayudar al algoritmo es introducir un heurístico específico del problema a resolver para que se aplique sobre los individuos creados. Podemos pensar en estos heurísticos como un operador más que realizan una búsqueda dirigida.

El operador de **Entrenamiento de ANNs con BP** se aplica sobre redes neuronales. Entrena la red usando el algoritmo de retropropagación descrito en la Sección 3.1.

El operador de **Entrenamiento de ANNs con LM** se aplica sobre redes neuronales. Entrena la red usando el algoritmo Levenberg–Marquardt descrito en la Sección 3.2.

El operador de λ–**opt para VRP** se aplica sobre soluciones del VRP. Aplica a la solución el algoritmo λ–opt descrito en la Sección 3.5.

El operador de λ–**Intercambio para VRP** se aplica sobre soluciones del VRP. Aplica a la solución el algoritmo λ–Intercambio descrito en la Sección 3.6.

El operador de **Repulsión para ECC** se aplica sobre soluciones del ECC. Aplica a estas soluciones el algoritmo de Repulsión descrito en la Sección 3.8.

3.3.4 Paradigmas Importantes en la Computación Evolutiva

Los Algoritmos Evolutivos han sido divididos en cuatro categorías. Su principal diferencia es debida a los operadores que usan y en general el modo en que implementan las tres etapas mencionadas: selección, reproducción y reemplazo. Estas categorías pueden verse en la Tabla 3.1. Otras técnicas meta–heurísitcas inspiradas en la naturaleza son:

- Colonias de Hormigas (Ant Colony, ACO)
- Scatted Search (SS)

- Búsqueda tabú (Tabu Search, TS)
- Recocido Simulado (Simulated Annealing, SA)

Paradigma	**Creado por**
Algoritmos Genéticos	J.H. Holland
Estrategias Evolutivas	I. Rechenberg y H.P. Schwefel
Programación Evolutiva	L.J. Fogel, A.J. Owens y M.J. Walsh
Programación Genética	J.R. Koza

Tabla 3.1: Paradigmas en la Computación Evolutiva.

Nosotros sólo hemos empleado en el presente proyecto dos de esos paradigmas: los Algoritmos Genéticos y las Estrategias Evolutivas.

3.3.5 Relación con otras Técnicas

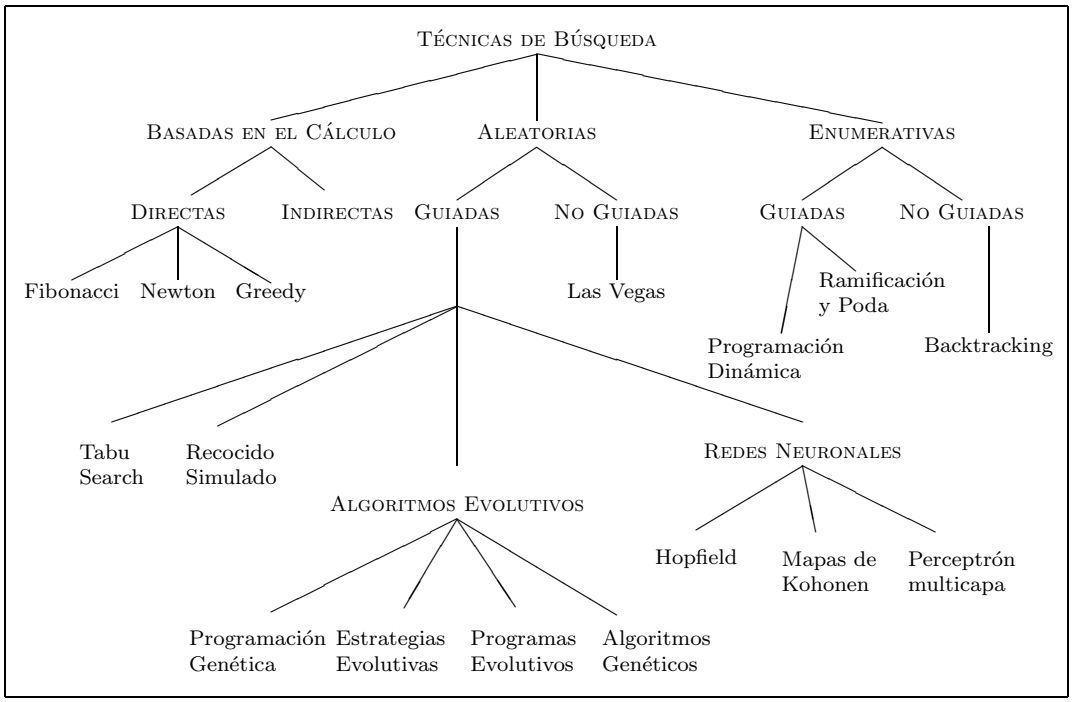

Figura 3.9: Clasificación de las Técnicas de Búsqueda.

La localización de esta clase de técnicas con respecto a otros procedimientos deterministas y no deterministas es mostrado en el siguiente árbol (ver Figura 3.9). Esta figura resume la situación de las técnicas naturales entre los otros procedimientos de búsqueda bien conocidos.

3.4 Algoritmo Savings para VRP

Como dijimos en la Sección 2.5 el VRP es un problema NP–completo y hay que recurrir a algoritmos heurísticos para encontrar una solución que sea aceptable. Existen varios algoritmos heurísticos para obtener soluciones del problema y otros tantos para mejorarlas. Estos últimos se basan en una solución ya existente para tratar de encontrar una nueva solución que sea mejor que la dada. Entre los algoritmos que obtienen soluciones se encuentra el algoritmo Savings de Clarke y Wright [CW64].

Este algoritmo se basa en unas cantidades denominadas *savings* que se calculan a partir de la matriz de costes de la siguiente forma: $s_{ij} = c_{i0} + c_{0j} - c_{ij}$. El valor de s_{ij} es la cantidad en que se reduciría el coste de la solución al unir una ruta que acaba en el nodo i y otra que comienza en el nodo j. En este algoritmo el número de vehículos es una variable más de la solución. Existen dos versiones del algoritmo: paralela y secuencial. Pero hay que aclarar que al decir "paralelo" no nos estamos refiriendo a un paralelismo de cómputo, simplemente es el nombre de la variante del algoritmo que poco tiene que ver con la concurrencia.

En la versión paralela se ordenan los *savings* por su valor de forma decreciente y se crean tantas rutas como clientes. Cada una va del almacén al cliente y vuelve. En cada paso se toma un saving s_{ij} y se comprueba si existe una ruta que acabe en i y otra que empiece en j (a lo sumo habrá una ruta de cada clase) tales que al unirlas no se exceda el coste máximo ni la capacidad del vehículo. En caso afirmativo se unen las rutas. En caso negativo, se sigue con el siguiente saving. Esto se hace hasta que no queden savings que produzcan una unión. Al tomar los savings en orden decreciente de su valor se consigue la máxima mejora posible uniendo dos rutas en cada paso.

En la versión secuencial se crean tantas rutas como clientes. Cada una va del almacén al cliente y vuelve igual que en la versión paralela. En esta ocasión tomamos una ruta $(0, i, \ldots, j, 0)$ y buscamos el saving de la forma s_{ki} o s_{jl} más grande para el que exista

una ruta acabando en i o empezando en j respectivamente, que al unirla a la actual dé lugar a una ruta factible (que cumple las restricciones de capacidad y coste máximo). Cuando no encontremos un saving así, tomamos otra ruta y hacemos lo mismo hasta que no tengamos más rutas que tomar. Vemos que en este caso la unión se hace con aquella ruta que, siendo factible, produce el mayor descenso del coste de la solución actual.

Experimentos realizados por Laporte, Gendreau, Potvin y Semet muestran que la versión paralela del algoritmo obtiene mejores resultados que la secuencial [LGPS99]. Por esto hemos implementado la versión paralela, cuyo pseudocódigo puede verse en la Figura 3.10.

3.5 Algoritmo λ–opt para VRP

De entre los algoritmos de mejora de soluciones del VRP encontramos uno que procede del problema TSP y ha sido adaptado para trabajar con VRP. Se trata del algoritmo λ–opt para VRP. El algoritmo λ–opt para VRP actúa aplicando un λ–opt del TSP en cada ruta por separado. Para realizar su cometido, λ–opt divide la ruta en λ segmentos y los reordena de todas las formas posibles para obtener nuevas soluciones que sean mejores. La división en segmentos también se hace de todas las formas posibles. La combinación de todas las posibles divisiones en segmentos con todas las posibles reordenaciones de los segmentos da lugar a una gran cantidad de nuevas soluciones. En cada paso del algoritmo, se procede a generar todas las nuevas soluciones de forma sistemática. Durante la generación, el algoritmo puede presentar dos tipos de comportamiento: cuando encuentre una solución mejor que la actual, la devuelve; o espera hasta haber explorado todas las nuevas soluciones y devuelve la mejor de ellas. El algoritmo completo consiste en la ejecución reiterada del paso anterior (Figura 3.11). Una vez concluido un paso, su salida es la entrada del paso siguiente y se procede de esta manera hasta que no se puede conseguir una mejora. Está claro que el algoritmo acaba, puesto que la solución de un TSP está acotada inferiormente (por el producto del menor coste asociado a un arco y el número de arcos, que es constante) y por tanto no puede obtener soluciones mejores indefinidamente. En la Figura 3.12 puede observarse el pseudocódigo de un paso del algoritmo cuando busca la mejor de las soluciones.

Este algoritmo es $O(n^\lambda)$, siendo n el número de nodos, y suele aplicarse con $\lambda = 2$ y

```
Calcular_savings; // s[i][j]=c[i][0]+c[0][j]-c[i][j]
Ordenar_savings;
Crear_rutas_iniciales; // de la forma (0, i, 0)

union := true;

MIENTRAS union HACER
    Inicializar_cursor_savings;
    union := false;

    MIENTRAS (NO union) Y quedan_savings HACER

        s := savings_actual;

        SI (hay_ruta_comenzando_en (s.i) Y
            hay_ruta_acabando_en (s.j) Y
            la_union_es_factible) ENTONCES

            Unir_rutas;
            union := true;

        FINSI;
    FINMIENTRAS;
FINMIENTRAS;

DEVUELVE rutas;
```

Figura 3.10: Pseudocódigo de la versión paralela del algoritmo Savings.

$\lambda = 3$.

3.6 Algoritmo λ–Intercambio

Siguiendo con el VRP le toca el turno ahora a otra técnica de mejora de soluciones que también fue desarrollada originalmente para otro problema y ha sido adaptada para el VRP. El algoritmo λ–Intercambio fue desarrollado por Osman y Christofides para el problema de agrupamiento capacitado (capacitated clustering problem). Está basado en

```
ruta := ruta_a_mejorar;
res := paso_de_λ-opt (problema, ruta);

MIENTRAS res != null Y res.coste < ruta.coste HACER

    ruta := res;
    res := paso_de_λ-opt (problema, ruta);

FINMIENTRAS;

DEVUELVE ruta;
```

Figura 3.11: Pseudocódigo del algoritmo λ–opt.

```
ruta_actual := ruta;

cdp := ruta_solucion.segmentaciones_posibles();

PARA CADA (seg DE cdp) HACER

    rutas := seg.recombinaciones_posibles();

    SI (rutas.mejor.coste < ruta_actual.coste) ENTONCES
        ruta_actual := rutas.mejor;
    FINSI;
FINPARA;

SI (ruta_actual == ruta_solucion) ENTONCES
    DEVUELVE null;
SINO
    DEVUELVE ruta_actual;
FINSI;
```

Figura 3.12: Pseudocódigo del procedimiento `paso_de_λ-opt`.

el intercambio de clientes entre las distintas rutas.

Vamos a representar las soluciones del VRP mediante una tupla

$$S = (R_1, \ldots, R_p, \ldots, R_q, \ldots, R_k)$$

donde R_i es el conjunto de clientes servidos por la ruta i-ésima y k es el número de rutas de la solución. Un λ–intercambio entre el par de rutas p y q es un reemplazo de un subconjunto $S_1 \subseteq R_p$ de tamaño $|S_1| \leq \lambda$ por otro subconjunto $S_2 \subseteq R_q$ de tamaño $|S_2| \leq \lambda$ para obtener dos nuevos conjuntos de rutas $R'_p = (R_p - S_1) \cup S_2$, $R'_q = (R_q - S_2) \cup S_1$ y una nueva solución $S' = (R_1, \ldots, R'_p, \ldots, R'_q, \ldots, R_k)$. Los clientes de los subconjuntos S_1 y S_2 se eligen de tal forma que sean consecutivos en sus respectivas rutas, en realidad son segmentos de las rutas. Además, en las nuevas rutas que se forman, el nuevo segmento ocupa el hueco que dejó el antiguo (Figura 3.13).

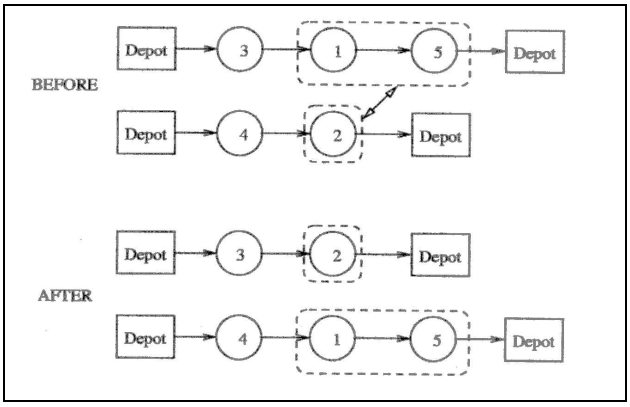

Figura 3.13: Ejemplo de λ–intercambio.

Puesto que hay k rutas podemos efectuar λ–intercambios entre $C_{k,2}$ pares posibles de rutas[2]. Llamamos *operador* de un λ–intercambio entre el par de rutas p y q al par $(|S_1|, |S_2|)$. Puesto que $|S_1|, |S_2| \leq \lambda$ hay $(\lambda + 1)^2 - 1$ operadores posibles de un λ–intercambio. Por ejemplo, si $\lambda = 2$ tenemos los operadores (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2). El operador (i, j) aplicado a un par de rutas p y q indica que se toma un segmento con i clientes de la ruta p y se intercambia con uno de j clientes de la ruta q. Para especificar completamente el λ–intercambio que realizamos tenemos que indicar cuál es el primer cliente de cada segmento. Esto se hace mediante la cuaterna (p, a, q, b), donde p y q son las rutas a las que se le aplica el intercambio y a y b son los índices de los primeros clientes de cada segmento. A esa cuaterna se le llama *operando* del λ–intercambio [TLZ99]. Dado el operador y el operando queda especificado completamente

[2]Usamos la notación $C_{n,m}$ para representar las combinaciones de n elementos tomados de m en m

el λ–intercambio. El algoritmo puede funcionar de acuerdo a dos estrategias: mejor global o primer mejor. En la estrategia mejor global se aplican todos los operadores sobre todos los operandos posibles (todos los pares de rutas y todos los índices de clientes de esas rutas) y se escoge el mejor resultado. En la estrategia primer mejor se van probando pares operando–operador secuencialmente hasta encontrar uno que obtenga una solución mejor que la de partida y se devuelve esa solución. En este segundo caso el orden en que se realiza la exploración puede influir en el resultado.

3.7 Algoritmo Greedy para TA

Para el problema de asignación de terminales existen varios algoritmos heurísticos. Algunos de ellos podemos encontrarlos en [ASW94] y [KC97]. De esta última fuente hemos tomado el algoritmo greedy que hemos implementado.

El algoritmo consiste en asignar cada terminal al concentrador factible más cercano. Por concentrador factible entendemos aquel que tiene aún capacidad para atender al terminal. Con "más cercano" queremos decir aquel cuyo coste del enlace terminal–concentrador es menor. El algoritmo procede como sigue. Se elige un terminal aleatoriamente de entre aquellos que no hayan sido asignados. Se busca el concentrador más cercano y se comprueba si tiene capacidad para atender al terminal. Si es así se asigna el terminal al concentrador y se comienza de nuevo con otro terminal elegido aleatoriamente si queda alguno. Si el concentrador no tiene capacidad para atender al terminal se toma el siguiente concentrador más cercano y se repite la comprobación. Se procede de esta forma hasta que se encuentre un concentrador factible o no haya más concentradores. En el último caso el algoritmo no encuentra una solución factible.

Nuestra versión del algoritmo difiere ligeramente con la anterior. En nuestro caso, el algoritmo puede funcionar siguiendo dos estrategias: admitir soluciones inválidas o no. Estas estrategias determinan qué hacer cuando el algoritmo no encuentra una solución factible. En el primer caso, cuando un terminal no puede ser asignado a ningún concentrador se asigna a uno cualquiera de forma aleatoria. En el segundo, cuando ocurre la misma situación se comienza de nuevo la ejecución del algoritmo y se seguirá iniciando mientras no se encuentre una solución factible. Esta última estrategia hace que el algoritmo sea más lento pero todas las soluciones que devuelve son válidas. La estrategia

que admite soluciones no válidas puede tener interés para inicializar la población de un Algoritmo Genético ([KC97]) donde el proceso de búsqueda puede conseguir soluciones factibles a partir de las no factibles.

3.8 Algoritmo de Repulsión para ECC

Este algoritmo ha sido desarrollado durante el transcurso de este proyecto para mejorar los resultados obtenidos con el operador de mutación por inversión de bits.

Si colocamos una serie de partículas cargadas con la misma carga en la superficie de una esfera éstas se repelerán y tratarán de alejarse unas de otras. Las fuerzas que provocan este alejamiento pueden calcularse mediante la Ley de Coulomb[3]. Las palabras de un código binario pueden ser vistas como partículas en un espacio n–dimensional donde n es la longitud de las palabras. Una solución para el ECC será tanto mejor cuanto mayor sea la distancia entre las palabras. Por tanto, usando una analogía con la física podemos considerar las palabras como partículas y aplicar la Ley de Coulomb para calcular las fuerzas entre ellas y simular su movimiento.

Existen algunas diferencias importantes entre ambas situaciones. Para empezar, las palabras del código se encuentran en los vértices de un hipercubo de n dimensiones (o n–cubo) mientras que la analogía la hemos hecho con partículas confinadas en la superficie de una esfera de tres dimensiones. Las dimensiones no son un problema puesto que podemos generalizar la Ley de Coulomb trabajando con distancias eculídeas en n dimensiones. Aún así, las partículas se pueden mover con total libertad sobre la superficie de una hiperesfera (una n–esfera) mientras que las palabras sólo pueden moverse por las aristas de un n–cubo para acabar en otro vértice inmediatamente (ni siquiera se pueden quedar en mitad de la arista).

El algoritmo calcula las fuerzas entre cada par de palabras[4] del código (considerando que tienen la misma carga) para después calcular las fuerzas resultantes que se aplica a cada una de ellas. Llamemos $\mathbf{f}_{ij}$ a la fuerza que la palabra j ejerce sobre la i y $\mathbf{F}_i$ a la

[3]La Ley de Coulomb establece que la fuerza que ejerce una partícula cargada sobre otra es $F = \frac{1}{4\pi\varepsilon}\frac{q_1 \cdot q_2}{d^2}$ donde ε es la permitividad eléctrica del medio, q_1 y q_2 son las cargas de las partículas y d es la distancia que las separa

[4]A partir de ahora las palabras son vistas como partículas además de cadenas binarias de ahí que puedan calcularse fuerzas entre ellas

fuerza resultante que se aplica sobre la palabra i-ésima. Las expresiones de estos vectores de fuerzas son:

$$\mathbf{f}_{ij} = \frac{1}{d_{ij}} \frac{\mathbf{p}_i - \mathbf{p}_j}{\sqrt{d_{ij}}} \tag{3.25}$$

$$\mathbf{F}_i = \sum_{j=1, j\neq i}^{M} \mathbf{f}_{ij} \tag{3.26}$$

$\mathbf{p}_i$ y $\mathbf{p}_j$ son las palabras i y j, M es el número de palabras del código y d_{ij} es la distancia de Hamming[5] entre las palabras i y j.

Una vez que conocemos la fuerza resultante sobre cada palabra tenemos que simular el movimiento. En el algoritmo cada palabra sólo se puede mover a un vértice adyacente. Eso significa invertir un bit de la palabra. Para averiguar sobre qué arista se moverá cada palabra descomponemos el vector de fuerza resultante como suma de dos vectores perpendiculares: uno normal al n–plano tangente de la n–esfera en que se haya circunscrito el n–cubo[6] al que llamaremos $\mathbf{F}_i^n$ (componente normal) y otro contenido en él al que llamaremos $\mathbf{F}_i^t$ (componente tangencial). Las expresiones de estos vectores son:

$$\mathbf{F}_i^n = (\mathbf{F}_i \cdot \hat{\mathbf{n}}_i)\hat{\mathbf{n}}_i \tag{3.27}$$

$$\mathbf{F}_i^t = \mathbf{F}_i - \mathbf{F}_i^n \tag{3.28}$$

donde $\hat{\mathbf{n}}_i$ es el versor normal a la n–esfera en $\mathbf{p}_i$ cuya expresión es:

$$\hat{\mathbf{n}}_i = \frac{\mathbf{p}_i - \frac{1}{2}\mathbf{o}_n}{|\mathbf{p}_i - \frac{1}{2}\mathbf{o}_n|}$$

donde $\mathbf{o}$ representa al vector con n componentes todas a 1.

La componente normal a la n–esfera la descartamos y nos quedamos con la tangencial ($\mathbf{F}_i^t$). Si las palabras se pudieran mover libremente por la superficie de la n–esfera este vector nos indicaría la aceleración del movimiento. Como sólo podemos movernos por una

[5]Para palabras binarias la distancia de Hamming coincide con el cuadrado de la distancia euclídea. Por tanto, si queremos respetar la Ley de Coulomb debemos usar la distancia de Hamming y no su cuadrado, en contra de lo que pudiera parecer.

[6]Sólo existe una n–esfera que circunscribe a un n–cubo.

arista, ahora determinamos cuál es la arista que forma menor ángulo con la componente tangencial de la fuerza[7] y esa será la arista por la que se mueva la palabra.

Una vez que hemos hecho esto para todas las palabras deberíamos moverlas todas ellas en la dirección adecuada (que será distinta en cada palabra). Sin embargo, pruebas preliminares demostraron que el algoritmo da mejores resultados si sólo se mueve una partícula elegida aleatoriamente. Esto tiene además la ventaja de que no es necesario hacer los cálculos para todas las palabras. El algoritmo adopta esta segunda estrategia, es decir, mueve sólo una palabra.

A veces puede ocurrir que la componente tangencial sea muy pequeña debido a que la palabra ya está bien situada. Para evitar que se muevan las palabras en ese caso se puede especificar el valor mínimo que ha de tener la componente tangencial de la fuerza para que actúe moviendo la palabra.

[7] Obsérvese que el parecido con el fenómeno físico acaba aquí. No sólo no nos movemos en una n–esfera sino que no tenemos en cuenta la velocidad de las partículas. Tratamos la componente tangencial de la fuerza como si fuera proporcional a la velocidad y no a la aceleración.

Capítulo 4

Detalles sobre la resolución de los problemas

En este capítulo detallaremos todo lo relacionado con las instancias de los problemas, la forma de evaluar las soluciones y los algoritmos usados para resolverlas con la intención de que los resultados sean reproducibles. Para cada problema describiremos los formatos de los ficheros con las instancias y detallaremos cualquier preprocesamiento que hayan recibido, indicaremos la forma de evaluar la calidad de las soluciones que los algoritmos aportan y detallaremos cúales han sido usados en cada problema junto con todos los parámetros involucrados así como la función de aptitud utilizada en el caso de los EAs.

4.1 Problema ANN

4.1.1 Instancias

Los ficheros con los patrones para el entrenamiento supervisado de las ANNs han sido obtenidos del conjunto de problemas PROBEN1 [Pre94]. Estos ficheros han sido generados automáticamente a partir de los datos originales que pueden obtenerse en el repositorio UCI y que se encuentran también entre los archivos de PROBEN1. Para la generación se ha usado el script de perl que para cada problema se adjunta. Este script transforma los patrones de acuerdo a una serie de normas que Prechelt menciona en [Pre94] y que se repiten más abajo.

Cómo representar los atributos de entrada y salida de un problema de aprendizaje

de redes neuronales es una de las decisiones claves que influyen en los resultados. Cada problema tendrá una serie de atributos de distintos tipos. Para cada tipo de atributo hay varias formas de representarlo en la red neuronal. Las representaciones que se han usado para los atributos de entrada en las instancias resueltas son las siguientes:

- Los **atributos reales** son transformados por una función lineal para que se encuentren en el rango $[0, 1]$ o $[-1, 1]$. Cada atributo es representado mediante una entrada en la red.

- Los **atributos enteros** son tratados como atributos reales.

- Los **atributos nominales** con m valores diferentes son representados usando un código 1–de–m o un código binario. En el primer caso se usan m entradas de la red para representar el atributo de las cuales una tiene valor 1 y el resto 0. En el segundo se asocia un número binario a cada valor del atributo y se emplean $\lceil \log_2 m \rceil$ entradas para representarlo. Para cada instancia se dirá qué representación se ha usado para los atributos nominales.

- Los **atributos de clases** con m valores diferentes son representados como los atributos nominales usando un código 1–de–m o un código binario. Los valores de estos atributos son números enteros que representan una de varias clases posibles (la magnitud del número no tiene interés).

- Los **valores ausentes** pueden ser reemplazados por un valor fijo (como la media de los valores presentes de ese atributo) o pueden ser representados explícitamente añadiendo otra entrada para el atributo. Esta entrada será 1 cuando el valor no está y 0 cuando está.

Los atributos de salida siguen las mismas reglas. Las instancias que hemos resuelto son todas de problemas de clasificación y la salida es una clase. La representación de la salida se hace siempre mediante un código 1–de–m.

A continuación mencionaremos para cada instancia el formato de los datos originales, el número de clases de salida, el preprocesamiento realizado, el tamaño de los vectores de entrada y salida de cada patrón una vez preprocesado y la red utilizada para aprender los patrones.

Instancia Cancer

Los patrones originales están formados por diez atributos y el diagnóstico, todos valores enteros positivos. El primero de estos atributos es un identificador que hay que descartar. Los otros nueve se encuentran en el rango $1 - 10$ y el diagnóstico puede tomar valor 2 o 4 dependiendo de si el tumor es benigno o maligno respectivamente. Esto determina dos clases de salida. Cada patrón se encuentra en una línea distinta y los valores están separados por comas. Existen 16 patrones con el séptimo atributo ausente. Hay un total de 699 patrones.

El preprocesamiento que se hace es el siguiente. El primer atributo es descartado puesto que no tiene ningún interés (es un identificador de patrón). El resto de los nueve atributos son divididos por 10 para que se encuentren dentro del intervalo $[0, 1]$. Con eso ya está formado el vector de entrada. Los valores desconocidos del séptimo atributo son sustituidos por la media del resto de los valores que es 0.35. En el fichero resultante cada patrón se encuentra en una línea. Los valores están separados por espacios y no existe separación especial entre el vector de entrada y el de salida. Los patrones están formados por 9 entradas y 2 salidas.

La red neuronal usada para que aprenda los patrones ha sido un perceptrón multicapa con tres capas. La capa de entrada tiene 9 neuronas, la de salida 2 y la oculta 6. La función de activación usada ha sido la logística.

Instancia Diabetes

Los patrones originales están formados por ocho atributos y el diagnóstico, todos numéricos. Cada atributo se mueve en un rango distinto y el diagnóstico puede tomar valor 0 o 1 dependiendo de si la persona tiene diabetes o no (dos clases). Cada patrón se encuentra en una línea distinta y los valores están separados por comas. No hay valores ausentes. Hay un total de 768 patrones.

El preprocesamiento que se hace es el siguiente. A cada uno de los ocho atributos se les aplica una transformación lineal (distinta para cada uno) para que se encuentren dentro del intervalo $[0, 1]$. Con eso ya está formado el vector de entrada. En el fichero resultante cada patrón se encuentra en una línea. Los valores están separados por espacios y no existe separación especial entre el vector de entrada y el de salida. Los patrones están

formados por 8 entradas y 2 salidas.

La red neuronal usada para que aprenda los patrones ha sido un perceptrón multicapa con tres capas. La capa de entrada tiene 8 neuronas, la de salida 2 y la oculta 6. La función de activación usada ha sido la logística.

Instancia Heart

Los patrones originales están formados por trece atributos y el diagnóstico. Todos vienen expresados mediante números, pero algunos son clases y otros son valores reales. Salvo para los dos primeros atributos, para todos los demás hay valores ausentes. El diagnóstico puede tomar valor 0 o 1 (dos clases). Cada patrón se encuentra en una línea distinta y los valores están separados por comas. Hay un total de 920 patrones.

El preprocesamiento que se hace es el siguiente. Lo valores ausentes son representados añadiendo una entrada más. Para las clases se combinan las dos representaciones posibles. En el segundo, sexto y noveno atributos se usa representación binaria (con 0 y 1 para representar el valor de cada bit); en el resto de los atributos de clases se usa 1–de–m. Para los atributos numéricos se usa una transformación lineal que los introduce en el rango $[0-1]$. En el fichero resultante cada patrón se encuentra en una línea. Los valores están separados por espacios y no existe separación especial entre el vector de entrada y el de salida. Los patrones están formados por 35 entradas y 2 salidas[1].

La red neuronal usada para que aprenda los patrones ha sido un perceptrón multicapa con tres capas. La capa de entrada tiene 35 neuronas, la de salida 2 y la oculta 6. La función de activación usada ha sido la logística.

Instancia Gene

Los patrones originales están formados por el tipo de unión, un nombre y la secuencia de nucleótidos, todos atributos nominales. El nombre no tiene interés y debe ser descartado. El tipo de unión es la salida y puede tomar tres valores (tres clases). La secuencia de nucleótidos está formada por una cadena de 60 caracteres que pueden ser G, A, G, o T. El tipo de unión, el nombre y la secuencia están separados por comas. Cada patrón se

[1]En realidad son 33 entradas útilies porque hay dos que siempre están a 0. Esto posiblemente sea fruto de un error en el script de perl que genera los datos, pues parece que el autor quería usar representación 1–de–m en los atributos sexto y noveno.

encuentra en una línea. De los 3190 patrones que trae el fichero, 15 presentan secuencias incompletas y deben ser descartados. Por lo tanto hay un total de 3175 patrones útiles.

El preprocesamiento que se hace es el siguiente. Cada nucleótido de la secuencia se considera un atributo de entrada que es representado mediante codificación binaria con valores -1 y 1 para cada bit. El atributo de salida es representado mediante codificación 1–de–m. En el fichero resultante cada patrón se encuentra en una línea. Los valores están separados por espacios y no existe separación especial entre el vector de entrada y el de salida. Los patrones están formados por 120 entradas y 3 salidas.

La red neuronal usada para que aprenda los patrones ha sido un perceptrón multicapa con tres capas. La capa de entrada tiene 120 neuronas, la de salida 3 y la oculta 6. La función de activación usada ha sido la logística.

Instancia Soybean

Los patrones originales están formados por 35 atributos y la clase de dolencia, que es un atributo nominal con 19 valores posibles (19 clases). La mayoría de los atributos son de clases. En el fichero todos son números excepto el nombre de la dolencia, que aparece al principio. Los valores ausentes son muchos. Hay un total de 683 patrones.

El preprocesamiento de este conjunto de patrones aplica varios tipos de representaciones para las clases y valores ausentes. Algo que supone una novedad es aplicar al valor de un atributo de clase una transformación lineal que lo confina en el intervalo $[0, 1]$ o $[-1, 1]$. Para el resto de los atributos de clases se emplea codificación 1–de–m o binaria (con valores 0 y 1). En algunos atributos, a los valores ausentes se les asigna un valor fijo. En otros se añade una entrada más. Y para el resto, se hace de nuevo algo novedoso: son tratados como un valor más y se emplea en el atributo codificación binaria (con valores 0 y 1). En el fichero resultante cada patrón se encuentra en una línea. Los valores están separados por espacios y no existe separación especial entre el vector de entrada y el de salida. Los patrones están formados por 82 entradas y 19 salidas.

La red neuronal usada para que aprenda los patrones ha sido un perceptrón multicapa con tres capas. La capa de entrada tiene 82 neuronas, la de salida 19 y la oculta 6. La función de activación usada ha sido la logística.

Instancia Thyroid

Los patrones originales están ya preprocesados y a excepción de la salida, que es un atributo de clase que toma valores entre 1 y 3 (tres clases), no hay que preprocesarlos. Hay un total de 7200 patrones. En el fichero resultante cada patrón se encuentra en una línea. Los valores están separados por espacios y no existe separación especial entre el vector de entrada y el de salida. Los patrones están formados por 21 entradas y 3 salidas.

La red neuronal usada para que aprenda los patrones ha sido un perceptrón multicapa con tres capas. La capa de entrada tiene 21 neuronas, la de salida 3 y la oculta 6. La función de activación usada ha sido la logística.

4.1.2 Evaluación

Para evaluar las redes se ha usado el Porcentaje del Error Cuadrático (SEP) y el Porcentaje del Error de Clasificación (CEP). El SEP se define como:

$$SEP = 100 \cdot \frac{o_{max} - o_{min}}{P \cdot S} \sum_{p=1}^{P} \sum_{i=1}^{S} (t_i^p - o_i^p)^2$$

donde o_{min} y o_{max} son los valores mínimo y máximo de las neuronas de salida (asumiendo que son los mismos para todas ellas), S es el número de neuronas de salida y P el número de patrones del conjunto considerado. En los problemas que hemos resuelto $o_{max} = 1$ y $o_{min} = 0$. El CEP es el porcentaje de patrones mal clasificados por la red. La salida de la red es interpretada siguiendo la técnica de "el ganador se lleva todo" (véase Sección 2.1).

A la hora de realizar las pruebas nos encontramos con diferentes formas de trabajar con las redes y los algoritmos. Podemos usar un conjunto de patrones para entrenar y otro para testear la red resultante, o bien, podemos usar algunos patrones para entrenar, otros para decidir cuándo detener el entrenamiento y otros para testear la red. Pero también podemos emplear Cross–Validation. En este último caso se divide el conjunto de patrones en varios grupos y se realizan tantas pruebas como grupos. En cada prueba se emplea uno de esos grupos para testear las red y el resto para entrenarla y al final se calcula la media de los resultados obtenidos para cada experimento. Se suelen hacer los grupos de forma que haya el mismo número de patrones en cada uno[2]. Existen por tanto

[2]Esto no siempre es posible porque el conjunto de patrones puede no ser un múltiplo del número de grupos. En ese caso habrá un grupo con más patrones que el resto.

muchas formas de abordar las pruebas. Nosotros hemos optado por aplicar dos de ellas: dividir el conjunto de patrones en un grupo de entrenamiento y otro de test (Training & Test) y usar Cross–Validation dividiendo el conjunto de patrones en cinco grupos (5–fold Cross–Validation).

A continuación indicamos para cada instancia la forma en que hemos dividido el conjunto total de patrones para Training & Test y 5–fold Cross–Validation.

Instancia Cancer

En el Training & Test se han tomado los primeros 525 patrones para entrenar la red y los 174 restantes para testearla. En el 5–fold Cross–Validation los patrones exactos que hay en cada grupo son los siguientes:

- Primer grupo: patrones 1 a 139 (139 patrones)
- Segundo grupo: patrones 140 a 278 (139 patrones)
- Tercer grupo: patrones 279 a 417 (139 patrones)
- Cuarto grupo: patrones 418 a 556 (139 patrones)
- Quinto grupo: patrones 557 a 699 (143 patrones)

Instancia Diabetes

En el Training & Test se han tomado los primeros 576 patrones para entrenar la red y los 192 restantes para testearla. En el 5–fold Cross–Validation los patrones exactos que hay en cada grupo son los siguientes:

- Primer grupo: patrones 1 a 153 (153 patrones)
- Segundo grupo: patrones 154 a 306 (153 patrones)
- Tercer grupo: patrones 307 a 459 (153 patrones)
- Cuarto grupo: patrones 460 a 612 (153 patrones)
- Quinto grupo: patrones 613 a 768 (156 patrones)

Instancia Heart

En el Training & Test se han tomado los primeros 690 patrones para entrenar la red y los 230 restantes para testearla. En el 5–fold Cross–Validation los patrones exactos que hay en cada grupo son los siguientes:

- Primer grupo: patrones 1 a 184 (184 patrones)
- Segundo grupo: patrones 185 a 368 (184 patrones)
- Tercer grupo: patrones 369 a 552 (184 patrones)
- Cuarto grupo: patrones 553 a 736 (184 patrones)
- Quinto grupo: patrones 737 a 920 (184 patrones)

Instancia Gene

En el Training & Test se han tomado los primeros 2382 patrones para entrenar la red y los 793 restantes para testearla. En el 5–fold Cross–Validation los patrones exactos que hay en cada grupo son los siguientes:

- Primer grupo: patrones 1 a 635 (635 patrones)
- Segundo grupo: patrones 636 a 1270 (635 patrones)
- Tercer grupo: patrones 1271 a 1905 (635 patrones)
- Cuarto grupo: patrones 1906 a 2540 (635 patrones)
- Quinto grupo: patrones 2541 a 3175 (635 patrones)

Instancia Soybean

En el Training & Test se han tomado los primeros 513 patrones para entrenar la red y los 170 restantes para testearla. En el 5–fold Cross–Validation los patrones exactos que hay en cada grupo son los siguientes:

- Primer grupo: patrones 1 a 136 (136 patrones)

- Segundo grupo: patrones 137 a 272 (136 patrones)
- Tercer grupo: patrones 273 a 408 (136 patrones)
- Cuarto grupo: patrones 409 a 544 (136 patrones)
- Quinto grupo: patrones 545 a 683 (139 patrones)

Instancia Thyroid

En el Training & Test se han tomado los primeros 5400 patrones para entrenar la red y los 1800 restantes para testearla. En el 5–fold Cross–Validation los patrones exactos que hay en cada grupo son los siguientes:

- Primer grupo: patrones 1 a 1440 (1440 patrones)
- Segundo grupo: patrones 1401 a 2880 (1440 patrones)
- Tercer grupo: patrones 2881 a 4320 (1440 patrones)
- Cuarto grupo: patrones 7321 a 5760 (1440 patrones)
- Quinto grupo: patrones 5761 a 7200 (1440 patrones)

4.1.3 Algoritmos

Los algoritmos usados para entrenar las redes han sido cinco. Estos son Retropropagación (BP), Levenberg–Marquardt (LM), Estrategias Evolutivas (ES) y dos híbridos: Algoritmos Genéticos + Retropropagación (GA+BP) y Algoritmos Genéticos + Levenberg–Marquardt (GA+LM). Antes de pasar a detallar los parámetros de estos algoritmos vamos a describir la particular forma en que se llevó a cabo la inicialización de los pesos y umbrales de las redes neuronales, ya que es un paso común a todos los algoritmos de entrenamiento.

En las primeras pruebas nos dimos cuenta de que la elección del intervalo de números reales en el que iban a estar los pesos (y umbrales) influía en el desarrollo del entrenamiento. Un intervalo demasiado amplio provocaba que apenas se modificaran los pesos

y la red no aprendía los patrones. Esto era debido a que el potencial sináptico de las neuronas era muy alto en valor absoluto y la derivada de la función logística era prácticamente nula en ese punto; con lo que la norma del gradiente era cercana a cero y el incremento en los pesos muy pequeño[3]. La solución que adoptamos para este problema fue inicializar los pesos de tal forma que el potencial sináptico máximo que pueda alcanzar una neurona se encuentre dentro de un intervalo en el que la derivada de la función de transferencia sea mayor que un cierto valor. Esto se consigue calculando para cada neurona un intervalo dentro del cual los pesos pueden tomar cualquier valor sin que el potencial sináptico alcance puntos donde la derivada de la función de transferencia esté por debajo del valor preestablecido.

La función de transferencia que hemos usado ha sido la logística, cuya derivada es par (simétrica con respecto al eje de ordenadas). Esta propiedad implica que el intervalo dentro del cual debe moverse el potencial sináptico está centrado en cero. Por lo tanto, si llamamos $\mathbf{w}$ al vector de pesos que ponderan la entrada $\mathbf{x}$ de una neurona determinada la desigualdad a la que deseamos llegar tras la inicializacón podemos escribirla del siguiente modo:

$$|\mathbf{w} \cdot \mathbf{x}| \leq h_{max} \tag{4.1}$$

donde h_{max} es el mayor valor del potencial sináptico permitido para asegurar que la derivada se encuentre por encima del valor deseado y la expresión $\mathbf{w} \cdot \mathbf{x}$ es el potencial sináptico de la neurona. Usando la definición de producto escalar de dos vectores y las propiedades del coseno podemos escribir:

$$|\mathbf{w} \cdot \mathbf{x}| \leq |\mathbf{w}| \cdot |\mathbf{x}| \cdot |\cos(\mathbf{w}, \mathbf{x})| \leq |\mathbf{w}| \cdot |\mathbf{x}| \tag{4.2}$$

En virtud de la expresión anterior, el objetivo inicial estará cumplido si conseguimos que se cumpla

$$|\mathbf{w}| \cdot |\mathbf{x}| \leq h_{max}$$

o equivalentemente

$$|\mathbf{w}| \leq \frac{h_{max}}{|\mathbf{x}|} \tag{4.3}$$

[3]Debido a la pérdida de precisión de las operaciones y a la representación discreta de los valores reales el incremento de los pesos podía ser nulo en algunos casos.

El valor de h_{max} es una constante conocida pero el valor de $|\mathbf{x}|$ depende por lo general de la entrada aplicada a la red y de los pesos de la misma. No obstante, podemos calcular una cota superior de $|\mathbf{x}|$ y usar dicha cota en su lugar sin que se incumpla la Desigualdad 4.3. Para calcular la cota necesitamos conocer una cota superior del cuadrado de cada componente de $\mathbf{x}$. Si llamamos c_i a una cota superior de x_i^2 entonces $\sqrt{\sum_{i=1}^{n} c_i}$ es una cota superior de $|\mathbf{x}|$. Es decir,

$$|\mathbf{x}| = \sqrt{\sum_{i=1}^{n} x_i^2} \leq \sqrt{\sum_{i=1}^{n} c_i} \tag{4.4}$$

donde n es el número de entradas de la neurona. Para calcular una cota superior del cuadrado de una entrada de la neurona hay que tener en cuenta si procede de una neurona de entrada o no. En el primer caso se explora el conjunto de patrones con los que se va a entrenar la red para determinar el valor máximo que alcanza el cuadrado de esa entrada. En el segundo caso el valor máximo del cuadrado de la entrada es el máximo global del cuadrado de la función de transferencia de la neurona a la que está conectada. Con todo esto llegamos a la expresión

$$|\mathbf{w}| \leq \frac{h_{max}}{\sqrt{\sum_{i=1}^{n} c_i}} \tag{4.5}$$

Para la inicialización de los pesos vamos a generar números aleatorios dentro del intervalo $[-w_{max}, w_{max}]$, donde w_{max} es un número positivo que aún no hemos determinado. El módulo de $\mathbf{w}$ se hace máximo cuando todos los pesos toman el valor $-w_{max}$ o w_{max}. En ese caso $|\mathbf{w}| = w_{max} \cdot \sqrt{n}$ y sustituyendo en la Desigualdad 4.5 podemos obtener una expresión para calcular w_{max} que es:

$$w_{max} = \frac{h_{max}}{\sqrt{n \cdot \sum_{i=1}^{n} c_i}} \tag{4.6}$$

De esta forma podemos asegurar tras la inicialización que el potencial sináptico de la neurona no supera h_{max} y la derivada de la función de transferencia de la neurona evaluada en el potencial sináptico está por encima del valor deseado. El valor de w_{max} se calcula para cada neurona y sólo los pesos que ponderen las entradas de dicha neurona se inicializan con valores dentro del intervalo $[-w_{max}, w_{max}]$. No hemos hablado explícitamente de los umbrales porque son tratados como un peso más asociados a una entrada que siempre toma valor -1.

El valor mínimo de la derivada que hemos considerado para la inicialización de los pesos es 0.01, lo cual implica que el potencial sináptico debe tomar como valor máximo $h_{max} \approx 4.6$

A continuación detallaremos los parámetros de cada algoritmo para cada instancia.

Retropropagación

El algoritmo **BP** se ha usado entrenando la red por lotes (se intenta minimizar el error cuadrático sumado de todos los patrones del conjunto de entrenamiento). Las épocas de entrenamiento, la constante de momentos (α) y la tasa de aprendizaje (η) usada para cada instancia se puede ver en la Tabla 4.1. Las abreviaturas usadas para los nombres de las instancias son las siguientes:

- **BC**: Instancia Cancer.
- **DI**: Instancia Diabetes.
- **HE**: Instancia Heart.
- **GE**: Instancia Gene.
- **SO**: Instancia Soybean.
- **TH**: Instancia Thyroid.

	BC	**DI**	**HE**	**GE**	**SO**	**TH**
Épocas	1000	1000	500	19	18	63
η	0.01	0.01	0.001	0.0001	0.001	0.0001
α	0.0	0.0	0.0	0.0	0.0	0.0

Tabla 4.1: Parámetros del algoritmo BP.

Para decidir el valor de los parámetros η y α se hicieron unas pocas ejecuciones con pocas épocas probando distintos valores para tomar los que mejores resultados dieron. Curiosamente los mejores resultados se obtuvieron haciendo $\alpha = 0.0$. Eso significa que el algoritmo entrena usando la regla delta; tan sólo usa la información del gradiente para

cambiar los pesos. Esto puede ser debido a que el valor de η es muy bajo, lo cual hace que el movimiento de los pesos sea lento y no se produzcan las oscilaciones que corregía la adición del término de momentos. El número de épocas de HE, GE, SO y TH han tenido que ser menos de 1000 debido a su lentitud en el entrenamiento, ya que las redes de estos problemas tienen muchas neuronas y por tanto muchos pesos y umbrales para entrenar.

Levenberg–Marquardt

El algoritmo **LM** se ha usado entrenando la red por lotes. A diferencia del algoritmo presentado en la Sección 3.2 la condición de parada es alcanzar un número determinado de épocas. Los parámetros usados en cada instancia se encuentran en la Tabla 4.2.

	BC	**DI**	**HE**	**GE**	**SO**	**TH**
Épocas	1000	1000	500	19	18	63
α	1.0	1.0	1.0	1.0	1.0	0.1
μ	0.001	0.001	0.001	0.001	0.001	0.001
β	10	10	10	10	10	10
μ_{max}	10^{10}	10^{10}	10^{10}	10^{10}	10^{10}	10^{10}

Tabla 4.2: Parámetros del algoritmo LM.

El valor de μ, β y μ_{max} lo tomamos de la implementación que hace MATLAB del algoritmo. Para decidir el valor de α se hicieron unas pocas ejecuciones con pocas épocas probando distintos valores para tomar el que mejores resultados dio. Obsérvese que tan sólo **TH** usa un valor distinto de α.

Estrategias Evolutivas

En la **ES** el entrenamiento de la red se hace por lotes. El vector de variables se forma concatenando los pesos con los umbrales de la red en ese orden. Hemos puesto juntos los pesos de los arcos que entran a la misma neurona. La ES implementada trata de maximizar la función de aptitud; por esto la función de aptitud que guía la búsqueda es la inversa del SEP para el conjunto de patrones de entrenamiento, esto es:

$$f(\mathbf{x}) = \frac{1}{100 \cdot \frac{o_{max}-o_{min}}{P \cdot S} \sum_{p=1}^{P} \sum_{i=1}^{S} (t_i^p - o_i^p(\mathbf{x}))^2} \tag{4.7}$$

donde $\mathbf{x}$ es el vector que contiene los pesos y umbrales de la red, de los cuales dependen las salidas ($o_i^p(\mathbf{x})$). La población consta de un único individuo y no hay recombinación. A partir de ese individuo se crean diez por mutación. La mutación no es correlada y por tanto los individuos no tienen vector de ángulos, sólo de desviaciones estándar. Tras evaluar los nuevos individuos se forma la nueva población reemplazando de forma elitista (reemplazo $(1 + 10)$). La condición de parada es alcanzar 1000 épocas. Este algoritmo sólo se ha aplicado a la instancia **BC**. Los parámetros del algoritmo se resumen en la Tabla 4.3.

	BC
Tamaño de población	1
Hijos	10
Recombinación	Ninguna
Mutación	$p_c = 1.0$
Reemplazo	$(\mu + \lambda)$
Criterio de parada	Alcanzar 1000 épocas

Tabla 4.3: Parámetros de ES.

Algoritmo Genético+Retropropagación

En **GA+BP** necesitamos una forma de representar los pesos de la red como cadenas binarias. Para realizar la codificación, se emplea un número fijo de bits por cada valor real. Al valor entero que representan esos bits se les aplica una transformación lineal para que el valor resultante se encuentre dentro de un intervalo. Esta transformación lineal queda totalmente definida dando los límites de ese intervalo, es decir, dando los valores mínimo y máximo que pueden tomar los umbrales y los pesos. Hemos usado 16 bits para representar los pesos y umbrales los cuales se encuentran confinados en el intervalo $[-1, 1]$. Un detalle clave en la codificación es el orden en que se colocan los pesos y los umbrales en la cadena. En nuestro caso hemos colocado primero los pesos y luego los umbrales. Además hemos puesto juntos los pesos de los arcos que entran a la misma neurona. El Algoritmo Genético que hemos implementado trata de maximizar la función de aptitud; por esto la función de aptitud que guía la búsqueda del GA es la inversa del SEP para el conjunto de patrones de entrenamiento (Ecuación 4.7).

El algoritmo GA+BP es similar a un Algoritmo Genético pero sustituye la mutación por una época de entrenamiento con BP. Este entrenamiento se hace con probabilidad p_t sobre uno de los individuos que genera la recombinación. La Tabla 4.4 muestra los parámetros del algoritmo para cada instancia.

	BC	**DI**	**HE**	**GE**	**SO**	**TH**
Tamaño de población	64					
Selección	Ruleta (2 individuos)					
Recombinación	Recombinación de 1 punto ($p_c = 1.0$)					
Reemplazo	$(\mu + \lambda)$					
Criterio de parada	Alcanzar 1000 épocas					
p_t	1.0	1.0	0.5	0.019	0.018	0.063
η	0.01	0.01	0.001	0.0001	0.001	0.0001
α	0.0	0.0	0.0	0.0	0.0	0.0

Tabla 4.4: Parámetros del algoritmo GA+BP.

La probabilidad p_t se ha elegido para que el número medio de épocas de entrenamiento con BP en GA+BP coincida con el número de épocas de entrenamiento usadas en BP. Los valores de α y η se han tomado de BP.

Algoritmo Genético+Levenberg–Marquardt

En **GA+LM** se codifican los individuos igual que en **GA+BP**: usando 16 bits para representar los pesos y umbrales en el intervalo $[-1, 1]$ y colocándolos en el cromosoma de la misma forma. La función de aptitud es la inversa del SEP para el conjunto de patrones de entrenamiento (Ecuación 4.7).

El algoritmo GA+LM es similar a un Algoritmo Genético pero sustituye la mutación por una época de entrenamiento con LM. Este entrenamiento se hace con probabilidad p_t sobre uno de los individuos que genera la recombinación. La Tabla 4.5 muestra los parámetros del algoritmo para cada instancia.

La probabilidad p_t se ha elegido para que el número medio de épocas de entrenamiento con LM en GA+LM coincida con el número de épocas de entrenamiento usadas en LM. Los valores de α, μ, β y μ_{max} han sido tomados de LM.

	BC	**DI**	**HE**	**GE**	**SO**	**TH**
Tamaño de población	64					
Selección	Ruleta (2 individuos)					
Recombinación	Recombinación de 1 punto ($p_c = 1.0$)					
Reemplazo	$(\mu + \lambda)$					
Criterio de parada	Alcanzar 1000 épocas					
p_t	1.0	1.0	0.5	0.019	0.018	0.063
α	1.0	1.0	1.0	1.0	1.0	0.1
μ	0.001	0.001	0.001	0.001	0.001	0.001
β	10	10	10	10	10	10
μ_{max}	10^{10}	10^{10}	10^{10}	10^{10}	10^{10}	10^{10}

Tabla 4.5: Parámetros del algoritmo GA+LM.

4.2 Problema ECC

4.2.1 Instancias

La instancia que hemos resuelto del ECC está formada por $M = 24$ palabras de $n = 12$ bits. Se sabe que la distancia máxima para un código con esos parámetros es $d = 6$ [AVZ01].

4.2.2 Evaluación

El parámetro que nos interesa maximizar es la mínima distancia de Hamming. Como el valor óptimo de este parámetro es conocido, para evaluar el resultado de los algoritmos hemos medido el porcentaje de éxito y la media, la desviación típica y el valor mínimo del número de épocas que necesitan para encontrar la solución. En el cálculo de estos valores estadísticos se han tenido en cuenta únicamente las ejecuciones que tuvieron éxito. Las épocas ejecutadas por el algoritmo es la suma de las épocas ejecutadas por cada uno de sus subalgoritmos.

4.2.3 Algoritmos

Para resolver el problema se han usado seis Algoritmos Genéticos Paralelos (PGAs). Tres de ellos usan un operador de mutación por inversión de bits y los otros tres em-

plean el algoritmo heurístico de Repulsión como búsqueda local. Para cada una de estas dos configuraciones se han usado 5, 10 y 15 islas. La topología de comunicación de los subalgoritmos es un anillo unidireccional. Para nombrar estos seis algoritmos usaremos la notación PGA<configuración><islas>. Así por ejemplo PGAmut15 es el Algoritmo Genético Paralelo que usa mutacíon y 15 islas. Los individuos son cadenas binarias formadas por la concatenación de todas las palabras del código. Para la instancia resuelta los individuos tienen una longitud de $24 \times 12 = 288$ bits.

Antes de emplear los PGAs para resolver el problema se usaron varios algoritmos genéticos secuenciales dando muy malos resultados; en raras ocasiones se encontraba el óptimo. Es por ello que decidimos abordar el problema usando PGAs.

Podríamos usar la mínima distancia de Hamming entre dos palabras como aptitud de una solución pero esta función es poco fina, ya que asignará la misma aptitud a un código que tenga las palabras muy separadas salvo dos de ellas y a otro que tenga las palabras muy cercanas. Hay una función de aptitud más fina que está inspirada en la ecuación de la energía potencial electrostática de un sistema de partículas cargadas con la misma carga. Esta función aparece en [DJ90] y su expresión es:

$$f(\mathbf{x}) = \frac{1}{\sum_{i=1}^{M} \sum_{j=1, j \neq i}^{M} \frac{1}{d_{ij}^2}} \tag{4.8}$$

donde d_{ij} es la distancia de Hamming entre las palabras i y j del código que se encuentra en el cromosoma $\mathbf{x}$. Esta función es más fina pero aún presenta un problema: puede asignar mayor aptitud a un código que tiene menor distancia mínima que otro, como se prueba a continuación.

Sean $C1$ y $C2$ los códigos de $n = 10$ bits y $M = 3$ palabras que se muestran a continuación:

$$C1 = \{0000000000, 0000011111, 0011100111\}$$

$$C2 = \{0000000000, 0000001111, 1111110011\}$$

La distancia mínima del primero es $d(C1) = 5$ y la del segundo $d(C2) = 4$, sin embargo sus aptitudes de acuerdo a la Ecuación 4.8 son $f(C1) = 4.639$ y $f(C2) = 5.333$.

Para evitar este problema hemos añadido a la Ecuación 4.8 un sumando que es creciente con respecto a la distancia mínima de Hamming. La función de aptitud que hemos

usado en los algoritmos es:

$$f(\mathbf{x}) = \frac{1}{\sum_{i=1}^{M}\sum_{j=1,j\neq i}^{M}\frac{1}{d_{ij}^2}} + (\frac{d_{min}}{12} - \frac{d_{min}^2}{4} + \frac{d_{min}^3}{6}) \tag{4.9}$$

El tamaño total de la población en todos los algoritmos es de 480 individuos repartidos equitativamente entre las islas. La migración es asíncrona. Cada 11 épocas se escoge un individuo de la población por torneo binario y se envía al siguiente subalgoritmo del anillo. En cada época se comprueba si se han recibido individuos por la red. Si es así se incorporan si son mejores que los peores individuos de la población. Los subalgoritmos se detienen cuando alcanzan las 100000 épocas o alguno de ellos encuentra el óptimo. Los parámetros de los algoritmos PGAmut$n$ se encuentran en la Tabla 4.6 y los de los algoritmos PGArep$n$ en la Tabla 4.7.

	PGAmut5	PGAmut10	PGAmut15
Número de islas	5	10	15
Tamaño de subpoblación	96	48	32
Selección	Torneo binario (2 individuos)		
Recombinación	Recombinación de 1 punto ($p_c = 1.0$)		
Mutación	Por inversión de bits ($p_m = 0.003$)		
Reemplazo	$(\mu + \lambda)$		
Topología	Anillo unidireccinal		
Tipo de migración	Asíncrona		
Hueco entre migraciones[4]	10		
Selección para migración	Torneo binario (1 individuo)		
Reemplazo de migración	Si es mejor		
Criterio de parada	Encontrar óptimo o alcanzar 100000 épocas		

Tabla 4.6: Parámetros de los algoritmos PGAmut$n$.

[4]Por hueco entre migraciones se entiende el número de épocas del algoritmo que no se migra entre dos épocas que migran. Un hueco de 0 sería migrar en todas las épocas.

	PGArep5	PGArep10	PGArep15
Número de islas	5	10	15
Tamaño de subpoblación	96	48	32
Selección	Torneo binario (2 individuos)		
Recombinación	Recombinación de 1 punto ($p_c = 1.0$)		
Búsqueda local	Repulsión moviendo 1 palabra		
Reemplazo	$(\mu + \lambda)$		
Topología	Anillo unidireccinal		
Tipo de migración	Asíncrona		
Hueco entre migraciones	10		
Selección para migración	Torneo binario (1 individuo)		
Reemplazo de migración	Si es mejor		
Criterio de parada	Encontrar óptimo o alcanzar 100000 épocas		

Tabla 4.7: Parámetros de los algoritmos PGArep$n$.

4.3 Problema TA

4.3.1 Instancias

Las instancias para el problema de asignación de terminales usadas en este proyecto han sido algunas de las utilizadas en [ASW94], concretamente las diez instancias con 100 terminales cada una. Los ficheros están divididos en tres secciones encabezadas por una línea con una palabra entre corchetes. La primera sección es la de terminales, encabezada por la palabra TERMINALS. La primera línea de esta sección contiene un número que indica los terminales que hay en el problema (100 en las instancias que hemos resuelto). Después, cada línea se refiere a un terminal y en ellas aparecen tres valores enteros: las coordenadas del terminal y su capacidad requerida. Las coordenadas se encuentran en el rango $[0-100]$ y las capacidades en $[1-6]$. La segunda sección es la de concentradores, encabezada por la palabra CONCENTRATORS. En ella aparece en primer lugar el número de concentradores y después, en cada línea, las coordenadas de éstos (como antes, números enteros en el rango $[0-100]$). El número de concentradores varía de una instancia a otra desde 27 a 33. La tercera y última sección está encabezada por la palabra CONCENTRATORS_CAPACITY y contiene únicamente un número entero que es la capacidad soportada por todos los concentradores. En las instancias resueltas esta capacidad es 12.

Las abreviaturas empleadas para las instancias son de la forma TAi con i variando entre 0 y 9.

4.3.2 Evaluación

Para evaluar una solución hemos usado su coste y su validez. El coste de una solución es la suma de todos los costes de las asignaciones de terminales a concentradores. Los terminales y concentradores se encuentran situados en un plano bidimensional y el coste asociado a cada par terminal–concentrador es la parte entera de la distancia euclídea entre el terminal y el concentrador. Una solución es válida si no presenta sobrecarga en ningún concentrador. Un concentrador se dice que está sobrecargado si la suma de las capacidades requeridas por los terminales asignados a él supera su capacidad soportada.

4.3.3 Algoritmos

Para resolver las instancias se han empleado un Algoritmo Genético y el Algoritmo Greedy presentado en la Sección 3.7.

Algoritmo Genético

Los individuos del Algoritmo Genético representan a las soluciones del problema por medio de permutaciones de los terminales. Esta representación aparece en [ASW94], donde es llamada LC2 (Least Cost Permutation Enconding). Para averiguar la asignación de terminales que representa una permutación se toman los terminales siguiendo el orden indicado por la permutación y se asignan al concentrador factible más cercano (véase la Sección 3.7). Si para algún terminal no se puede encontrar ningún concentrador factible, la solución que representa la permutación es inválida. En este caso el terminal es asignado al concentrador más cercano, provocando una sobrecarga en él que será penalizada.

El objetivo es minimizar una función compuesta por dos sumandos: el coste de la solución y una penalización para las soluciones inválidas. La penalización P se calcula de la siguiente forma:

$$P = \begin{cases} 0 & \text{si la solución es válida} \\ d_{max} \cdot T + O_n \cdot O_c & \text{en otro caso} \end{cases}$$

donde T es el número de terminales, d_{max} es la distancia máxima entre un terminal y un concentrador, O_n es el número de concentradores sobrecargados y O_c es la sobrecarga total (suma de la sobrecarga de todos los concentradores sobrecargados). Con el primer sumando de la penalización, que es constante y depende de la instancia, se asegura que la peor solución válida recibirá menor valor que la mejor de las soluciones inválidas[5]. Con el segundo sumando se penaliza más a las soluciones con más sobrecarga. Puesto que nuestro Algoritmo Genético trata de maximizar y no minimizar la función de aptitud, la expresión que usamos como aptitud de un individuo es:

$$f(\mathbf{x}) = \frac{1}{C(\mathbf{x}) + P(\mathbf{x})} \tag{4.10}$$

donde $\mathbf{x}$ representa una solución al problema y $C(\mathbf{x})$ y $P(\mathbf{x})$ representan el coste y la penalización de esa solución, respectivamente.

La inicialización de los individuos se hace partiendo de la permutación identidad (los terminales ordenados por su identificador) y aplicándole 100 intercambios aleatorios. Los parámetros del algoritmo se resumen en la Tabla 4.8.

Inicialización	100 Intercambios
Tamaño de población	128
Selección	Ruleta (2 individuos)
Recombinación	PMX ($p_c = 1.0$)
Mutación	Intercambio ($p_m = 0.2$)
Reemplazo	$(\mu + \lambda)$
Criterio de parada	Alcanzar 100000 épocas

Tabla 4.8: Parámetros del GA para TA.

Algoritmo Greedy para TA

El algoritmo greedy se ha usado siguiendo la estrategia de no admitir soluciones inválidas, es decir todas las soluciones que devuelve son soluciones válidas para la instancia.

[5]Una solución inválida es mejor que otra si su coste es menor

4.4 Problema SWENG

4.4.1 Instancias

La instancia del problema de Ingeniería del Software que hemos usado se corresponde con la primera instancia empleada en [CCZ01] (pág. 121) compuesta por 18 tareas y con 10 empleados. Su TPG es el que se muestra en la Figura 4.1. Las habilidades requeridas por las tareas y el esfuerzo necesario para llevarlas a cabo se muestran en la Tabla 4.9. Las habilidades que posee cada empleado y su salario se muestran en la Tabla 4.10.

Tarea	**Esfuerzo**	**Habilidades**	**Tarea**	**Esfuerzo**	**Habilidades**
T_0	10	0,2	T_9	20	2,4
T_1	15	2,3	T_{10}	10	0,1
T_2	20	4	T_{11}	15	2,4
T_3	10	0,2	T_{12}	20	3,4
T_4	15	1,2,3	T_{13}	25	1,4
T_5	20	1,2	T_{14}	15	3,4
T_6	10	1,3	T_{15}	10	1,3
T_7	15	0,4	T_{16}	15	1,4
T_8	20	2,3	T_{17}	10	1,2

Tabla 4.9: Esfuerzo y habilidades requeridas por las tareas.

Empleado	**Habilidades**	**Salario**	**Empleado**	**Habilidades**	**Salario**
E_1	2,3	5000	E_6	1,2,3	6000
E_2	0,2	4000	E_7	1,2	6000
E_3	1	3000	E_8	1,2,3	5000
E_4	2,3,4	5000	E_9	1,2	8000
E_5	1	3000	E_{10}	0,1,4	9000

Tabla 4.10: Habilidades y salario de los empleados.

Como se dijo en la Sección 2.4, los empleados pueden dedicarse a más de una tarea pero la dedicación no es del tipo todo o nada: los empleados tienen asociado un grado de dedicación con respecto a todas las tareas. Así, si el empleado E_i se dedica a la tarea T_j con un grado de dedicación de 0.5 significa que el empleado pasa la mitad de su jornada realizando la tarea T_j. Un grado de dedicación 0 indica que el empleado no se dedica a la

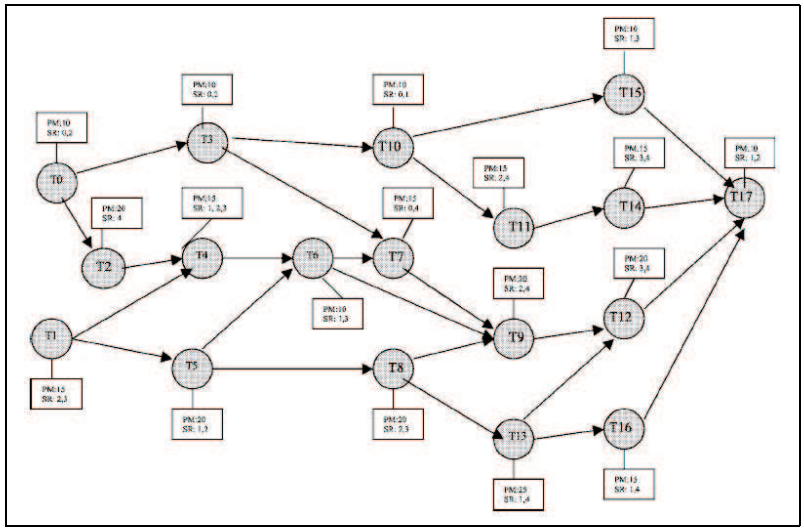

Figura 4.1: TPG de la instancia resuelta para SWENG.

tarea correspondiente y un grado 1 indica que el empleado se dedica durante su jornada laboral a esa tarea. Un grado de dedicación mayor que uno indica que el empleado está dedicando a esa tarea más tiempo de una jornada. Teniendo en cuenta esto, las soluciones al problema son tablas cuyas filas están asociadas a los empleados y las columnas a las tareas. En la casilla (i, j) se encuentra el grado de dedicación del empleado E_i a la tarea T_j. Los valores de las casillas están discretizados y se encuentran en el rango $[0, 1]$ con igual separación entre los valores discretos. En nuestro caso hemos usado 5 valores discretos: 0.00, 0.25, 0.50, 0.75, 1.00 (ver Tabla 4.11).

	T1	T2	T3	T4
E1	0.50	0.25	0.00	0.25
E2	0.00	0.75	0.50	0.25

Tabla 4.11: Un ejemplo de solución.

4.4.2 Evaluación

Para evaluar una solución tenemos en cuenta cuatro aspectos: validez de la solución, duración del proyecto, coste del proyecto y sobrecarga (véase la Sección 2.4). Este es un problema multiobjetivo. Ahora pasamos a detallar la forma de calcular estos valores a

partir de una solución.

Para computar la validez de una solución debemos primero comprobar que todas las tareas están asignadas a alguien. Para ello, basta con comprobar que las sumas de las columnas de la tabla son todas mayores que cero. El segundo requisito para la validez es la posesión de las habilidades necesarias. Por cada tarea calculamos la unión de los conjuntos de habilidades de los empleados que tienen un grado de dedicación mayor que cero para esa tarea y comprobamos que las habilidades requeridas por la tarea están incluidas en dicha unión.

Para calcular el tiempo empleado en la realización del proyecto es necesario conocer el tiempo empleado para realizar cada tarea. Ese tiempo es calculado como el esfuerzo necesario por la tarea dividido por la dedicación total a esa tarea (que es la suma de los elementos de la columna asociada a la tarea). Una vez obtenidos los tiempos, se calcula el tiempo del proyecto basándose en el TPG mediante el algoritmo de la Figura 4.2.

```
dur_proy := 0;
temporizadas := ∅
PARA CADA T_i CON Pred(T_i) = ∅ HACER
    ini(T_i) := 0;
    fin(T_i) := ini(T_i) + duración (T_i);
    SI (fin(T_i) > dur_proy) ENTONCES
        dur_proy := fin(T_i);
    FINSI;
    temporizadas := temporizadas ∪ {T_i};
FINPARA;

PARA CADA T_i CON Pred(T_i) ∈ temporizadas HACER
    ini(T_i) := max {fin(T_j) | T_j ∈ Pred(T_i) };
    fin(T_i) := ini(T_i) + duración (T_i);
    SI (fin(T_i) > dur_proy) ENTONCES
        dur_proy := fin(T_i);
    FINSI;
FINPARA;
```

Figura 4.2: Algoritmo para calcular los instantes de inicio y fin de tareas.

Para calcular el coste del proyecto se suman los salarios que hay que pagar a los

empleados por su dedicación al proyecto. Para cada empleado este dato se calcula multiplicando su salario mensual (que es un dato conocido) por el tiempo que ha dedicado al proyecto. Este tiempo se calcula sumando el tiempo dedicado a cada tarea que es a su vez el producto de su grado de dedicación a la tarea por la duración de la misma.

Por último el cálculo de la sobrecarga requiere conocer el instante de comienzo y de finalización de cada tarea suponiendo que ninguna tarea se retrasa y que cada tarea es comenzada en cuanto todas sus precedentes en el grafo se han completado. Estos instantes de inicio y fin de cada tarea se calculan a la vez que la duración del proyecto en el algoritmo de la Figura 4.2 (de hecho, la duración del proyecto coincide con el mayor instante de finalización calculado). Una vez obtenidos estos datos creamos para cada empleado su función de trabajo, que permite conocer la carga de trabajo de cada empleado en cualquier momento durante el desarrollo del proyecto. Si en algún momento la carga supera la dedicación máxima del empleado se produce una sobrecarga de trabajo. Esta sobrecarga se integra con respecto al tiempo para calcular el trabajo extra del empleado. El valor de la sobrecarga total S es la suma de los trabajos extra de cada empleado como indica la Ecuación 4.11.

$$S = \sum_{i=1}^{m} S_i \tag{4.11}$$

$$S_i = \int_{t=0}^{t=t_f} ramp(load_i(t) - L_i)dt \tag{4.12}$$

donde S_i es el trabajo extra del empleado E_i, L_i es su dedicación máxima, $load_i(t)$ es su función de trabajo, m es el número de empleados y t_f es la duración del proyecto. La función $ramp$ es la función rampa que viene dada por la siguiente expresión:

$$ramp(x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases}$$

4.4.3 Algoritmos

Para resolver el problema se ha usado un Algoritmo Genético. Para representar las soluciones hemos usado cadenas binarias. La tabla de solución se almacena por filas en el cromosoma y para cada valor discreto se usa una secuencia de tres bits (puesto que

hemos usado cinco valores discretos). La relación entre las secuencias de tres bits y los valores discretos se encuentran en la Tabla 4.12.

Cadena	**Valor**
000	0.00
001	0.25
010	0.50
011	0.75
100	1.00
101	0.00
110	0.25
111	0.50

Tabla 4.12: Relación entre las cadenas de tres bits y los valores que representan.

Para calcular la aptitud de una solución se normalizan la duración, el coste y la sobrecarga de la solución dividiéndolos por la máxima duración, coste y sobrecarga presentes entre los individuos de la población, respectivamente. En el caso de que la sobrecarga máxima sea cero ésta no se normaliza (en ese caso la sobrecarga del individuo será también cero). Una vez hecha la normalización, la función de aptitud de las soluciones válidas se calcula haciendo la suma ponderada de la inversa del coste, la inversa de la duración y la inversa de la sobrecarga más 0.5 (esto se hace para evitar la división por cero cuando la sobrecarga es nula). Si la solución no es válida su aptitud es 0.001. Todo esto se resume en las siguientes expresiones:

$$f(\mathbf{x}) = \begin{cases} \frac{w_{dur}}{dur_{norm}} + \frac{w_{cos}}{cos_{norm}} + \frac{w_{sob}}{sob_{norm}+0.5} & \text{si la solución es válida} \\ 0.001 & \text{en otro caso} \end{cases} \tag{4.13}$$

$$dur_{norm} = \frac{dur}{dur_{max}} \tag{4.14}$$

$$cos_{norm} = \frac{cos}{cos_{max}} \tag{4.15}$$

$$sob_{norm} = \begin{cases} \frac{sob}{sob_{max}} & \text{si } sob_{max} > 0 \\ sob & \text{en otro caso} \end{cases} \tag{4.16}$$

donde dur_{max}, cos_{max}, sob_{max} son la duración, coste y sobrecarga máximas presentes entre los individuos de la población; *dur*, *cos*, *sob* son la duración, el coste y la sobrecarga del proyecto y w_{dur}, w_{cos}, w_{sob} son los pesos que ponderan cada uno. A estos pesos se les puede asignar distintos valores dependiendo del interés que tengamos en minimizar cada componente. Para las pruebas nosotros hemos usado $w_{dur} = 2.0$, $w_{cos} = 0.7$, $w_{sob} = 4.0$ y con eso queda determinada completamente la función de aptitud.

Los parámetros del algoritmo genético empleado se presentan en la Tabla 4.13.

Tamaño de población	64
Selección	Torneo binario (2 individuos)
Recombinación	1 punto para 2–D($p_c = 1.0$)
Mutación	Inversión de bits ($p_m = 0.002$)
Reemplazo	$(\mu + \lambda)$
Criterio de parada	Alcanzar 5000 épocas

Tabla 4.13: Parámetros del Algoritmo Genético para SWENG.

4.5 Problema VRP

4.5.1 Instancias

Para el VRP se han resuelto las 14 instancias de Christofides [CMT79]. Estos problemas son muy conocidos en el ámbito del VRP y son usados a modo de benchmark. Las instancias de Christofides se encuentran en 14 ficheros de texto plano siguiendo el formato que se presenta Figura 4.3.

Los archivos de las instancias de Christofides contienen una secuencia de números separados por espacios, tabuladores o retornos de carro. En primer lugar encontramos el número de clientes (sin contar el almacén) que posee el problema. Luego de define la capacidad de los vehículos, que es la misma para todos. Los siguientes dos números son el máximo coste que puede tener una ruta y el coste de servicio (el mismo para todos los clientes), respectivamente. A continuación se define la posición del almacén en un espacio de dos dimensiones y en coordenadas cartesianas. Con esto acaba la parte de longitud fija del fichero. Seguidamente aparece una parte de longitud variable en la que se definen la posición en el plano y la demanda de cada cliente. Por cada uno de ellos hay tres

```
<problema> ::= <num.  clientes> <capacidad>
               <max.  coste ruta> <coste servicio>
               <coord.  x almacén> <coord.  y almacén>
               <lista clientes>


<lista clientes> ::= <datos cliente> |
                     <lista clientes> <datos cliente>


<datos cliente> ::= <coord.  x cliente> <coord.  y cliente>
                    <demanda>
```

Figura 4.3: Formato de los archivos de Christofides.

números: las coordenadas y la demanda. Con esto acaba el fichero. El coste de viajar de un cliente a otro se calcula mediante la distancia euclídea entre cada par de clientes.

Las abreviaturas que usaremos para las instancias de este problema son de la forma VRi con i variando entre 1 y 14.

Las instancias de Christofides se pueden organizar en varios grupos. En primer lugar tenemos las cinco primeras, que no tienen restricciones de coste máximo. Las cinco siguientes son iguales que las primeras pero con coste máximo. Por último, las intancias VR11 y VR13 son similares salvo por las restricciones de coste de la VR13. Lo mismo ocurre con las instancias VR12 y VR14.

4.5.2 Evaluación

Para evaluar las soluciones se emplea el coste de la solución que es la suma de los costes de las rutas que forman parte de la solución. El coste de una ruta es la suma de los costes c_{ij} de los arcos de la ruta más los costes de servicio f_i de sus vértices.

4.5.3 Algoritmos

Hemos resuelto las instancias usando el algoritmo Savings combinado con 3–opt y un Algoritmo Genético.

Savings+3–opt

La combinación de algoritmos Savings+3–opt consiste en aplicar el algoritmo Savings a una instancia y al resultado aplicarle el algoritmo de mejora λ–opt con $\lambda = 3$.

Algoritmo Genético

Para representar las soluciones en el Algoritmo Genético se han usado permutaciones de los clientes. Para formar las rutas se recorre la permutación añadiendo los clientes a una ruta. Cuando la adición de un nuevo cliente provoca que la ruta incumpla alguna de las restricciones de capacidad o coste se inicia una nueva ruta donde se introduce el cliente y se da por finalizada la ruta anterior. La aptitud de un individuo es la inversa de su coste:

$$f(\mathbf{x}) = \frac{1}{C(\mathbf{x})} \tag{4.17}$$

donde $C(\mathbf{x})$ es el coste de la solución $\mathbf{x}$.

Los parámetros del algoritmo genético empleado se encuentran en la Tabla 4.14. La inicialización de los individuos se hace partiendo de la permutación identidad (los clientes ordenados por su identificador) y aplicándole 100 intercambios aleatorios.

Inicialización	100 Intercambios
Tamaño de población	100
Selección	Ruleta (200 individuos)
Recombinación	PMX($p_c = 1.0$)
Mutación	Intercambio ($p_m = 1.0$)
Reemplazo	$(\mu + \lambda)$
Criterio de parada	Alcanzar 10000 épocas

Tabla 4.14: Parámetros del Algoritmo Genético para VRP.

Capítulo 5

Resultados

En este capítulo presentaremos los resultados obtenidos para los problemas por los algoritmos que detallamos en el Capítulo 4. Dedicaremos una sección a cada problema. Todas las pruebas realizadas se hicieron bajo Linux en Pentium III entre 550 y 733 MHz y Pentium 4 a 2.4GHz.

5.1 Problema ANN

En esta sección se muestran los resultados obtenidos entrenando las redes neuronales para los problemas de clasificación enunciados. Para cada combinación de instancia, algoritmo y método de evaluación se han lanzado 30 ejecuciones y se ha calculado la media, la desviación estándar y el mejor valor para el porcentaje de error cuadrático (SEP) y el porcentaje de error de clasificación (CEP). Los resultados se presentan en una tabla que contiene todos esos valores calculados. Las abreviaturas empleadas en las tablas son las siguientes:

- **TT**: Evaluación usando Training & Test.
- **CV5**: Evaluación usando 5–fold Cross–Validation.
- **BP**: Algoritmo de Retropropagación.
- **LM**: Algoritmo de Levenberg–Marquardt.
- **GABP**: Algoritmo Genético con búsqueda local usando Retropropagación.

- **GALM**: Algoritmo Genético con búsqueda local usando Levenberg–Marquardt.
- **ES**: Estrategia Evolutiva.
- **CEP**: Porcentaje de error en la clasificación.
- **SEP**: Porcentaje de error cuadrático.
- $\overline{\mathbf{x}}$: Media.
- $\sigma_{\mathbf{n}}$: Desviación típica.
- **min**: Valor mínimo.

A continuación presentaremos y comentaremos los resultados obtenidos en cada instancia y después haremos unas valoraciones generales.

5.1.1 Cancer

Aquí se muestran los resultados obtenidos para cada combinación de algoritmo y método de evaluación en la instancia Cancer. Las Tablas 5.1 y 5.2 muestran la media ($\overline{\mathbf{x}}$), la desviación típica ($\sigma_{\mathbf{n}}$) y el mejor valor (**min**) del Porcentaje de Error de Clasificación (CEP) y el Porcentaje de Error Cuadrático (SEP) para las 30 ejecuciones realizadas.

		TT				**CV5**			
		BP	**LM**	**GABP**	**GALM**	**BP**	**LM**	**GABP**	**GALM**
CEP	$\overline{\mathbf{x}}$	0.98	3.03	0.13	0.08	3.70	6.77	3.89	3.20
	$\sigma_{\mathbf{n}}$	0.26	1.34	0.24	0.25	0.08	2.30	0.24	0.29
	min	0.57	1.15	0.00	0.00	3.45	5.02	3.45	2.73
SEP	$\overline{\mathbf{x}}$	0.50	3.03	7.70	4.84	2.66	6.48	11.48	6.35
	$\sigma_{\mathbf{n}}$	0.02	1.37	0.48	0.46	0.02	1.87	0.35	0.14
	min	0.46	1.14	6.50	4.28	2.62	4.73	10.86	6.10

Tabla 5.1: Resultados para la instancia Cancer (1).

A la vista de los resultados, los algoritmos híbridos junto con ES son los mejores para esta instancia en cuanto al error de clasificación, llegando incluso a conseguir en algunos casos un error nulo en TT. Si pasamos a evaluar el SEP, la ES mantiene los buenos

ES		TT	CV5
CEP	$\overline{\mathbf{x}}$	0.90	3.87
	$\sigma_{\mathbf{n}}$	0.38	0.49
	min	0.00	2.73
SEP	$\overline{\mathbf{x}}$	0.85	3.20
	$\sigma_{\mathbf{n}}$	0.33	0.29
	min	0.53	2.66

Tabla 5.2: Resultados para la instancia Cancer (2).

resultados pero los híbridos son superados por los algoritmos clásicos de entrenamiento (BP y LM).

Algo que podemos deducir es que la relación entre el SEP y el CEP no es directa; el descenso de uno no significa el descenso del otro. No debemos olvidar que el objetivo de los algoritmos es minimizar el valor de SEP; en este sentido los algoritmos que mejor han hecho su trabajo han sido los clásicos y las ES, que son los que han trabajado con los pesos usando representación real. No obstante, esta afirmación hay que hacerla con reservas ya que durante el entrenamiento se intenta minimizar el SEP de los patrones del conjunto de entrenamiento y para hacer la tabla de resultados se ha medido el SEP de los patrones del conjunto de test. Los algoritmos híbridos, además de discretizar los valores que pueden tomar los pesos, los confinan dentro de un intervalo, lo cual puede explicar los malos resultados obtenidos para el SEP.

Si comparamos los resultados de los dos métodos de evaluación vemos que CV5 es mucho más exigente que TT en esta instancia. Esto no significa que siempre sea así, puesto que depende del conjunto de patrones. En CV5 se usan más patrones para entrenar que en TT y, en general, ante patrones suficientemente heterogéneos eso se traduce en una mayor lentitud de la red para aprenderlos. Nosotros hemos entrenado siempre el mismo número de épocas en TT y CV5; por lo que el motivo de los peores resultados de CV5 puede ser que la red no ha tenido tiempo suficiente para aprender los patrones.

Por último destacar que la desviación típica tanto del SEP como del CEP en el algoritmo LM es la más alta de todas. El algoritmo LM es determinista y la única variación entre una ejecución y otra del algoritmo se encuentra en el valor inicial de los pesos y umbrales. El algoritmo es, por tanto, bastante sensible a estos valores iniciales.

5.1.2 Diabetes

Aquí se muestran los resultados obtenidos para cada combinación de algoritmo y método de evaluación en la instancia Diabetes. La Tabla 5.3 muestra la media ($\overline{\mathbf{x}}$), la desviación típica ($\sigma_{\mathbf{n}}$) y el mejor valor (**min**) del Porcentaje de Error de Clasificación (CEP) y el Porcentaje de Error Cuadrático (SEP) para las 30 ejecuciones realizadas.

		TT				**CV5**			
		BP	**LM**	**GABP**	**GALM**	**BP**	**LM**	**GABP**	**GALM**
CEP	$\overline{\mathbf{x}}$	21.81	25.94	36.44	28.84	23.47	26.69	34.89	27.50
	$\sigma_{\mathbf{n}}$	0.37	3.50	0.09	0.91	1.25	1.26	0.07	0.42
	min	20.83	20.83	35.94	26.56	22.78	24.61	34.63	26.68
SEP	$\overline{\mathbf{x}}$	14.48	22.55	21.60	18.37	15.95	22.38	21.32	18.41
	$\sigma_{\mathbf{n}}$	0.07	2.98	0.39	0.11	0.90	1.44	0.19	0.10
	min	14.37	17.13	21.07	18.16	15.57	20.04	21.13	18.31

Tabla 5.3: Resultados para la instancia Diabetes.

En este caso BP ha obtenido mejores valores medios para el CEP y el SEP. Las desviaciones típicas son pequeñas salvo para el algoritmo LM, por eso podemos decir con cierta garantía que BP es mejor que los algoritmos híbridos para esta instancia.

En este caso el método de evaluación no afecta notablemente a los resultados. La cercanía de las medias no permite decir que un método sea más exigente que otro en esta instancia.

Cabe destacar la alta desviación típica del algoritmo LM tal y como ocurría en la instancia Cancer. Sin embargo, esta vez BP se acerca a ella cuando se evalúa con CV5.

5.1.3 Heart

Aquí se muestran los resultados obtenidos para cada combinación de algoritmo y método de evaluación en la instancia Heart. La Tabla 5.4 muestra la media ($\overline{\mathbf{x}}$), la desviación típica ($\sigma_{\mathbf{n}}$) y el mejor valor (**min**) del Porcentaje de Error de Clasificación (CEP) y el Porcentaje de Error Cuadrático (SEP) para las 30 ejecuciones realizadas.

Un detalle que llama la atención en estos resultados es la alta desviación típica del CEP y el SEP en todos los algoritmos menos GALM, el cual obtiene los mejores resultados para esta instancia cuando se evalúa con CV5. La más alta de estas desviaciones la presenta

		TT				CV5			
		BP	**LM**	**GABP**	**GALM**	**BP**	**LM**	**GABP**	**GALM**
CEP	$\overline{\mathbf{x}}$	27.13	34.81	47.83	22.54	26.86	28.80	53.28	16.21
	$\sigma_{\mathbf{n}}$	1.54	3.77	22.72	0.82	8.21	1.74	7.41	0.38
	min	24.78	26.09	21.74	21.30	19.57	24.57	44.67	15.33
SEP	$\overline{\mathbf{x}}$	18.58	33.90	25.05	15.05	18.54	28.00	25.74	12.14
	$\sigma_{\mathbf{n}}$	0.45	3.93	1.78	0.30	4.48	1.69	0.53	0.11
	min	17.75	25.82	21.62	14.72	14.72	23.00	24.71	11.95

Tabla 5.4: Resultados para la instancia Heart.

el CEP para el algoritmo GABP cuando se evalúa con TT. Cuando se evalúa la red con CV5 el algoritmo BP tiene la más alta desviación típica.

En esta instancia el algoritmo que mejores resultados consigue GALM tanto en la evaluación con TT como con CV5. En este último caso, los resultados son mejores.

5.1.4 Gene

Aquí se muestran los resultados obtenidos para cada combinación de algoritmo y método de evaluación en la instancia Gene. La Tabla 5.5 muestra la media ($\overline{\mathbf{x}}$), la desviación típica ($\sigma_{\mathbf{n}}$) y el mejor valor (**min**) del Porcentaje de Error de Clasificación (CEP) y el Porcentaje de Error Cuadrático (SEP) para las 30 ejecuciones realizadas.

		TT				CV5			
		BP	**LM**	**GABP**	**GALM**	**BP**	**LM**	**GABP**	**GALM**
CEP	$\overline{\mathbf{x}}$	7.15	19.70	28.66	18.58	48.09	37.93	57.72	22.05
	$\sigma_{\mathbf{n}}$	24.95	4.39	34.64	8.28	0.00	3.82	11.58	7.02
	min	0.00	11.22	0.00	7.57	48.09	26.46	45.73	11.56
SEP	$\overline{\mathbf{x}}$	32.89	11.03	19.63	10.64	33.07	21.08	24.24	12.83
	$\sigma_{\mathbf{n}}$	1.62	2.10	2.18	3.35	1.87	1.80	1.16	1.58
	min	24.38	7.14	15.50	5.50	30.16	16.08	21.90	10.06

Tabla 5.5: Resultados para la instancia Gene.

Un detalle llamativo de estos resultados podemos verlo en la columna del algoritmo BP con la evaluación TT. La media obtenida para el CEP es la más pequeña de todas, sin embargo, el valor de la desviación típica es muy alto y en algunas ejecuciones se ha

obtenido un error de clasifiación nulo. Esto se debe a que el resultado de las 30 ejecuciones contiene valores bien muy altos o bien muy bajos, no hay valores intermedios. El valor más bajo de la media del CEP contrasta con uno de los valores más altos de la media del SEP. Este contraste es posible debido a la forma de interpretar la salida de la red, la técnica "el ganador se lleva todo". Si hubiésemos usado la técnica de los umbrales la media del CEP sería bastante más alta.

De nuevo BP es el protagonista de la segunda observación. Cuando se evalúa con CV5 siempre llega al mismo CEP. Además, los valores del SEP obtenidos son los más altos de la tabla. Esto puede deberse a una alteración modesta de los pesos.

En esta instancia la evaluación con CV5 consigue desviaciones más bajas. El algoritmo que mejores resultados consigue es GALM, mientras que BP y GABP son los se encuentran a la cola.

5.1.5 Soybean

Aquí se muestran los resultados obtenidos para cada combinación de algoritmo y método de evaluación en la instancia Soybean. La Tabla 5.6 muestra la media ($\overline{\mathbf{x}}$), la desviación típica ($\sigma_{\mathbf{n}}$) y el mejor valor (**min**) del Porcentaje de Error de Clasificación (CEP) y el Porcentaje de Error Cuadrático (SEP) para las 30 ejecuciones realizadas.

		TT				**CV5**			
		BP	**LM**	**GABP**	**GALM**	**BP**	**LM**	**GABP**	**GALM**
CEP	$\overline{\mathbf{x}}$	100.00	53.45	96.98	70.92	87.48	49.08	93.45	73.41
	$\sigma_{\mathbf{n}}$	0.00	22.98	5.02	19.51	2.32	12.08	3.73	7.16
	min	100.00	20.00	77.06	33.53	86.78	29.74	85.44	60.42
SEP	$\overline{\mathbf{x}}$	5.26	4.90	10.91	6.51	5.26	4.32	10.87	6.84
	$\sigma_{\mathbf{n}}$	0.01	3.55	0.27	1.12	0.00	1.54	0.08	0.61
	min	5.22	1.55	10.21	4.39	5.26	2.39	10.76	5.49

Tabla 5.6: Resultados para la instancia Soybean.

Para empezar podemos decir que los algoritmos BP y GABP obtienen los peores resultados, llegando incluso a clasificar erróneamente todos los patrones de test en el caso de BP usando evaluación TT. El algoritmo LM obtiene mejores medias que GALM pero también mayores desviaciones típicas.

5.1.6 Thyroid

Aquí se muestran los resultados obtenidos para cada combinación de algoritmo y método de evaluación en la instancia Thyroid. La Tabla 5.7 muestra la media ($\overline{\mathbf{x}}$), la desviación típica ($\sigma_{\mathbf{n}}$) y el mejor valor (**min**) del Porcentaje de Error de Clasificación (CEP) y el Porcentaje de Error Cuadrático (SEP) para las 30 ejecuciones realizadas.

		TT				**CV5**			
		BP	**LM**	**GABP**	**GALM**	**BP**	**LM**	**GABP**	**GALM**
CEP	$\overline{\mathbf{x}}$	7.28	1.58	7.28	7.28	7.42	1.30	7.42	7.42
	$\sigma_{\mathbf{n}}$	0.00	1.06	0.00	0.00	0.00	0.30	0.00	0.00
	min	7.28	0.83	7.28	7.28	7.42	0.97	7.42	7.42
SEP	$\overline{\mathbf{x}}$	4.85	0.92	7.64	5.76	4.94	0.75	7.91	5.87
	$\sigma_{\mathbf{n}}$	0.00	0.63	0.79	0.88	0.01	0.18	0.44	0.51
	min	4.85	0.55	6.24	4.60	4.90	0.57	6.71	5.16

Tabla 5.7: Resultados para la instancia Thyroid.

El único algoritmo capaz de obtener buenos resultados es LM. Obsérvese que el valor del CEP para todas las ejecuciones de los otros algoritmos siempre es el mismo. Además ese valor depende sólo del método de evaluación. Debe ser el valor máximo de CEP que puede devolver la red para ese conjunto de patrones de test.

5.1.7 Conclusiones generales

El algoritmo BP obtiene buenos resultados para las instancias más ligeras, en concreto obtiene buenos resultados para Cancer, Diabetes y Heart. Estas son instancias que tienen dos salidas (el resto tiene tres como mínimo) y un máximo de 920 patrones (correspondientes a heart).

El algoritmo LM presenta en general altas desviaciones típicas para el CEP y el SEP. Para las tres primeras instancias, en las que el algoritmo BP se comporta bien, el algoritmo LM no supera a BP. Sin embargo, en las instancias pesadas, cuando BP deja de ofrecer buenos resultados, el algoritmo LM se convierte casi en el único capaz de abordar la instancia.

El algoritmo GABP no consigue nunca superar a BP y su coste computacional es más alto. Podemos considerarlo el peor de los algoritmos empleados.

Por último, el algoritmo GALM presenta en general bajas desviaciones típicas, lo cual contrasta con las altas desviaciones de LM. Aunque no obtiene muy buenos resultados en comparación con LM en las últimas instancias, en el resto suele obtener resultados similares o incluso mejores y con una mayor estabilidad (desviación típica baja).

5.2 ECC

En esta sección se muestran los resultados obtenidos para la instancia resuelta del problema de diseño de códigos correctores de errores. Para cada uno de los seis algoritmos empleados hemos realizado 30 ejecuciones. Para realizar las pruebas se han distribuido los subalgoritmos en cinco máquinas, poniendo el mismo número de subalgoritmos en cada máquina. Así por ejemplo, para los algoritmos que tienen 15 islas cada máquina ejecuta tres de esas islas. Hemos medido el porcentaje de éxito y la media ($\overline{\mathbf{x}}$), la desviación típica ($\sigma_{\mathbf{n}}$) y el valor mínimo (**min**) del número de épocas que necesitan para encontrar la solución. En la Tabla 5.8 se encuentran los resultados obtenidos por los algoritmos PGAmut$n$ y en la Tabla 5.9 los obtenidos por los algoritmos PGArep$n$. Las abreviaturas empleadas en las tablas son las siguientes:

- n: Número de islas del algoritmo.
- **PGAmut**n: Algoritmo Genético Paralelo con mutación.
- **PGArep**n: Algoritmo Genético Paralelo con repulsión.
- $\overline{\mathbf{x}}$: Media.
- $\sigma_{\mathbf{n}}$: Desviación típica.
- **min**: Valor mínimo.

Tanto con los algoritmos PGAmut$n$ como con los PGArep$n$ el porcentaje de éxito aumenta con el número de islas. Cada isla se concentra en una región del espacio de búsqueda y el intercambio de individuos entre islas trae genes nuevos a una población. En este caso los genes son palabras del código y puede ocurrir que en una población las palabras se concentren en una región del espacio de búsqueda y en otra población se

	PGAmut$n$			
	%	**Épocas**		
n	**Éxito**	$\overline{\mathbf{x}}$	$\sigma_{\mathbf{n}}$	**min**
15	16.67	373026.00	301809.52	92570.00
10	20.00	352230.50	219170.12	120630.00
5	6.67	173251.00	31149.00	142102.00

Tabla 5.8: Resultados de PGAmut$n$ para el problema ECC.

	PGArep$n$			
	%	**Épocas**		
n	**Éxito**	$\overline{\mathbf{x}}$	$\sigma_{\mathbf{n}}$	**min**
15	93.33	42200.43	25937.40	15329.00
10	80.00	57966.75	64945.30	17464.00
5	43.33	35767.31	22039.52	13664.00

Tabla 5.9: Resultados de PGArep$n$ para el problema ECC.

concentren en otra región. Mediante el intercambio de individuos entre las poblaciones y la recombinación se pueden conseguir códigos con las palabras más separadas y por tanto mejores.

Una de las conclusiones más importantes es que los algoritmos PGArep$n$ obtienen resultados claramente mejores que los algoritmos PGAmut$n$, demostrando la ventaja del algoritmo de repulsión desarrollado frente a la mutación por inversión de bits. La ventaja no sólo se advierte en el porcentaje de éxito sino también en el número de épocas necesarias para encontrar el óptimo.

Por último vamos a comentar de qué forma afecta el incremento en el número de islas al número medio de épocas necesarias para encontrar la solución. Por un lado al haber más islas hay más subalgoritmos ejecutando, lo que contribuye a aumentar el número de épocas necesarias. Pero por otro lado, el mayor número de islas contribuye a una mayor diversidad y la solución puede ser encontrada antes. Por lo tanto, el comportamiento de la media del número de épocas no tiene una tendencia clara, dependerá de otros parámetros del algoritmo. En los algoritmos PGAmut$n$ el número medio de épocas aumenta con las islas, esto significa que el aumento de la diversidad no es suficiente para contrarrestar el aumento del número de subalgoritmos. Por otro lado, en PGArep$n$ el valor máximo de la

media de las épocas se alcanza cuando $n = 10$.

5.3 TA

En esta sección se muestran los resultados obtenidos para las instancias resueltas del problema de asignación de terminales a concentradores. Para cada una de las diez instancias se han lanzado 30 ejecuciones usando el algoritmo genético y hemos tomado la media, la desviación estándar y el mejor valor del coste. El algoritmo greedy se ha ejecutado 20000 veces por cada instancia y se ha tomado la mejor solución. Estos resultados se muestran en la Tabla 5.10 junto con los mejores resultados obtenidos para las instancias en [ASW94]. Las abreviaturas empleadas en la tabla son las siguientes:

- **GA**: Algoritmo Genético.
- **GR**: Algoritmo Greedy para TA.
- **ASW**: Mejor resultado obtenido en [ASW94].
- $\overline{\mathbf{x}}$: Media.
- $\sigma_{\mathbf{n}}$: Desviación típica.
- **min**: Valor mínimo.

Lo primero que debemos señalar es que todas las soluciones obtenidas son válidas. Cabe destacar que la mejor solución obtenida por el algoritmo genético es siempre mejor que la mejor obtenida por el algoritmo greedy. Salvo para la instancia TA7, las mejores soluciones del algoritmo genético superan a las mejores soluciones encontradas en [ASW94]. En dicha referencia esas soluciones fueron obtenidas también mediante un algoritmo genético.

En cuanto a las medias, todas superan al mejor resultado del algoritmo greedy y en al caso de TA1 incluso supera al mejor resultado de [ASW94].

Para la instancia TA0 hemos obtenido una traza de la evolución de la aptitud del peor individuo, el mejor y la media de la población. Para obtenerla hicimos 30 ejecuciones y calculamos la media de ellas. Estas trazas pueden verse en la Figura 5.1.

	GA			GR	
	$\overline{x}$	σ_n	min	min	ASW
TA0	1107.00	11.12	1092.00	1154.00	1099.00
TA1	1143.97	12.84	1126.00	1202.00	1146.00
TA2	1123.26	35.67	1085.00	1264.00	1090.00
TA3	1318.00	23.13	1296.00	1420.00	1303.00
TA4	1440.10	30.21	1413.00	1583.00	1423.00
TA5	1314.32	20.73	1293.00	1407.00	1309.00
TA6	1758.16	36.30	1722.00	1934.00	1725.00
TA7	1668.58	37.99	1637.00	1855.00	1615.00
TA8	1410.55	31.06	1383.00	1563.00	1394.00
TA9	1190.52	18.99	1160.00	1268.00	1182.00

Tabla 5.10: Resultados para el problema TA.

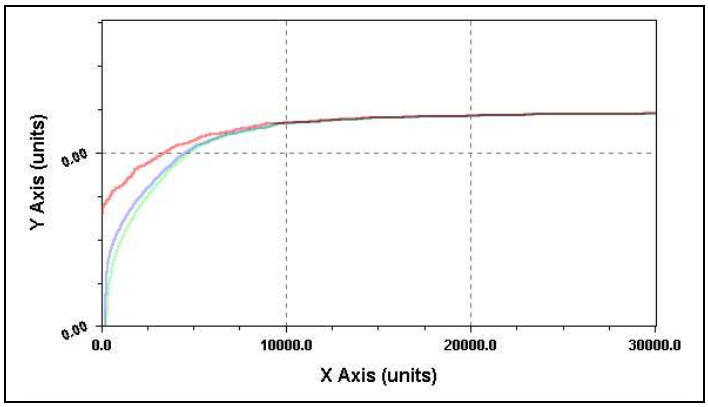

Figura 5.1: Evolución de la aptitud en la población.

Podemos destacar varias cosas sobre la evolución de la población. En primer lugar, las tres curvas son crecientes; esto es debido al elitismo del operador de reemplazo. Las tres curvas se ajustan de acuerdo a una función logística tal y como predice la teoría.

5.4 SWENG

En esta sección se muestran los resultados obtenidos para la instancia resuelta del problema de ingeniería del software. Se han lanzado 30 ejecuciones usando el algoritmo genético y hemos tomado la media, la desviación estándar y el mejor valor del coste de

proyecto y de la duración. La sobrecarga ha sido nula en todas las ejecuciones y por eso no la mostramos. En la Tabla 5.11 se muestran los resultados junto con los valores medios obtenidos en el **test 4** que se encuentra en [CCZ01] para esa misma instancia. Las abreviaturas empleadas en la tabla son las siguientes:

- **GA**: Algoritmo Genético.
- **CCZ**: Resultados de [CCZ01].
- **DUR**: Duración del proyecto.
- **COS**: Coste del proyecto.
- $\overline{\mathbf{x}}$: Media.
- $\sigma_{\mathbf{n}}$: Desviación típica.
- **min**: Valor mínimo.

	GA			**CCZ**
	$\overline{\mathbf{x}}$	$\sigma_{\mathbf{n}}$	**min**	$\overline{\mathbf{x}}$
DUR	37.70	3.41	33.53	≈ 33.00
COS	1399984.61	31350.28	1337920.81	≈ 1610000.00

Tabla 5.11: Resultados para el problema SWENG.

El resultado obtenido de la bibliografía tiene una menor duración media pero mayor coste medio. Eso significa que han resuelto el problema dando mayor importancia a la duración que al coste. La proporción entre el peso asociado a la duración y el peso asociado al coste debe ser mayor a la que nosotros hemos usado.

5.5 VRP

En esta sección se muestran los resultados obtenidos para las instancias resueltas del problema de guiado de vehículos. Se han lanzado 30 ejecuciones usando el algoritmo genético y hemos tomado la media, la desviación estándar y el mejor valor del coste de la solución. Para cada instancia hemos aplicado también Savings+3–opt. En la Tabla 5.12

se muestran los resultados junto con los menores costes conocidos para cada instancia. Estos costes han sido tomados de [LGPS99][1]. Las abreviaturas empleadas en la tabla son las siguientes:

- **GA**: Algoritmo Genético.
- **SA3OPT**: Savings+3–opt.
- **MC**: Menor coste conocido.
- $\overline{\mathbf{x}}$: Media.
- $\sigma_{\mathbf{n}}$: Desviación típica.
- **min**: Valor mínimo.

	GA				
	$\overline{\mathbf{x}}$	$\sigma_{\mathbf{n}}$	**min**	**SA3OPT**	**MC**
VR1	610.09	28.29	565.88	578.56	524.61
VR2	965.60	24.02	901.48	888.04	835.26
VR3	1098.83	53.68	964.81	864.09	826.14
VR4	1566.59	59.17	1443.80	1096.36	1028.42
VR5	2049.99	93.22	1856.05	1402.24	1291.45
VR6	1227.62	29.70	1154.66	1120.65	1055.43
VR7	1907.86	37.52	1790.24	1721.72	1659.63
VR8	2348.26	79.90	2209.34	1952.44	1865.94
VR9	3496.40	93.79	3262.98	2789.87	2662.55
VR10	4617.21	103.20	4389.83	3512.00	3385.85
VR11	1618.33	98.35	1360.38	1046.93	1042.11
VR12	1227.04	71.93	1090.92	824.42	819.56
VR13	8425.37	197.35	8037.98	7575.59	7541.14
VR14	10342.66	72.86	10240.71	9868.50	9866.37

Tabla 5.12: Resultados para el VRP.

[1]En las instancias con coste de servicio distinto de cero el coste de la mejor solución presentada en esta referencia no tiene en cuenta el coste de servicio y su valor es menor. Nosotros hemos añadido el coste de servicio a esos valores para poder comparar los resultados.

Lo primero que podemos observar es que Savings+3–opt consigue mejores resultados que el algoritmo genético. Tan sólo en la instancia VR1 el algoritmo genético ha obtenido una solución mejor que Savings+3–opt. En cuanto a las mejores soluciones conocidas, ninguno de los algoritmos usados llegan a ellas. Tan sólo Savings+3–opt consigue acercarse en las instancias VR11 y VR14.

Además de estas pruebas hicimos algunas ejecuciones usando un algoritmo genético que empleaba el operador de recombinación de aristas (ERX) pero los resultados no fueron muy buenos. También usamos, sin mucho éxito, el algoritmo λ–Intercambio y λ–opt como operadores de búsqueda dirigida del algoritmo genético. Para mejorarlo probamos a introducir en la población inicial la solución obtenida mediante el algoritmo Savings para que partiera de ella y encontrara una mejor solución. Pero esto no fue así y decidimos no seguir por ese camino.

Capítulo 6

Resultados secundarios

En este capítulo presentamos un problema de entrenamiento de redes neuronales y una comparativa entre dos representaciones de permutaciones. Ninguno de los dos forma parte de los objetivos originales del PFC, pero tienen relación con los contenidos del mismo. Dada su relevancia hemos decidido dedicarles un capítulo. Tanto el problema como la comparativa fueron realizados durante el desarrollo del proyecto.

El problema consiste en modelar el crecimiento de los microorganismos en los alimentos y la comparativa de las representaciones se hace resolviendo el problema de guiado de vehículos usando ambas representaciones. Las siguientes secciones profundizan en ambos asuntos.

6.1 Problema del Crecimiento Microbiano (MI)

Este problema consiste en obtener los parámetros de crecimiento de los microorganismos en los alimentos a partir de las condiciones medioambientales en que se encuentran. Estos parámetros son dos: *grow rate* y *lang*. Las condiciones vienen dadas por medio de cuatro parámetros. Todos estos atributos son reales. El conjunto de patrones está formado por 150 patrones y dividido en conjunto de entrenamiento (100 patrones) y conjunto de test (50 patrones).

El preprocesamiento que reciben los patrones consiste en aplicar una función lineal a las cuatro entradas y al *grow rate* para que se encuentren en el intervalo [0,1]. Al *lang* se le aplica la función logaritmo neperiano.

Nosotros hemos usado perceptrones multicapa para aprender los patrones. Para cada parámetro de salida se ha empleado una red distinta en lugar de usar una única red con dos salidas. De esta forma no hay pesos comunes que entorpezcan el proceso de aprendizaje. Hemos empleado dos capas ocultas con 6 neuronas cada una. La capa de entrada tiene 4 neuronas y la de salida 1. La función de activación usada es la logística. Los algoritmos empleados para resolver el problema han sido Levenberg–Marquardt y el híbrido Algoritmo Genético+Levenberg–Marquardt. En el algoritmo híbrido, se aplica con probabilidad p_t una época de entrenamiento usando LM a uno de los individuos resultantes de la recombinación. Los parámetros de los algoritmos se encuentran en las Tablas 6.1 y 6.2 para cada una de las redes entrenadas. Las abreviaturas empleadas en las tablas son las siguientes:

- **LA**: Red neuronal para el parámetro *lang*.
- **GR**: Red neuronal para el parámetro *grow rate*.
- $\overline{\mathbf{x}}$: Media.
- $\sigma_{\mathbf{n}}$: Desviación típica.
- **min**: Valor mínimo.

	LA	**GR**
Épocas	1000	1000
α	1.0	1.0
μ	0.001	0.001
β	10	10
μ_{max}	10^{10}	10^{10}

Tabla 6.1: Parámetros del algoritmo LM para MI.

Al igual que con el problema ANN se ha usado como función de aptitud la inversa del SEP. Los resultados con el valor medio del SEP la desviación estándar y el mínimo se muestran en la Tabla 6.3.

En este caso el mejor algoritmo es LM. Llega siempre al mismo resultado. Durante el entrenamiento siempre alcanza el valor mínimo del SEP para el conjunto de entrenamiento.

	LA	**GR**
Tamaño de población	64	
Selección	Ruleta (2 individuos)	
Recombinación	Recombinación de 1 punto ($p_c = 1.0$)	
Reemplazo	$(\mu + \lambda)$	
Criterio de parada	Alcanzar 1000 épocas	
p_t	1.0	1.0
α	1.0	1.0
μ	0.001	0.001
β	10	10
μ_{max}	10^{10}	10^{10}

Tabla 6.2: Parámetros del algoritmo GA+LM para MI.

		GR		**LA**	
		LM	**GALM**	**LM**	**GALM**
SEP	$\overline{\mathbf{x}}$	0.02	2.33	0.34	2.73
	$\sigma_{\mathbf{n}}$	0.00	0.14	0.00	0.12
	min	0.02	2.19	0.34	2.62

Tabla 6.3: Resultados para MI.

Este valor mínimo no es cero ya que en los patrones no existe una relación funcional entre el vector de entradas y el de salidas. Para un mismo vector de entrada pueden existir varios vectores de salida distintos. Una vez entrenada a red se evalúa con el conjunto de patrones de test y se obtiene el valor de SEP que se muestra en la tabla.

6.2 Permutaciones

Cuando queremos resolver mediante algoritmos genéticos un problema cuyas soluciones se expresan mediante permutaciones debemos emplear operadores específicos. Una alternativa consiste en representar la permutación mediante una cadena binaria. Esta representación no es trivial, puesto que las cadenas usadas deben seguir representando permutaciones tras la aplicación de los operadores de recombincación y mutación del algoritmo genético. Durante la realización del proyecto desarrollamos una de estas representaciones. Los algoritmos para pasar de una a otra representación se encuentran en las

Figuras 6.1 y 6.2, donde las funciones `nat2bin()` y `bin2nat()` realizan la conversión de números naturales a binarios y viceversa, respectivamente.

```
PARA i=n..1 HACER
     val = pi[i];
     MIENTRAS val > i HACER
         val = f[val];
     FINMIENTRAS;
     f[i]=val;
FINPARA;
ind = 0;
bits = 1;
count = 1;
PARA i=1..n HACER
     b[ind:ind+bits-1] = nat2bin(f[i], bits);
     ind = ind + bits;
     count = count - 1;
     SI count = 0 ENTONCES
         count = 1 << bits;
         bits = bits + 1;
     FINSI;
FINPARA;
```

Figura 6.1: Algoritmo para pasar una permutación de representación entera a binaria.

Para probar la representación usamos las 14 instancias del problema de guiado de vehículos usadas en el proyecto. Para resolverlo se utilizó un algoritmo genético con la parametrización que se indica en la Tabla 6.4. La mutación es por inversión de bits y la probabilidad de invertir un bit es la inversa de la longitud de la cadena binaria, que para cada problema es distinta.

Para cada instancia se han lanzado 30 ejecuciones usando el algoritmo genético y hemos tomado la media, la desviación estándar y el mejor valor del coste de la solución. En la Tabla 6.5 se muestran los resultados junto con los mostrados en la Sección 5.5 donde se utilizaba la representación tradicional de las permutaciones como vector de enteros. Las abreviaturas empleadas en la tabla son las siguientes:

- **GA–ENT**: Algoritmo Genético con representación tradicional para las permuta-

```
ind = 0;
bits = 1;
count = 1;
PARA i=1..n HACER
     f[i] = bin2nat (b[ind:ind+bits-1]);
     SI f[i] > i ENTONCES
         f[i] = f[i] - [i+1];
     FINSI;
     ind = ind + bits;
     count = count - 1;
     SI count = 0 ENTONCES
         count = 1 << bits;
         bits = bits + 1;
     FINSI;
FINPARA;
PARA i = 0..n HACER
     pi[i]=i;
FINPARA;

PARA i=n..1 HACER
     val = f[i];
     SI val <> i ENTONCES
        aux = pi[val];
        pi[val] = pi[i];
        pi[i] = aux;
     FINSI;
FINPARA;
```

Figura 6.2: Algoritmo para pasar una permutación de representación binaria a entera.

Tamaño de población	100
Selección	Ruleta (200 individuos)
Recombinación	2 puntos ($p_c = 1.0$)
Mutación	Inversión de bits ($p_m = 1/long$)
Reemplazo	$(\mu + \lambda)$
Criterio de parada	Alcanzar 10000 épocas

Tabla 6.4: Parámetros del Algoritmo Genético para VRP con representación binaria.

ciones.

- **GA–BIN**: Algoritmo Genético con representación binaria para las permutaciones.
- $\overline{\mathbf{VRi}}$: Media de todas las instancias.
- $\overline{\mathbf{x}}$: Media.
- $\sigma_{\mathbf{n}}$: Desviación típica.
- **min**: Valor mínimo.

	GA–ENT			**GA–BIN**		
	$\overline{\mathbf{x}}$	$\sigma_{\mathbf{n}}$	**min**	$\overline{\mathbf{x}}$	$\sigma_{\mathbf{n}}$	**min**
VR1	610.09	28.29	565.88	889.24	39.52	823.47
VR2	965.60	24.02	901.48	1451.83	68.27	1321.98
VR3	1098.83	53.68	964.81	1797.32	67.27	1623.51
VR4	1566.59	59.17	1443.80	2749.60	109.83	2534.39
VR5	2049.99	93.22	1856.05	3732.37	110.80	3456.81
VR6	1227.62	29.70	1154.66	1458.84	52.48	1368.26
VR7	1907.86	37.52	1790.24	2284.17	68.81	2175.22
VR8	2348.26	79.90	2209.34	2950.35	108.74	2766.27
VR9	3496.40	93.79	3262.98	4493.75	115.06	4263.78
VR10	4617.21	103.20	4389.83	5972.46	155.08	5673.07
VR11	1618.33	98.35	1360.38	2708.91	146.52	2429.94
VR12	1227.04	71.93	1090.92	1959.47	107.42	1727.94
VR13	8425.37	197.35	8037.98	9357.44	243.78	8866.92
VR14	10342.66	72.86	10240.71	11047.04	147.26	10725.39
$\overline{\mathbf{VRi}}$	2964.42	74.50	2804.93	3775.20	110.06	3554.07

Tabla 6.5: Resultados para VRP con representación binaria.

Los resultados obtenidos con la representación binaria de las permutaciones son peores que los obtenidos con la representación tradicional. El motivo de los altos costes en GA–BIN es que no usamos operadores que sean equivalentes a los que sabemos que son mejores para las permutaciones (como los empleados en GA–INT). Obsérvese que el mejor coste obtenido por GA–BIN para cada instancia es mayor que el coste medio que obtiene para la misma instancia GA–ENT.

Capítulo 7

Conclusiones y Trabajo Futuro

En este último capítulo indicaremos algunas conclusiones generales y mencionaremos posibles extensiones de este trabajo.

7.1 Conclusiones

Hemos podido comprobar que hay problemas de optimización para los que los algoritmos tradicionales o clásicos no son aplicables; bien porque no pueden resolver el problema o bien porque tienen un coste computacional elevado. Cuando esto ocurre debemos emplear otro tipo de algoritmos que, si bien no siempre encuentran una solución óptima, en la mayoría de los casos obtienen soluciones muy cercanas a la óptima.

A lo largo del proyecto hemos aplicado distintos algoritmos modernos a instancias de problemas que tienen bastante interés por su presencia en el mundo real y por su complejidad. La familia de los algoritmos evolutivos ha sido el denominador común de la resolución de todas las instancias. Hemos aplicado algoritmos evolutivos a todos los problemas de una forma o de otra y los resultados obtenidos demuestran que tienen mucho que decir en el campo de la optimización. Algunos de los resultados obtenidos en el proyecto con estos algoritmos superan los que se pueden leer en la bibliografía. Este era uno de los objetivos del proyecto, tratar de superar las mejores marcas. También hemos desarrollado una herramienta para generar automáticamente instancias del problema de ingeniería del software, aunque no ha sido usada en el proyecto.

El uso de un lenguaje de programación independiente de la plataforma ha facilitado

mucho la labor de implementación y pruebas. Hemos podido ejecutar los algoritmos en distintas máquinas sin tener que cambiar nada. Donde más se notan las ventajas de este lenguaje es cuando conectamos dos máquinas distintas a través de la red sin que debamos resaltar ningún problema. Por ser un lenguaje orientado a objetos, el cambio de los problemas y/o algoritmos se puede hacer de forma sencilla sin tener que modificar el código de ninguno. Un ejemplo de esto lo encontramos en la resolución de los problemas secundarios abordados (crecimiento microbiano y permutaciones con representación binaria) con las herramientas usadas para los problemas principales.

Pero el desarrollo de este trabajo no ha estado exento de dificultades. Debido a problemas técnicos no han podido realizarse pruebas con los algoritmos genéticos paralelos en un entorno WAN. El estudio del comportamiento de los algoritmos en una WAN tiene mucho interés en los tiempos que corren. Mediante el uso de miles de máquinas conectadas en red los algoritmos pueden resolver algunos problemas que de otro modo sería imposible debido a la potencia necesaria. Los resultados de las pruebas realizadas con algoritmos evolutivos paralelos en entornos LAN demuestran que estos algoritmos no son simplemente una versión paralela de un algoritmo secuencial: son algoritmos distintos con características distintas.

El campo de la computación evolutiva está en pleno desarrollo. Hay muchos detalles que deben ser estudiados para poder comprender el comportamiento de estos algoritmos y determinar casi sistemáticamente los parámetros que mejor funcionan con un determinado problema.

7.2 Trabajo Futuro

El presente proyecto ha abierto muchas líneas a partir de las cuales se pueden ir desarrollando nuevos trabajos. En esta sección mencionamos algunas de ellas.

En los resultados obtenidos para el entrenamiento de redes neuronales hemos observado que el algoritmo BP se comporta mal cuando la instancia no es pequeña. Podría hacerse un estudio detallado de por qué ocurre esto y proponer soluciones. Sin abandonar el problema de entrenamiento de redes neuronales se podría analizar qué parámetros de los algoritmos debemos usar en una instancia concreta. Incluso, se podrían proponer nuevos algoritmos basados en el gradiente o en métodos de segundo orden que, siguiendo la idea

de las estrategias evolutivas, autoadapten sus parámetros.

Sobre el problema de diseño de códigos correctores de errores se podría hacer un estudio de cómo afectan los distintos parámetros del algoritmo genético paralelo a su resolución. En particular, podría resolverse el problema en la WAN.

Para el problema de guiado de vehículos se pueden probar otras representaciones o hibridar los algoritmos genéticos con técnicas heurísticas para tratar de mejorar las soluciones.

Se puede hacer un estudio sobre las condiciones en que se consiguen las mejores marcas en las instancias. Se pueden resolver más instancias de estos problemas e incluso de nuevos problemas.

Por último, puede extenderse el software desarrollado con nuevos operadores y/o algoritmos.

Apéndices

Apéndice A

Software de Redes Neuronales

En este capítulo presentaremos el software desarrollado para trabajar con redes neuronales. El objetivo final de dicho software es poder resolver el problema de entrenamiento de ANNs presentado en el Capítulo 2. Para cumplir este objetivo necesitamos en primer lugar una forma de representar las redes neuronales y los patrones con los que deben trabajar. De esto se encargan las clases ubicadas en los paquetes ***pfc.networks*** y ***pfc.networks.ffnn***.

Los patrones deben ser leídos desde algún sitio (normalmente ficheros) y seguirán algún formato. Nuestro software tendrá que ser capaz de interpretar ese formato para poder cargar los patrones correctamente. Para no imponer el formato, los patrones se leen mediante el uso de unos objetos que interpretan los datos procedentes de la fuente de patrones. Las clases que realizan dicha interpretación se han agrupado en el paquete ***pfc.pr***. La implementación se hizo pensando en futuras ampliaciones del abanico de formatos que puede leer el sistema.

El corazón del software está formado por los algoritmos de entrenamiento. Se han implementado los dos explicados en el Capítulo 3 (Retropropagación y Levenberg-Marquardt) y una clase que sirve de nexo con el software de algoritmos evolutivos, permitiendo entrenar las redes usando EAs y algoritmos híbridos. Los paquetes ***pfc.algorithms***, ***pfc.algorithms.bp***, ***pfc.algorithms.marquardt*** y ***pfc.algorithms.ea*** contienen las clases que implementan estos algoritmos.

También hemos implementado una serie de clases que permite ampliar de forma sencilla el abanico de métodos de evaluación que se pueden realizar. Estas clases se encuentran

en el paquete ***pfc.proccess***.

Por último, una vez elegida la red, los patrones, el algoritmo, y el método de evaluación, hace falta ponerlo todo en funcionamiento. Esto se lleva a cabo mediante el uso de las clases que se encuentran en el paquete ***pfc.solvers***.

Después de esta vista de pájaro del software, a continuación vamos a profundizar en cada una de las partes por separado, indicando los detalles de la implementación, métodos, atributos, etc. que sean relevantes.

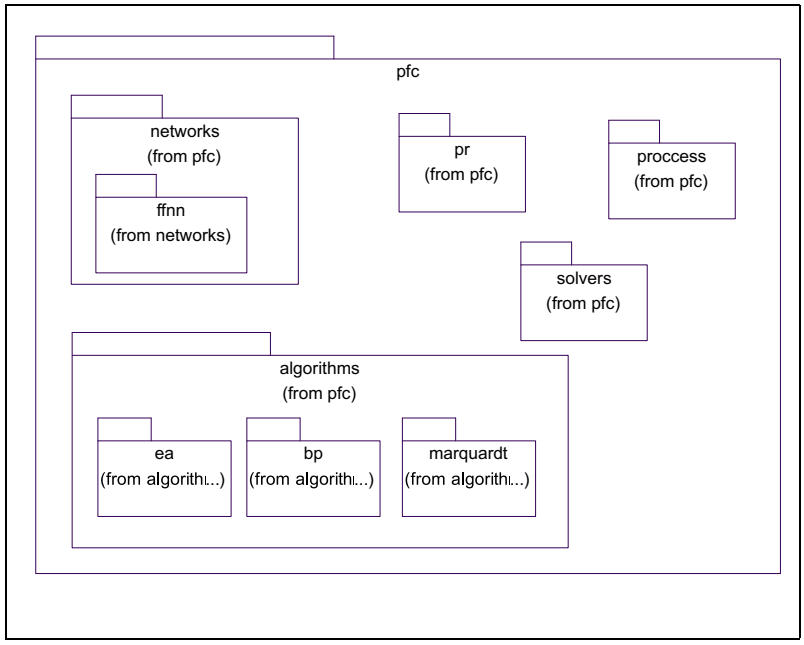

Figura A.1: Un esquema de los paquetes del software de redes neuronales.

A.1 Redes

Los perceptrones multicapa generalizados vienen representados en nuestro software por objetos de la clase **FFNN** dentro del paquete ***pfc.networks.ffnn***. Para especificar una red de este tipo necesitamos:

- La matriz de conectividad, que indica de qué forma están conectadas las neuronas.
- La matriz de pesos, que contiene los pesos sinápticos de las conexiones.

- El vector de umbrales, que almacena los umbrales de las neuronas que no son de entrada.
- El conjunto de neuronas de entrada.
- El conjunto de neuronas de salida.
- Las funciones de activación de cada neurona.

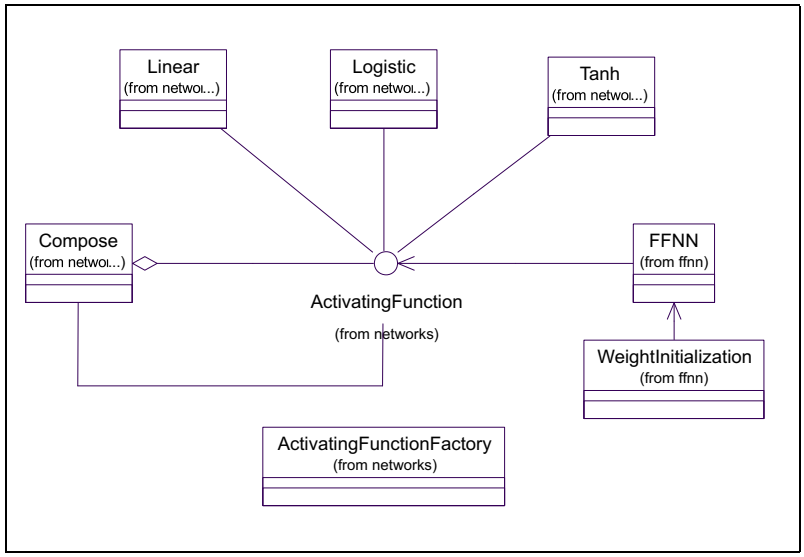

Figura A.2: Clases para representar las redes.

Toda esta información debe proporcionarse al constructor de la clase para crear un objeto que represente a la red. Igualmente, toda esta información está accesible mediante la invocación de ciertos métodos de acceso. Vamos a comentar detalles técnicos sobre cada uno de los anteriores elementos.

La matriz de conectividad es representada mediante una matriz de enteros `con` donde `con[i][j]` es 1 si existe el arco (N_i, N_j) y 0 si no existe dicho arco. El grafo subyacente a un perceptrón multicapa generalizado es acíclico y dirigido. Por lo tanto existe al menos un modo de numerar las neuronas para que sólo existan arcos de la forma (N_i, N_j) con $i < j$ y además las neuronas de entrada reciban los primeros números y las de salida los últimos. Todo el software asume que se emplea tal numeración, es decir, se asume que `con[i][j]`=0 si $i \geq j$. Esto no quiere decir que esos valores estén a

cero, simplemente significa que los algoritmos no los tienen en cuenta. La numeración de las neuronas comienza por cero. La matriz de conectividad puede obtenerse mediante el método `getConnectivity()`. También pueden usarse los métodos `getConnectivity(int from, int to)` y `setConnectivity(int from, int to, int con)` para consultar y establecer los arcos del grafo[1]. El número de neuronas de la red puede obtenerse mediante `getNeurons()`.

De acuerdo a la numeración empleada, para indicar cuáles son las neuronas de entrada basta con dar el número de neuronas de entradas. Este número es almacenado en el atributo `in_neurons` y accesible mediante `getInputNeurons()`. Si el número de neuronas de entrada es k, las neuronas de entrada son $0, 1, \ldots, k-1$.

De igual forma, tan sólo hay que indicar el número de neuronas de salida para saber cuáles son estas neuronas. Este valor se encuentra en el atributo `out_neurons` accesible mediante `getOutputNeurons()`. Si el número de neuronas de salida es k, estas neuronas son $n-k, n-k+1, \ldots, n-1$, siendo n el número de neuronas.

La matriz de pesos sinápticos está representada mediante una matriz de reales y se encuentra almacenada en el atributo `weights` accesible mediante `getWeights()`. Además de este, existe un par de métodos para obtener y modificar los pesos individualmente; son `getWeight(int from, int to)` y `setWeight(int from, int to, double weight)`. El valor de `weights[i][j]` es el peso sináptico del arco que va de la neurona i a la j si tal arco existe. Los valores `weights[i][j]` para los que `con[i][j]`=0 son ignorados por el software.

El vector de umbrales se representa mediante un vector de reales almacenado en el atributo `thresholds` accesible mediante `getThresholds()`. Al igual que antes, existen dos métodos alternativos para acceder a los umbrales: `getThreshold(int neuron)` y `setThreshold(int neuron, double threshold)`. Los umbrales de las neuronas de entrada son ignorados.

[1]Las primeras versiones de la clase **FFNN** no poseían estos dos últimos métodos. Al usar redes como el perceptrón multicapa, que deja muchos huecos en las matrices de conectividad y de pesos, se pensó que sería adecuado crear subclases específicas de tales redes que almacenaran la información usando menos memoria. Para ello se hizo necesaria la implementación de métodos que permitieran trabajar con elementos de forma individual y no con grandes matrices de elementos, muchos de los cuales no tendrían sentido para una red particular. Por eso se añadieron los pares de métodos `get-set`, que se aconsejan en lugar de los métodos que devuelven matrices y vectores de elementos.

Las funciones de activación de las neuronas son representadas mediante un vector de objetos **ActivatingFunction**. Este vector es accesible mediante `getActivatingFunction()`. De igual forma que antes, existen los métodos `getActivatingFunction(int neuron)` y `setActivatingFunction(int neuron, ActivatingFunction f)`. **ActivatingFunction** es un interfaz que declara dos métodos: `value(double x)` y `derivative(double x)`. El primero devuelve el valor de la función de activación en x y el segundo devuelve su derivada. Estos valores son necesarios para el cálculo de la salida de la red y para su entrenamiento respectivamente. El uso de estos objetos permite añadir nuevas funciones de activación al software sin realizar cambios en lo que ya está codificado. Las funciones de activación de las neuronas de entrada son ignoradas.

Las funciones de activación implementadas han sido la sigmoide o logística, la tangente hiperbólica y la lineal, cuyas expresiones se muestran a continuación.

$$logist(x) = \frac{1}{1 + e^{-x}} \tag{A.1}$$

$$tanh(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{A.2}$$

$$lineal(x) = ax + b \tag{A.3}$$

Además se ha implementado el operador de composición de funciones para construir funciones de activación más complejas a partir de las anteriores. Todas estas funciones se encuentran junto con el interfaz **ActivatingFunction** en el paquete ***pfc.networks***. Pero hay en ese paquete una clase relacionada con estas funciones que tiene especial interés. Se trata de **ActivatingFunctionFactory**. Esta clase tiene dos métodos públicos de clase que permiten crear funciones de activación indicándole como argumento simplemente su nombre y los posibles parámetros de la misma. El nombre de una función de activación no es el nombre de su clase. Cuando desarrollamos una función de activación le asignamos un nombre y la llamaremos por ese nombre en los ficheros de configuración de las redes. La relación entre los nombres de las funciones y las clases que implementan dichas funciones se encuentra plasmada en un fichero denominado *act-funct* situado en el paquete ***pfc.networks***. Este fichero está formado por una serie de pares `<nombre>=<clase>`.

Cuando se invoca el método `createActivatingFcuntion(String str)` de la clase **ActivatingFunctionFactory**, se analiza la cadena `str` para separar el nombre de la función de los parámetros. Luego, se busca en *act-funct* el nombre de la función para averiguar la clase donde se implementa. Y por último, si dicha clase tiene un constructor que admita un **String** con los parámetros, se invoca dicho constructor para crear el objeto solicitado. Este esquema permite ampliar el número de funciones de activación sin alterar el código existente y sin emplear extraños nombres de clases con nombres de paquetes incluidos para designar a una de esas funciones. Este patrón de diseño, el patrón fábrica, es empleado en otras partes del software. Incluso se implementó una clase auxiliar para facilitar el trabajo y no repetir código.

Hemos indicado hasta aquí la forma en que se guarda la información de la red en los objetos de la clase **FFNN** y cómo pueden consultarse y modificarse. En las pruebas hemos usado un perceptrón multicapa, por eso hemos añadido a la clase un método estático que crea un objeto **FFNN** representando a un perceptrón multicapa. Este método es `createMultilayerNN(int [] layers, ActivatingFunction [] f_act)`. El primer argumento le indica cuantas neuronas por capa hay (además de indicarle el número de capas) y el segundo qué función de activación hay que emplear en cada capa (la función de activación de la capa de entrada es ignorada). Este método se encarga de crear la matriz de conectividad y rellenar los atributos de la clase **FFNN** de la forma adecuada. Así no tenemos que trabajar con la gran cantidad de argumentos de los constructores de la clase.

Por último, queda mencionar cómo se usa la red para obtener una salida a partir de una entrada. El método clave es `evaluate(double []input)`. Recibe un vector de reales como entrada y devuelve un vector de reales a la salida que se corresponde con la salida de la red. Una vez llamado este método los métodos `getOut()` y `getSum()` devuelven en un vector de reales todas las salidas y potenciales sinápticos respectivamente de las neuronas para la entrada evaluada. Esta información es de especial utilidad para los algoritmos de entrenamiento.

No obstante, la red presenta métodos de más alto nivel para evaluarla. Tal es el caso de `getRMSE(Pattern [] patterns)`, `getSEP(Pattern [] patterns)`, `getErrorPercentage(Pattern [] patterns)` o `getErrorPercentage(Pattern [] patterns,`

`double threshold)`. Todos ellos toman un conjunto de patrones y obtienen distintas medidas que indican cómo de bien se comporta la red ante dichos patrones. Así, el primer método devuelve la raíz del error cuadrático medio (Root Medium Square Error[2]), el segundo devuelve el porcentaje de error cuadrático (Squared Error Percentage[3]) y el tercero y el cuarto devuelven el porcentaje de patrones mal clasificados teniendo en cuenta la técnica de "el ganador se lleva todo" y la técnica de los umbrales para interpretar la salida, respectivamente. Hay algunos métodos más de este tipo.

A.2 Patrones

Ahora pasamos a estudiar la representaciń de los patrones. Un patrón está formado por un vector de entrada y un vector de salida deseada. Ambos son vectores de reales. La clase **Pattern** es la encargada de representar a los patrones. Contiene dos atributos que son los mencionados vectores de reales accesibles mediante los métodos `getInput()` y `getOutput()`.

La clase abstracta **PatternsReader** del paquete ***pfc.networks*** representa a los lectores de patrones. Las descendientes de esta clase deben implementar el método `read(Reader r)` que en **PatternsReader** se declara como abstracto. Este método devuelve un vector de patrones (**Pattern**) leídos del **Reader** que se pasa como parámetro. El método `read(String filename)` se encarga de abrir el fichero cuyo nombre se pasa como parámetro y lo lee gracias a la ayuda del método abstracto anterior. Para cada formato distinto, habrá que implementar una subclase distinta. En nuestro caso, hemos usado los ficheros preprocesados de PROBEN1 que tienen todos el mismo formato: una línea por cada patrón, primero aparece el vector de entrada y luego el de salida. Lo único que no puede deducirse de este formato es el número de elementos de cada vector, aparecen los componentes de los dos vectores juntos sin ningún tipo de separación. La clase que interpreta este formato es **PRProben1** que se encuentra en el paquete ***pfc.pr***. A través de método `setParameters(Properties pro)` se le indica cuántos elementos tienen

[2] $RMSE = \sqrt{\frac{\sum_{p=1}^{P}\sum_{i=1}^{S}(t_i^p - o_i^p)^2}{P \cdot S}}$ donde P es el número de patrones y S el número de neuronas de salida

[3] $SEP = 100 \cdot \frac{o_{max} - o_{min}}{P \cdot S}\sum_{p=1}^{P}\sum_{i=1}^{S}(t_i^p - o_i^p)^2$ donde o_{max} y o_{min} son los valores máximo y mínimo que pueden arrojar las neuronas de salida.

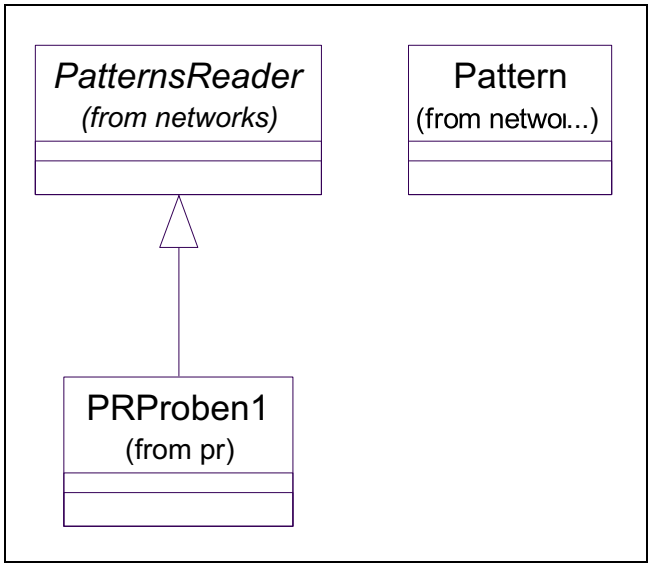

Figura A.3: Clases para representar los patrones.

los vectores de entrada y de salida de los patrones. Este método aparece ya definido en **PatternsReader**, pero allí su cuerpo está vacío.

Nos detenemos un momento aquí para discutir el uso de los objetos **Properties** para tareas de configuración, ya que es una constante en todo el software implementado. En lugar de desarrollar un software y un formato propios para la lectura de ficheros de configuración, hemos decidido echar mano de la jerarquía de clases que java proporciona en su *Standard Edition* y así ahorrar tiempo. Los objetos **Properties** representan conjuntos de pares `<clave>-<valor>`. Tanto la clave como el valor han de ser cadenas de caracteres. Además existen métodos para escribir dichos pares en cualquier **OutputStream** siguiendo el formato `<clave>=<valor>` y leerlos de un **InputStream**. Esto nos permite sin mucho esfuerzo crear ficheros de configuración legibles que serán leídos también sin ningún esfuerzo. Por eso nos decantamos por el uso de esta clase para representar configuraciones. Las claves de un objeto **Properties** son llamadas propiedades. Para configurar los objetos que representan las distintas partes del sistema se le pasa mediante algún método un objeto **Properties** con la configuración.

Las propiedades que espera **PRProben1** que se definan son `inputs` y `outputs`. La primera es el número de elementos del vector de entrada y la segunda el del vector de

salida. Ambos expresados mediante números naturales.

A.3 Algoritmos de entrenamiento

Los algoritmo de entrenamiento son representados mediante objetos de clases que implementan la interfaz **TrainingAlgorithm** del paquete ***pfc.algorithms***. Hay tres elementos que deben ser proporcionados externamente a cualquier algoritmo de entrenamiento:

- La red neuronal a entrenar. Esto se hace mediante el método `setNet(FFNN net)`.
- Los patrones de entrenamiento. Esto se hace mediante `setPatterns(Pattern [] patterns)`.
- El criterio de parada. Esto se hace mediante `setTerminatingCondition(TerminatingCondition tc)`.

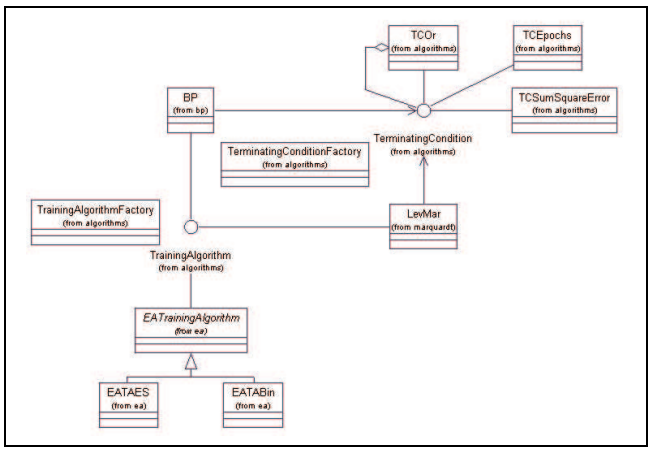

Figura A.4: Clases para representar los algoritmos.

El último método interesante de **TrainingAlgorithm** es `train()` que realiza el entrenamiento de la red. Los algoritmos de entrenamiento siguen el patrón fábrica. Así pues, existe una clase denominada **TrainingAlgorithmFactory** que implementa dos métodos de clase para crear algoritmos de entrenamiento. Lo pararámetros de estos algoritmos

hay que dárselos en un objeto **Hashtable**[4]. Si para crear un algoritmo de entrenamiento se emplea el método que no requiere el nombre del algoritmo como argumento, éste debe encontrarse en la propiedad `name`. Cada algoritmo de entrenamiento reconoce distintos parámetros que serán mencionados cuando se detalle cada uno de ellos. El archivo que contiene la relación entre los nombres de los algoritmos y las clases que los implementan se denomina *train-alg* y se encuentra en el paquete ***pfc.algorithms***.

El algoritmo de Retropropagación, llamado *bp*, se encuentra implementado en la clase **BP** del paquete ***pfc.algorithms.bp***. De entre sus propiedades, las más interesantes son `learning-rate` y `momentum`. La primera establece la tasa de aprendizaje η del algoritmo mientras que la segunda establece la constante de momentos α (véanse las Ecuaciones 3.11 y 3.12).

El algoritmo Levenberg-Marquardt, llamado *lev-mar*, se encuentra implementado en la clase **LevMar** del paquete ***pfc.algorithms.marquardt***. Sus propiedades más interesantes son `learning-rate`, `mu`, `beta` y `mu-max`. La primera es el valor de α en la Ecuación 3.23, la segunda es el valor de μ en dicha ecuación, la tercera es el factor de incremento-decremento de μ y la cuarta es el valor máximo de μ permitido.

El entrenamiento mediante algoritmos evolutivos se lleva a cabo mediante el uso del software que se presenta en el Apéndice B. Para entrenar las redes podemos elegir entre varios algoritmos evolutivos y podemos usar distintos tipos de individuos. Así por ejemplo, para entrenarlas con Algoritmos Genéticos de codificación binaria habrá que realizar una conversión entre los valores reales de los pesos y umbrales de la red y la cadena binaria que los representa pero si usamos Estrategias Evolutivas no habrá que realizar dicha conversión. En ambos casos habrá que decidir en qué orden se colocan los pesos y los umbrales. Luego, una vez decidido el algoritmo a usar y la codificación tenemos que ajustar los parámetros de los operadores del algoritmo. En el paquete ***pfc.algorithms.ea*** se encuentran las clases que se encargan de entrenar la red empleando un algoritmo evolutivo. Estas clases sirven de nexo entre el software de ANNs y el software de EAs y descienden de **EATrainingAlgorithm** que también está en ***pfc.algorithms.ea*** y que ofrece una funcionalidad básica. Cada una de estas clases recibe un nombre igual que el resto de algoritmos y pueden ser tratados como tales a efectos de configuración

[4]En esta ocasión no se ha usado la clase **Properties** debido a que existen parámetros que no son cadenas de caracteres. Tal es el caso de la propiedad `net` que tiene como valor una red neuronal

del algoritmo de entrenamiento. Por cada codificación diferente que se quiera usar debe implementarse una subclase de **EATrainingAlgorithm**. Existe una serie de propiedades que son comunes a todas las subclases y que son interpretadas por la clase anterior. De entre ellas la más relevante es `config` que indica el fichero de configuración del algoritmo evolutivo, donde se detallan los parámetros del mismo.

La clase **EATABin**, conocida como *eatabin* entre los algoritmos, usa los individuos *GABinNN* de los EAs para representar a las redes. Estos individuos usan codificación binaria para almacenar los valores de los pesos y umbrales. Para realizar la codificación, *eatabin* emplea un número fijo de bits por cada valor real. Al valor entero que representan esos bits se les aplica una transformación lineal para que el valor resultante se encuentre dentro de un intervalo. Las propiedades de esta clase son: `bits-per-real`, `maxvalue` y `minvalue` con nombres autoexplicativos.

La clase **EATAES**, cuyo nombre en el software es *eataes* usa los individuos *GAESNN* de los EAs para representar a las redes. Estos individuos representan los pesos mediante un vector de pesos. Además de ese vector almacena un vector de desviaciones típicas y una matriz de ángulos para ser usadas en las Estrategias Evolutivas.

Por último, vamos a dedicar unas líneas a las condiciones de parada. Las clases relacionadas con las condiciones de parada se encuentran en el paquete ***pfc.algorithms***. Las condiciones de parada siguen también el patrón de fábrica siendo **TerminatingConditionFactory** la clase encargada de crear los objetos y *term-cond* el fichero con la relación entre los nombres de la condiciones y las clases que las implementan. Las condiciones de terminación implementadas son: *epochs*, *sum_square_error* y *or*. La primera se detiene cuando se llega a un determinado número de etapas, la segunda cuando la suma del error cuadrático se encuentra por debajo del valor indicado, y la tercera cuando alguna de sus condiciones componentes se cumple. Los parámetros de las condiciones se encuentran junto a los nombres de las condiciones en los ficheros de configuración. Al construir un objeto de estas clases reciben una cadena de caracteres que es analizada para obtener los parámetros.

A.4 Realización de las pruebas

En esta sección vamos a presentar la parte del software que se dedica a los métodos de evaluación de la red. La clase de la que deben heredar todos los métodos de evaluación es **NNProccess**. Esta clase sirve a la vez como fábrica para crear los métodos de pruebas. La relación entre las clases y los nombres de los métodos se encuentran en el archivo *nnproccess* del paquete ***pfc.proccess***. Para crear uno de estos métodos de evaluación tenemos que indicar el conjunto de propiedades para configurarlo, la red neuronal, el conjunto de patrones, el algoritmo de entrenamiento (ya configurado) y un **OutputStream** donde se arrojarán los resultados obtenidos. Una vez creado el objeto, se pone en marcha la evaluación mediante el método `execute()`. Nosotros hemos implementado dos métodos distintos: *TrainingTest* y *CrossValidation*.

El primero divide el conjunto de patrones en dos grupos de patrones y usa uno de ellos para entrenar la red y el otro para testearla. Las propiedades de este método son:

- `trainfrac`: indica qué fracción de la cantidad total de patrones son usados para el entrenamiento. Se toman los primeros patrones del conjunto para el entrenamiento.
- `class-method`: indica el criterio que se usa para interpretar la salida de la red como clase. Si toma valor *winner-takes-all* se usa la técnica de "el ganador se lleva todo" y si toma valor *threshold* se emplea la técnica del umbral.
- `threshold`: indica el umbral que hay que usar cuando se elige la técnica del umbral en la interpretación de la salida de la red.

El *CrossValidation* divide el conjunto de patrones en k grupos y realiza k pruebas. En cada una de ellas entrena la red con $k-1$ grupos y testea con el que queda. Esto lo hace eligiendo cada vez un grupo distinto para testear. Las propiedades de este método son:

- `folds`: indica el número de grupos en que se divide el conjunto de patrones.
- `class-method`: igual que en *TrainingTest*.
- `threshold`: igual que en *TrainingTest*.

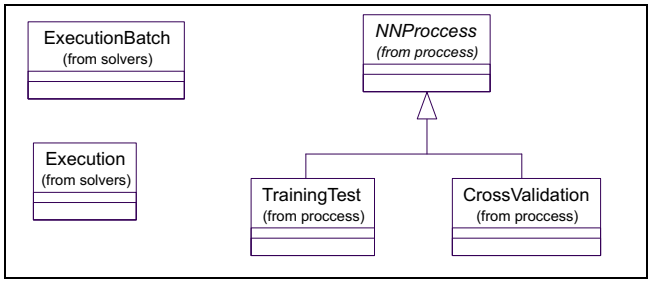

Figura A.5: Clases para representar los métodos de evaluación.

Una de las primeras cosas que hacen estas clases es inicializar los pesos de la red. Para ello hacen uso de una clase del paquete ***pfc.networks.ffnn*** llamada **WeightInitialization** que inicializa los pesos de forma "inteligente" tal y como se describió en el Apartado 4.1.3.

Por último, para poner en funcionamiento las pruebas se pueden usar una serie de clases que se encuentran en el paquete ***pfc.solvers*** y que se encargan de leer los ficheros de configuración que reciben como parámetro en la línea de órdenes y preparan la red, los patrones, el algoritmo y el método de evaluación para posteriormente iniciar tales pruebas. La clase usada para nuestras pruebas ha sido **Execution** la cual, además de lo anterior, guarda los resultados en un fichero de texto legible y almacena la red resultante en un fichero binario. Otra clase de utilidad es **ExecutionBatch** que es capaz de realizar varias pruebas de forma consecutiva. El resto de las clases fueron usadas en las primeras fases del proyecto y presentan una interfaz más antigua y menor flexibilidad.

Apéndice B

Software de Algoritmos Evolutivos

Este capítulo se dedica a presentar el software implementado para trabajar con algoritmos evolutivos. El diseño se ha hecho teniendo siempre en mente el desarrollo de un software flexible y fácilmente ampliable que sea capaz de abordar cualquier algoritmo evolutivo [AT01]. Para resolver un problema usando estos algoritmos necesitamos una serie de componentes que nosotros vamos a representar mediante clases. Los componentes son los siguientes:

- El **problema**. Necesitamos evaluar las soluciones que encuentra el algoritmo para conocer su calidad. Esto se hace a través de la función de evaluación o función de aptitud (*fitness*) que depende del problema que se desea resolver.
- Los **individuos**. Las posibles soluciones del problema son representadas mediante individuos. No existe un único modo de representar las soluciones de un problema en forma de individuos. No todo operador se puede aplicar a todo tipo de individuo.
- La **población**. Está formada por un conjunto de individuos y propiedades estadísticas.
- Los **operadores**. Son los encargados de trabajar con los individuos para dar lugar sucesivamente a las nuevas generaciones.
- La **condición de parada**. Un algoritmo evolutivo sigue un proceso iterativo que debe parar en algún momento. El algoritmo se detendrá cuando cierta condición se cumpla.

- El **algoritmo**. El problema, los individuos, los operadores y la condición de parada deben colaborar de alguna forma para resolver el problema. La forma de colaborar la decide el algoritmo.

En la Sección 3.3 vimos el pseudocódigo de los algoritmos evolutivos secuenciales y paralelos. Estos algoritmos están formados por un bucle gobernado por la condición de parada y una serie de operadores que se aplican a los individuos de la población. En cada generación se aplican los mismos operadores. En realidad, lo que diferencia un algoritmo como la Estrategia Evolutiva de otro como el Algoritmo Genético son los operadores y los individuos empleados. Así pues, para hacer flexible nuestro software nos hemos asegurado de que es fácil cambiar ambos componentes mediante archivos de configuración y hemos implementado una única clase para representar a los algoritmos evolutivos. De esta forma, mediante la elección adecuada de individuos, operadores y condición de parada podemos obtener cualquier algoritmo evolutivo.

Los problemas resueltos en el proyecto se encuentran en el paquete ***pfc.ea***, los individuos usados en ***pfc.ea.individuals***, los operadores en ***pfc.ea.operators***, las condiciones de parada en ***pfc.ea.stopcond*** y el algoritmo en ***pfc.ea.algorithms***. A continuación veremos con más detalle cada uno de estos componentes.

B.1 El problema

Los problemas a resolver son representados por subclases de **Problem**. Dicha subclase deberá implementar los métodos `setParameters(Properties pro)`, `getParameters()` y `evaluate(Object o)`. El primero configura el objeto que representa el problema, el segundo obtiene la configuración del problema y el tercero se usa para evaluar las soluciones.

El método `evaluate(Object o)` es el que implementa la función de evaluación del problema. No olvidemos que esa función es lo único que necesita conocer un algoritmo evolutivo del problema para realizar su trabajo. Ese método devuelve un valor real que es la aptitud de la solución que se le pasa como parámetro. En el software implementado se asume, sin pérdidad de generalidad, que este valor es positivo y que una solución es mejor cuando tiene mayor aptitud (maximización). Obsérvese que el parámetro del método es un objeto **Object** y no un individuo. Esto se debe a que un individuo en general contiene

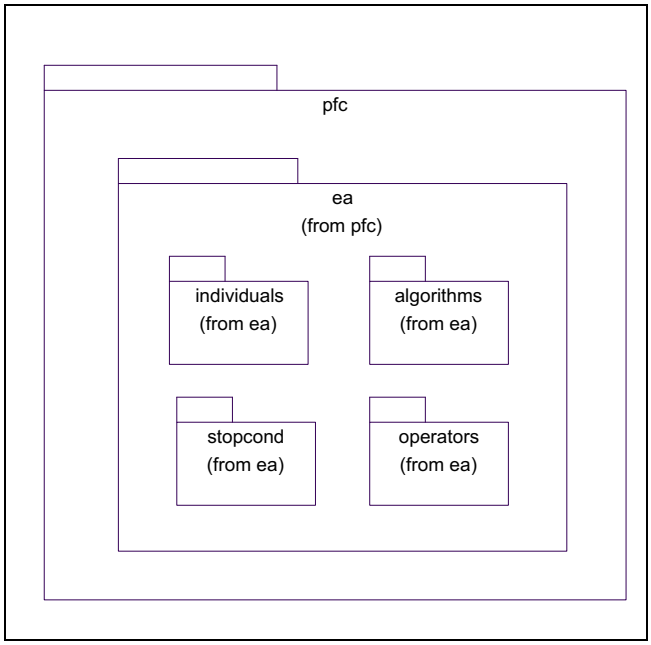

Figura B.1: Un esquema de los paquetes del software de algoritmos evolutivos.

más información que la solución al problema. Por ejemplo, todos los individuos tienen además la aptitud y los individuos usados en las estrategias evolutivas poseen parámetros de autoadaptación usados por los operadores. El objeto que se le da como argumento al problema para que lo evalúe tan sólo contiene la solución codificada de acuerdo al genotipo escogido. Este objeto puede ser de lo más diverso, desde un vector de bytes en el caso de usar cadenas binarias como individuos hasta cualquier estructura creada específicamente para un problema.

Por último, el método `represent(Object o)` toma una solución y devuelve una cadena que representa dicha solución de forma legible. Este método es usado cuando hay que mostrar la solución al usuario.

B.2 Individuos y Población

Los distintos tipos de individuos son representados por subclases de **Individual**. Los individuos siguen el patrón fábrica, lo cual significa que existe un nombre para cada tipo de individuo implementado. La relación entre estos nombres y las clases que los implementan

se encuentran en el archivo *individual* del paquete ***pfc.ea.individuals***. La propia clase **Individual** actúa como clase fábrica. Algunos métodos de **Individual** son abstractos y deben ser implementados por las subclases. Otros son definidos en la propia clase. Los métodos abstractos son:

- `evaluableObject()`: Devuelve el objeto contenido en el individuo que representa la solución y que es tomado por el problema para ser evaluado o representado.
- `initialize()`: Inicializa el individuo.
- `copyFrom(Individual ind)`: Modifica el individuo para que sea una copia exacta del individuo que se le pasa como parámetro.

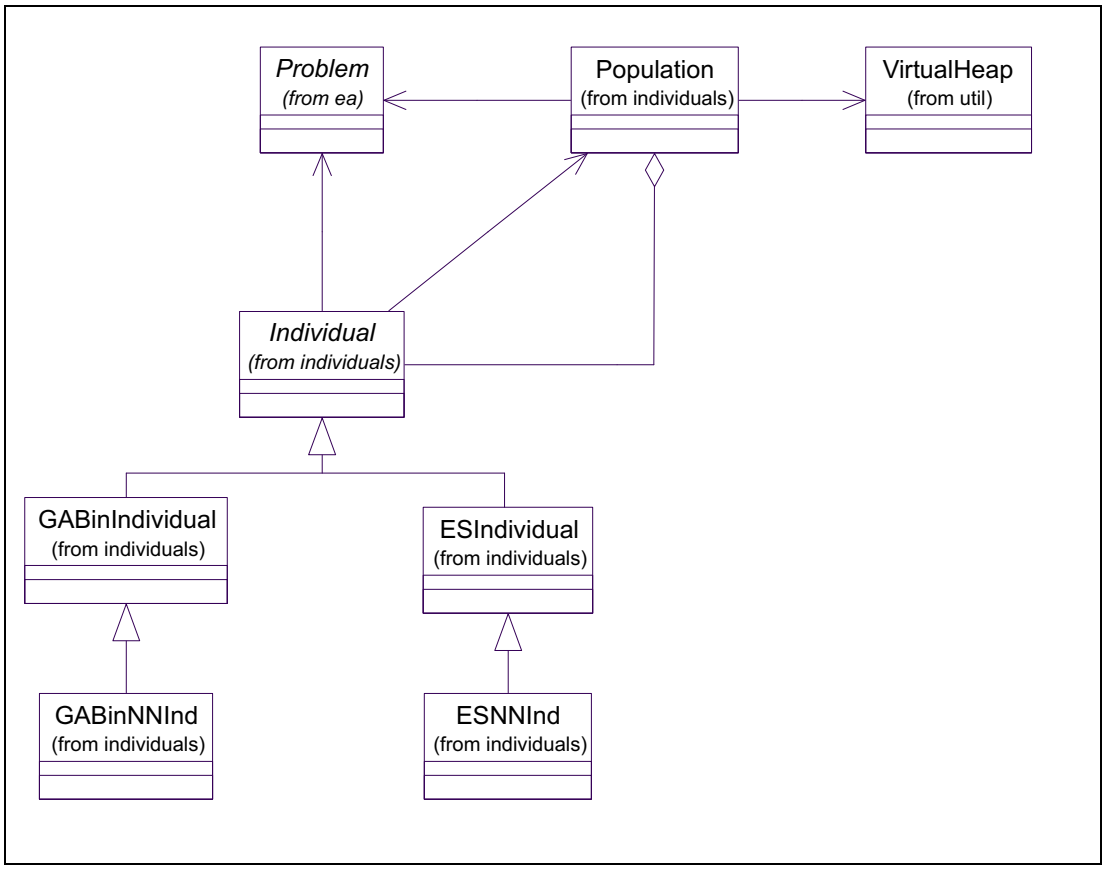

Figura B.2: Clases para representar los individuos y la población.

La aptitud del individuo es almacenada en el atributo `fitness` y puede consultarse con `getFitness()`. Todos los individuos tienen una referencia a la población y al problema. La referencia al problema es usada por los métodos `evaluate()` y `represent()`. El primero evalúa el individuo y para ello invoca al método `evaluate(Object o)` del problema pasándole como argumento el objeto obtenido por `evaluableObject()`. El segundo devuelve una cadena de caracteres representando la solución, delegando su trabajo al método `represent(Object o)` del problema.

Existen dos constantes públicas llamadas `bestFirst` y `worstFirst` que son objetos que implementan la interfaz **Comparator**. La primera de ellas establece que un individuo es menor que otro cuando su aptitud es más alta y la segunda a la inversa. Esto permite ordenar los individuos por orden decreciente y creciente de calidad, respectivamente. Además, la propia clase **Individual** implementa la interfaz **Comparable** y lo hace considerando que un individuo es menor que otro cuando su aptitud es más baja.

Los tipos de individuos implementados son: *GABin*, *GABinNN*, *GAInt*, *ES*, *ESNN* y *VRP*. Los dos primeros son cadenas binarias y presentan una propiedad que determina su longitud: `length`. La diferencia entre ambos es que el segundo se usa en el entrenamiento de redes neuronales y por ello la inicialización es distinta para tener en cuenta las consideraciones sobre inicialización de pesos mencionadas en el Apartado 4.1.3. La clase **GABinNNIndividual** que representa a los individuos *GABinNN* es subclase de **GABinIndividual** que es la que representa a *GABin*. El tercer tipo de individuo es un vector de enteros cuya longitud se establece mediante la propiedad `length`. Los individuos *ES* y *ESNN* se usan en las Estrategias Evolutivas. Poseen dos propiedades: `length` y `angles`. La primera establece el tamaño del vector de variables y la segunda indica si las variables se consideran correladas o no. Al igual que ocurría con *GABinNN*, los individuos *ESNN* son usados en el entrenamiento de redes neuronales. Por últmo, los individuos *VRP* representan soluciones del problema VRP. La propiedad `customers` indica el número de clientes del problema.

Para ahorrar memoria se reutilizan los objetos que representan a los individuos. Esto se hace con ayuda de una clase auxiliar llamada **VirtualHeap**. El objeto **VirtualHeap** del sistema guarda los individuos que no se usan en un momento dado. Cuando es necesario emplear nuevos individuos, son solicitados a este objeto. Si tiene alguno disponible

lo devuelve, en otro caso debe crearlo. Para esto último necesita información sobre el constructor o cualquier otro método que permita crear el individuo. El objeto que representa la población (objeto de la clase **Population**) es el encargado de controlar este objeto **VirtualHeap** y ofrece entre sus métods algunos para crear y "eliminar" individuos[1].

La clase **Population** posee un método para inicializar la población: `initialize()`. Este método recorre la población invocando el método del mismo nombre de los individuos. El resto de los métodos que ofrece esta clase (sin contar los de creación y eliminación de individuos) son de consulta. Los métodos de consulta permiten conocer la aptitud media de la población, la aptitud del mejor individuo, la aptitud del peor, el tamaño de la población, el mejor individuo, el peor, el número de individuos creados, el número de individuos en el heap, etc. Al crearse la población, se toman los individuos del heap. Cada vez que un operador necesita individuos auxiliares debe tomarlos del heap via el método `newIndividual()` de la clase **Population**. De igual forma, cuando hay individuos que no son necesarios deben devolverse al heap por medio de alguno de los métodos `deleteIndividual(Individual ind)` o `deleteIndividuals(Individual [] ind)` de la misma clase. Sin embargo, el heap no es la única fuente de individuos del sistema. Cuando se emplea un algoritmo evolutivo paralelo, los individuos pueden haber llegado desde otro subalgoritmo. En tal caso hay que establecer las referencias del individuo al objeto población y problema del subalgoritmo al que llega. A este proceso lo hemos denominado *registro* del individuo y de ello se encarga el método `registerIndividual(Individual ind)`. Todos los individuos creados y procedentes de la red deben ser registrados.

B.3 Operadores

La cantidad de operadores en el campo de la computación evolutiva es enorme. Para poder emplear el patrón de fábrica con los operadores debemos diseñar una clase que declare la funcionalidad básica de todos los operadores o, al menos, de la mayoría. Esta clase es **EAOperator** que se encuentra en el paquete ***pfc.operators***. Hay operadores que actúan sobre un individuo (como la mutación), otros actúan sobre dos (como el cruza-

[1] Al decir eliminar nos referimos a que el objeto pasa a disposición del **VirtualHeap** para ser reutilizado posteriormente si se solicita un nuevo individuo

miento de dos puntos), otros sobre la población (como la selección), etc. El único nexo que hemos encontrado entre todos ellos es que toman individuos, los transforman y los devuelven transformados. Esos individuos los pueden tomar de la población o de la salida de otro operador. Por esto, el método que realiza el trabajo del operador, acepta un array de individuos y devuelve otro array de individuos: `operation(Individual [] ind)`. El acceso a la población se hace mediante una referencia que mantienen los operadores a la misma y que se establece usando el método `setPopulation(Population pop)`. Para configurar el operador se usa `setParameters(Properties p)`. La clase **EAOperator** actúa también como clase fábrica de los operadores. La relación entre los nombres y las clases de los operadores se encuentra en el fichero *eaoperator* del paquete ***pfc.operators***. Hemos implementado una gran cantidad de operadores. Algunos son específicos de un problema e incluso de un tipo de individuo, otros en cambio, actúan sobre cualquier operador (como es el caso de los operadores de selección). Para no extender esta sección demasiado y poder profundizar en algunos detalles interesantes del funcionamiento de los operadores les hemos dedicado el Apéndice C.

B.4 Algoritmo

Como mencionamos anteriormente, existe una única clase que representa a los algoritmos evolutivos, se trata de **EvolutionaryAlgorithm** en el paquete ***pfc.ea.algorithms***. El constructor de esta clase toma: un objeto **Problem** que representa el problema a resolver, un objeto **Population** con la población, un array de operadores con los operadores que hay que aplicar en cada paso del algoritmo, un **OutputStream** donde el algoritmo vuelca la información resultante, un **StopCondition** con la condición de parada y, por último, un objeto **MessageManager**. Todos estos objetos deben estar ya configurados. Una vez creado el objeto **EvolutionaryAlgorithm** se pone en funcionamiento el algoritmo usando el método `execute()`. Podemos consultar la población, el número de pasos ejecutados y el gestor de mensajes por medio de los métodos `getPopulation()`, `getStep()` y `getMessageManager()`, respectivamente.

En cada paso, el algoritmo parte de un array nulo de individuos y aplica los operadores secuencialmente, esto es, al vector nulo le aplica el primero, la salida de ese operador es la entrada del siguiente y así procede hasta acabar con todos los operadores. La salida

del último operador es entregada al heap. Este código podemos verlo en la Figura B.3. El procedimiento anterior se repite hasta que se cumple la condición de parada.

```
private void goOneStep() throws Exception
{

    Individual [] ind;
    int i;

    ind = null;

    for (i=0; i < ea_ops.length; i++)
    {
        ind = ea_ops[i].operation(ind);
    }

    pop.deleteIndividuals (ind);

}
```

Figura B.3: Código de un paso del algoritmo.

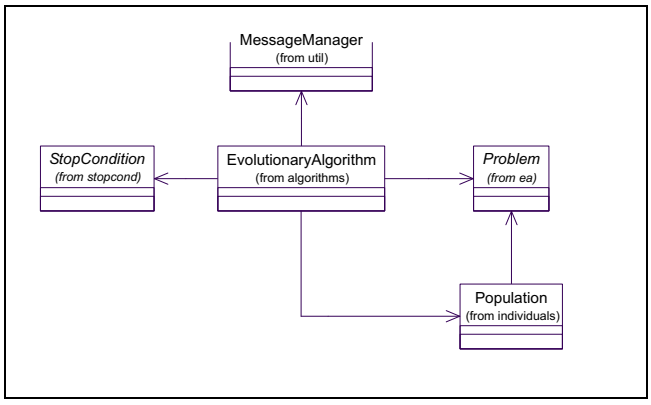

Figura B.4: Clases para representar el algoritmo evolutivo.

El objeto **MessageManager** se usa en los algoritmos evolutivos paralelos. En estos algoritmos existe un proceso que se encarga de sincronizar al resto. Este proceso está implementado en la clase **DistributionManager** del paquete ***pfc.util*** y cuando se inicia

se pone a la escucha de conexiones. Los subalgoritmos del algoritmo paralelo emplean la clase **MessageManager** como interfaz con este proceso central. El objeto **MessageManager** que recibe el algoritmo evolutivo debe haber sido configurado adecuadamente y puesto en contacto con el proceso central previamente. A partir de entonces el intercambio de mensajes entre los subalgoritmos y el **DistributionManager** es como sigue:

- Antes de comenzar la ejecución. El algoritmo solicita el inicio de la misma y se bloquea hasta que recibe el permiso del proceso central.
- Durante el transcurso del algoritmo el proceso central puede mandar mensajes informativos al algoritmo. Estos mensajes son almacenados en el **MessageManager** y pueden ser consultados por cualquiera de los componentes del algoritmo evolutivo.
- Cuando el algoritmo acaba envía al proceso central la notificación de que ha acabado así como la razón de su terminación y un objeto de clase **Results** con el mejor individuo de la población y el número de pasos realizados. Después de esto se corta la comunicación.

Los mensajes que se intercambian son objetos de la clase **SpecialPacket** del paquete ***pfc.util***. Esta clase declara tres atributos: `type`, `obj` y `stop_reason`. El primero es un entero que indica el tipo de mensaje y que debe tomar alguno de los valores constantes declarados en la propia clase. Los tipos de mensaje son:

- `DUMMY`: Mensaje ignorable, no contiene nada útil.
- `CLOSE_SOCKET`: Indica que debe cerrase la conexión.
- `START_REQUEST`: Con este mensaje el algoritmo indica al proceso central que está listo para comenzar la ejecución y a la espera del permiso.
- `START_CONFIRMATION`: Con este mensaje el proceso central indica a los algoritmos que deben comenzar la ejecución.
- `STOP_INDICATION`: Cuando un algoritmo acaba envía este mensaje al proceso central. El atributo `stop_reason` debe contener el objeto **StopCondition** que ha provocado la parada del algoritmo. El atributo `obj` debe contener un objeto **Results** con información sobre la ejecución (mejor individuo y pasos del subalgoritmo).

- `STOP_REQUEST`: Este mensaje es enviado por el proceso central a todos los algoritmos que sigan activos cuando se recibe una indicación de parada de algún algoritmo. El atributo `stop_reason` del paquete `STOP_INDICATION` es copiado en este paquete.

El proceso central sabe cuantos subalgoritmos forman el algoritmo paralelo. Cuando recibe un mensaje `START_REQUEST` de cada uno de ellos les envía un mensaje `START_CONFIRMATION` para que comiencen la ejecución. De esta forma consigue que comiencen todos aproximadamente a la vez. Cuando un subalgoritmo acaba, éste le envía al proceso central una indicación de tal hecho y el proceso central lo notifica al resto de subalgoritmos. Una vez que todos los subalgoritmos han acabado escribe todos los resultados recibidos de éstos en un fichero y acaba. Por tanto el proceso central tiene tres funciones: sincronizar el comienzo de la ejecución, notificar las defunciones y guardar de forma centralizada los mejores resultados.

La importancia de la notificación de la finalización de los subalgoritmos estriba en que permite de una forma elegante saber si algún subalgoritmo ha encontrado ya la solución óptima. De esta forma se pueden detener el resto de los subalgoritmos puesto que la búsqueda ya ha acabado. Para esto no basta con notificar la defunción de un algoritmo, en los subalgoritmos debe haber "algo" que se encargue de comprobar dichas notificaciones y que sea capaz de detener el subalgoritmo. El único componente de los descritos al comienzo del capítulo que es capaz de detener el subalgoritmo es la condición de parada. Por lo tanto, debe haber una condición de parada que se encargue de revisar los mensajes que llegan al **MessageManager** y que detenga el algoritmo cuando se ha recibido una notificación de parada cuya razón sea la consecución del óptimo.

En el caso de que el algoritmo sea secuencial el objeto **MessageManager** que se le pasa al constructor de **EvolutionaryAlgorithm** es `null`, en tal caso, no existe ningún tipo de intercambio de mensajes. En futuras ampliaciones, podrían añadirse nuevos tipos de mensaje para que se comunicaran otros elementos entre sí. Por ejemplo, se podrían intercambiar parámetros de operadores entre dos subalgoritmos o cualquier otro tipo de información. En la actual implementación la única información que puede llegar a un subalgoritmo durante su ejecución es una notificación de parada.

B.5 Condición de Parada

Las clases que representan condiciones de parada deben ser descendientes de **StopCondition**. Esta clase se encuentra en el paquete ***pfc.ea.stopcond*** junto con sus descendientes. Las condiciones de parada siguen el patrón fábrica siendo la propia clase **StopCondition** la creadora de los objetos y el fichero *stopcondition* del mismo paquete el que establece la relación entre los nombres y las clases de las condiciones de parada. En **StopCondition** se declaran dos métodos abstractos: `stopReason()` y `stop(EvolutionaryAlgorithm ea)`. El primero devuelve una cadena de caracteres que indica de forma legible para un humano por qué motivo es cierta la condición de parada. El segundo recibe como parámetro el algoritmo evolutivo y devuelve un valor de verdad que indica si se cumple la condición de parada o no.

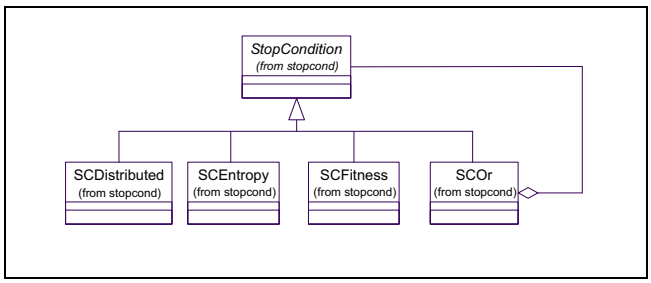

Figura B.5: Clases para representar la condición de parada.

Las condiciones de parada implementadas son:

- *SCSteps*: Se hace cierta cuando se han realizado un número de pasos. Este número se le indica a la condición por medio de su propiedad `steps`.
- *SCFitness*: Se hace cierta cuando la aptitud del mejor individuo de la población supera un valor dado. Este valor se establece por medio de la propiedad `fitness`.
- *SCEntropy*: Se hace cierta cuando la entropía media de la población ha caído por debajo de un valor. Este valor se establece mediante la propiedad `entropy`.
- *SCDistributed*: Se hace cierta cuando se recibe una notificación procedente del proceso central indicando que un subalgoritmo ha acabado encontrando la solución óptima.

- *SCOr*: Se hace cierta cuando se cumple alguna de las condiciones componentes. Las condiciones componentes se establecen usando las propiedades `scnum`, `sc.<i>` y `sc.<i>.parameter.<param>`, donde `i` va entre 0 y `scnum` y `param` es el nombre del parámetro que se establece para el operador `i`.

SCDistributed es la condición de parada encargada de escrutar el **MessageManager** en busca de notificaciones de paradas. Cuando encuentra una la elimina de **MessageManager** y para sólo si la condición por la que paró el subalgoritmo fue *SCFitness*. Esto es algo más general que parar por encontrar el óptimo. Los mensajes de **MessageManager** que no sean defunciones son respetados por *SCDistributed*. Las cuatro primeras condiciones son condiciones simples mientras que la última es una condición compuesta puesto que se forma a partir de otras condiciones. Como mencionamos en la sección anterior, en el mensaje de notificación de parada hay que indicar la razón de la misma usando un objeto **StopCondition**. Si este objeto es un *SCOr* la razón de la parada es una de las componentes de ese objeto. Para conocer la condición simple que provocó una parada existe un método en **StopCondition** que devuelve dicha condición. Este método es `getStopReasonObject()`. Para las condiciones simples este método devuelve una referencia a ellas mismas. Para *SCOr* devuelve el resultado de invocar dicho método en la condición componente que produjo la parada.

B.6 Puesta en marcha

Una vez que tenemos todos lo componentes de un algoritmo evolutivo debemos ponerlo en marcha para que realice su trabajo. Esto lo hace la clase **Driver** del paquete ***pfc.ea***. En su método `main()` lee el fichero de configuración que se le pasa por parámetro y crea el problema, la población, los operadores, el gestor de mensajes, la condición de parada y el algoritmo. Hay operadores a los que puede llevarles un tiempo la inicialización. Ese es el caso por ejemplo, del operador para enviar individuos en una migración. Estos operadores crean una hebra para realizar su inicialización. El método `waitForReady()` de **EAOperator** se bloquea hasta que la inicialización del operador haya concluido. Este método es invocado en cada operador tras crearlos. Una vez que todo está listo, se ejecuta el algoritmo. Cuando concluye, se invocan los métodos `endOperation()` y `waitForEnd()`

de los operadores. El primero de estos métodos informa al operador de que ya no va a ser usado y debe liberar los recursos adquiridos. Este proceso puede no ser inmediato y el operador puede crear una hebra para tal fin. El segundo método se bloquea hasta que la finalización haya concluido[2]. Otra de las funciones de **Driver** es escribir en un fichero de texto legible el resultado de la ejecución.

[2]El motivo por el cual existe un método no bloqueante para indicar que se debe inicializar o finalizar un operador y otro bloqueante para esperar la conclusión de esa inicialización o finalización es evitar interbloqueos. En el Apéndice C se profundizará en ello.

Apéndice C

Operadores

Ante la gran cantidad de operadores implementados, hemos dedicado un apéndice a todos ellos. En las siguientes secciones vamos a dedicar unas líneas a cada uno de ellos. Para poner un poco de orden los hemos clasificado dentro de nueve grupos. Comenzaremos en la siguiente sección con consideraciones generales sobre los operadores y luego veremos los operadores de cada grupo.

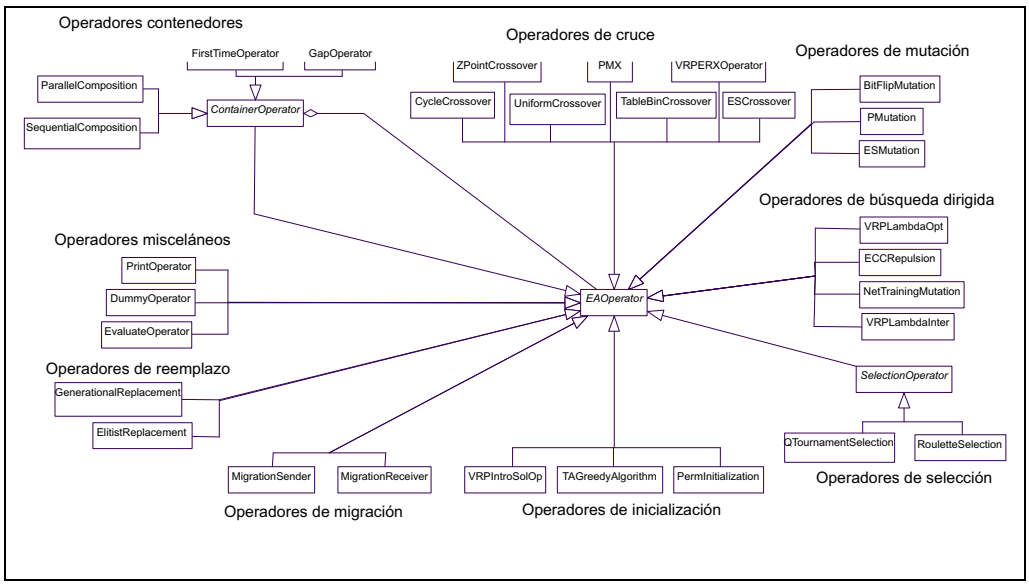

Figura C.1: Clases para representar los operadores.

C.1 Generalidades

Todos los operadores son subclases de la clase **EAOperator**. Esta clase declara los métodos abstractos `operation(Individual [] ind)` y `setParameters(Properties p)`. Estos métodos serán implementados por los operadores. Esta case también permite la consulta de los parámetros de cada operador. Para ello existe un atributo protegido de clase **Properties** llamado `params` que puede ser consultado por medio de `getParams()`. Este atributo debe ser actualizado por cada operador, añadiendo las propiedades específicas de cada uno de ellos. Otro método de importancia es `setPopulation(Population pop)` usado al inicializar el operador para que tenga una referencia a la población. Esta referencia se usa cuando un operador accede o modifica la población, como ocurre por ejemplo con los operadores de selección y reemplazo. El método `toString()` debe ser sobreescrito por cada operador para que devuelva un mensaje que lo identifique. Este mensaje será escrito en los ficheros de resultados para indicar la configuración del algoritmo. Los métodos `waitForReady()`, `endOperation()` y `waitForEnd()` fueron introducidos al crear los operadores de migración y están relacionados con el interbloqueo.

En general, un operador puede necesitar ciertos recursos para su labor, recursos que debe liberar cuando acabe su trabajo. Con "recurso" no nos referimos a la memoria, puesto que Java tiene un recolector de basura. Un recurso posible puede ser una conexión con otro proceso a través de la red por medio de un socket. Cuando el operador es inicializado se establece esa conexión y cuando no es necesaria debe cerrase. Las comunicaciones mediante sockets fueron las causantes de la inclusión de los tres nuevos métodos a la clase **EAOperator**. Pero se pueden usar para cualquier otro tipo de recurso que requiera ser inicializado y finalizado. Se asume que los operadores comienzan la inicialización cuando se invoca `setParameters(Properties p)`. Ese método debe volver sin bloquearse. Si la inicialización requiere alguna acción que puede acabar en un bloqueo, el operador debe crear una hebra para realizar dicha acción. En cambio el método `waitForReady()` debe bloquearse hasta que la inicialización se haya completado. Una alternativa habría sido crear una hebra por cada operador e inicializar un operador en cada hebra, bloqueándose ésta si fuera necesario. Pero de esta forma se crearían hebras innecesarias. A la hora de realizar la finalización se procede del mismo modo. El método `endOperation()` ordena la finalización y no debe bloquearse. Posteriormente el método `waitForEnd()` se

bloquea hasta que la finalización haya acabado. En la implementación actual la mayoría de los operadores no requieren recursos especiales así que no usan estos métodos. Por ese motivo en **EAOperator** se define un comportamiento por defecto para los métodos `waitForReady()`, `endOperation()` y `waitForEnd()` que consiste en no hacer nada.

El método `operation(Individual [] ind)` debe cuidar de que no se pierda ningún individuo. Todos los individuos que recibe deben acabar en el heap o en el array de salida del método. Y lo mismo ocurre con todo individuo que cree o que reciba de fuentes externas. Para crear individuos los operadores deben hacer uso del método `newIndividual()` del objeto **Population**. Si recibe un individuo de una fuente externa debe registrarlo usando `registerIndividual(Individual ind)` del mismo objeto (los individuos creados no hace falta registrarlos porque ya lo hace el método `newIndividual()`). Por otro lado, no se puede devolver en la salida un individuo de la población, puesto que éste puede ser eliminado por el siguiente operador, en su lugar hay que hacer una copia del individuo deseado y devolver la copia.

Habrá operadores que esperen un número concreto de individuos en su entrada, otros actuarán con cualquier número de ellos. Lo ideal es que todos los operadores sean capaces de hacer algo independientemente del número de individuos que reciban. Un caso especial a tener en cuenta es el hecho de que puede que el array de entrada sea `null`. El operador puede tratarlo como equivalente a un vector sin individuos o darle un significado especial. Cuando describamos cada operador mencionaremos el comportamiento que adopta en función del número de individuos recibidos.

C.2 Operadores de Selección

Nuestros operadores de selección permiten elegir individuos de la población y/o del array de individuos de entrada. Puesto que gran parte del código de un operador de selección es común a todos ellos hemos implementado una clase abstracta **SelectionOperator** que extienden los operadores de selección específicos y que implementa el método `operation(Individual [] ind)`. Esta clase declara un método abstracto llamado `selectOne()` que devuelve un individuo. Las subclases deben implementar este método. Para elegir varios individuos, se invoca a este método tantas veces como sea necesario. Existe un atributo protegido llamado `ind` que es el array de individuos desde

donde la subclase debe elegir los individuos. Este array se forma en la clase **SelectionOperator** a partir de los individuos de entrada y/o los individuos de la población. La subclase no puede devolver los individuos de ese array, debe hacer una copia del individuo elegido y devolver dicha copia. La clase padre tomará ese individuo devuelto y lo colocará en el vector de salida. La razón de esta copia es la siguiente. El operador de selección no puede devolver los individuos de la población, debe copiarlos; sin embargo, sí puede devolver los individuos del vector de entrada. Como en el atributo `ind` se pueden encontrar los individuos de ambas fuentes mezclados hay que realizar la copia siempre.

En la clase **SelectionOperator** se definen dos propiedades que son comunes a todos los operadores de selección. Estas son `select` e `include_population`. La primera es un número entero que indica el número de individuos a elegir. La segunda puede tomar valores *yes* y *no* e indica si ha de tenerse en cuenta la población a la hora de elegir individuos. Si no se tiene en cuenta los individuos son elegidos exclusivamente del vector de entrada. Si se tiene en cuenta los individuos son elegidos de la unión del vector de entrada y la población. La única forma de elegir individuos exclusivamente de la población es incluyendo la población en la selección y dándole como entrada un array sin individuos o nulo. Si se recibe un array nulo y no se incluye la población en la elección el operador devuelve un array nulo. En el resto de casos devuelve siempre un array con tantos individuos como indique la propiedad `select` siempre que haya individuos de entre los que elegir. Si no hay individuos donde elegir el comportamiento depende del operador específico.

Los operadores de selección implementados son: el torneo y la ruleta. El torneo está implementado en la clase **QTournamentSelection** y su nombre es *TournamentSelection* mientras que la ruleta está implementada en **RouletteSelection** y su nombre es *RouletteSelection*. Ambos operadores lanzan una excepción cuando no hay individuos donde elegir y la propiedad `select` tiene un valor mayor que cero. La selección por ruleta no añade ninguna propiedad más, pero la selección por torneo presenta una nueva propiedad llamada `q` que establece el valor de q en el torneo. La clase **QTournamentSelection** posee también métodos para consultar y establecer dicho valor.

C.3 Operadores de Recombinación

Lo operadores de recombinación implementados son muy distintos y no existen comportamientos comunes entre ellos, salvo que lanzan una excepción si el array de individuos es nulo. Aparte de eso sólo podemos mencionar algunas carácterísticas que son comunes en los operadores sexuales[1]. Estos operadores toman el array de individuos y los recombina de dos en dos, el primer individuo con el segundo, el tercero con el cuarto, y así sucesivamente. Si recibe con array con un número impar de individuos, el último individuo pasa intacto al array de salida. Si el array no contiene ningún individuo, se devuelve un array también sin individuos. Hay operadores que obtienen dos individuos de una recombinación mientras que otros tan sólo obtienen uno. Los primeros devuelven un array con el mismo tamaño que el de entrada puesto que incluyen los dos individuos fruto de la recombinación. Los segundos devuelven un array con la mitad del tamaño del de entrada si éste es par. Si fuera impar hay que incluir además el último individuo que no tiene pareja. A continuación vamos a presentar los operadores de recombinación implementados.

C.3.1 Recombinación de z Puntos

Este operador se encuentra implementado en la clase **ZPointCrossover** y recibe el nombre de *PointCrossover*. Actúa sólo sobre individuos *GABin* y *GABinNN* (que es un tipo de individuo *GABin*). Sólo tiene una propiedad, que es el número de puntos en que se divide el cromosoma y se llama `z`.

C.3.2 Recombinación Uniforme

Este operador se encuentra implementado en la clase **UniformCrossover** y recibe el nombre de *UniformCrossover*. Este operador actúa sólo con *GABin* y *GABinNN*. Crea un sólo hijo a partir de dos padres. Tiene una propiedad llamada *bias* que determina la probabilidad de tomar el bit del mejor padre.

[1]Los nuevos operadores sexuales que aparezcan no tienen por qué cumplir estas características.

C.3.3 Recombinación de 1 Punto para 2–D

Este operador se encuentra implementado en la clase **TableBinCrossover** y recibe el nombre *TableBinCrossover*. Actúa sólo con individuos *GABin* y *GABinNN*. Está pensado para usarlo cuando el cromosoma representa una tabla donde se han colocado los elementos de la tabla por filas y cada elemento viene representado por una palabra binaria de longitud fija. Las propiedades que tiene este operador son: `rows`, `columns` y `bits`. La primera indica el número de filas de la tabla, la segunda el número de columna y la tercera la longitud en bits de cada elemento de la tabla. En nuestro trabajo este operador sólo ha sido aplicado al problema de ingenería del software.

C.3.4 Recombinación de Aristas (ERX)

Este operador se implementa en la clase **VRPERXOperator** y su nombre es *VRPERXOperator*. Actúa sólo sobre individuos *VRP*. No tiene propiedades. Este operador ha sido aplicado únicamente al problema VRP.

C.3.5 Recombinación Parcialmente Mapeada (PMX)

Este operador se implementa en la clase **PMX** y su nombre es *PMX*. Actúa sobre individuos *GAInt* y asume que representan permutaciones, así pues el array de enteros de los individuos debe estar estar formado por números consecutivos y comenzando en cero. Este operador sólo genera un hijo y no posee propiedades.

C.3.6 Recombinación Cíclica

Este operador se implementa en **CycleCrossover** y su nombre es *Cycle*. Como el anterior, es un operador para permutaciones y actúa sobre individuos *GAInt*. Este operador devuelve un solo hijo y no tiene propiedades.

C.3.7 Recombinación para Estrategias Evolutivas

Este operador está implementado en **ESCrossover** y su nombre es *ESCrossover*. Recombina el vector de variables usando la recombinación discreta uniforme y los vectores de ángulos y desviaciones usando recombinación intermedia. Posee dos propiedades:

`probability` y `bias`. La primera establece la probabilidad de aplicar el operador y la segunda la probabilidad de tomar cada variable del mejor padre. Devuelve un solo individuo.

C.4 Operadores de Mutación

Los operadores de mutación implementados lanzan una excepción si reciben un array nulo. En el resto de los casos devuelven un array de la misma longitud que el de entrada. Son tres los operadores de mutación: *BitFlipMutation*, *PMutation* y *ESMutation*.

BitFlipMutation se implementa en **BitFlipMutation**. El operador actúa sobre individuos *GABin* y *GABinNN*. Recorre la cadena de bits invirtiéndolos con cierta probabilidad determinada por la propiedad `probability`.

PMutation se implementa en **PMutation**. Actúa sobre individuos *GAInt* y asume que representan permutaciones. Su labor consiste en intercambiar los elementos de dos posiciones distintas escogidas aleatoriamente. La probabilidad de realizar el intercambio se establece en la propiedad `probability`.

ESMutation se implementa en **ESMutation**. Actúa sobre individuos *ES* y *ESNN*. Es el operador de mutación de las Estrategias Evolutivas. La probabilidad de aplicarse a un individuo viene determinada por la propiedad `probability`.

C.5 Operadores de búsqueda dirigida

Los operadores de búsquedad dirigida son específicos de cada problema y trabajan sobre una solución dada para tratar de mejorarla. Nosotros hemos implementado cuatro de estos operadores que describimos a continuación.

Los operadores *VRPLambdaOpt* y *VRPLambdaInter* se encuentran implementados en las clases **VRPLambdaOpt** y **VRPLambdaInter** respectivamente. Ambos actúan sólo sobre individuos *VRP*. El primero mejora las soluciones aplicando el algoritmo λ-opt en su versión generalizada para el problema VRP (Sección 3.5) mientras que el segundo emplea el algoritmo λ-Intercambio (Sección 3.6). Ambos poseen una propiedad llamada `probability` que indica la probabilidad de aplicar el operador a un individuo. El parámetro λ de *VRPLambdaOpt* se establece por medio de la propiedad `lambda`. El

operador *VRPLambdaInter* también tiene una propiedad llamada `lambda` para establecer su parámetro λ (que no tiene nada que ver con el λ de λ-opt).

El operador *NetTraining* se encuentra implementado en la clase **NetTrainingMutation**. Este operador aplica un algoritmo de entrenamiento de redes neuronales a los individuos con cierta probabilidad. Puede actuar en principio sobre cualquier individuo. Posee un objeto que implementa la interfaz **IndividualToNet** del paquete ***pfc.algorithms.ea*** que es el que realiza la interpretación del individuo para convertirlo a un objeto **FFNN** con el que pueden trabajar los algoritmos de entrenamiento de redes. La probabilidad de aplicación viene determinada por la propiedad `probability`. El nombre del algoritmo de entrenamiento se encuentra en la propiedad `name-training`. La propiedades del algoritmo de entrenamiento se establecen como si se tratara de propiedades de *NetTraining*. En la Figura C.2 vemos un ejemplo de uso de este operador para entrenar las redes con una época del algoritmo Levenberg-Marquardt.

```
operator.1 = NetTraining
operator.1.parameter.probability = 1.0
operator.1.parameter.name-training=lev-mar
operator.1.parameter.learning-rate=1.0
operator.1.parameter.beta = 10
operator.1.parameter.mu = 0.001
operator.1.parameter.mu_max = 1e10
operator.1.parameter.terminating-condition=epochs 1
operator.1.parameter.mode=MATLAB_STYLE
```

Figura C.2: El operador *NetTraining* para entrenar con Levenberg-Marquardt.

El operador *ECCRepulsion* se encuentra implementado en la clase **ECCRepulsion** y trata de mejorar las soluciones del problema ECC usando el algoritmo presentado en la Sección 3.8. Este operador se aplica a individuos *GABin* que representan soluciones del ECC. Las propiedades del operador son:

- `steps`: Número de partículas que se mueven una detrás de otra.
- `words`: Número de palabras del código.
- `bits`: Número de bits de cada palabra.

- `threshold`: Valor mínimo que ha de tener la componente de la fuerza en la dirección del movimiento para que se mueva una partícula.
- `probability`: Probabilidad con la que se aplica el operador a un individuo.

C.6 Operadores de Reemplazo

Los operadores de reemplazo lanzan una excepción cuando reciben un array nulo. Los operadores de reemplazo son los encargados de generar la nueva población. Hay dos de estos operadores llamados: *GenerationalReplacement* y *ElitistReplacement* implementados en las clases **GenerationalReplacement** y **ElitistReplacement** respectivamente. Ambos operadores devuelven un array nulo.

El primer operador realiza un reemplazo (μ, λ). Toma los individuos de entrada y elige a los mejores para formar la nueva población. Si el número de individuos del array de entrada es menor que el tamaño de la población se repiten individuos.

El segundo operador realiza un reemplazo $(\mu + \lambda)$. De la unión de la población y los individuos del array de entrada escoge los mejores para formar la población.

C.7 Operadores compuestos

Estos operadores son en realidad constructores[2] de operadores. Se usan para formar operadores más complejos que son combinación de varios operadores. Todos tienen uno o más operadores contenidos que deben definirse usando las propiedades de estos constructores. Existen cuatro de estos constructores de operadores: *FirstTime*, *Gap*, *Parallel* y *Sequential* implementados en las clases **FirstTimeOperator**, **GapOperator**, **ParallelComposition** y **SequentialComposition** respectivamente.

El constructor *FirstTime* contiene un único operador. Éste es aplicado sólo la primera vez que se ejecuta el método `operation(Individual [] ind)` de *FirstTime*. El resto de las veces se devuelve el mismo array de individuos que recibe como parámetro. El nombre del operador contenido se indica mediante la propiedad `opname` y sus propiedades

[2] Al hablar de constructor en esta sección no nos referimos al concepto de constructor de una clase Java. Lo que queremos decir es que estos operadores toman otros operadores y construyen un operador más complejo a partir de ellos

mediante las propiedades `op.parameter.<propiedad>`. En la Figura C.3 podemos ver un fragmento de fichero de configuración en el que *FirstTime* contiene a un operador de mutación. La utilidad principal de este operador es usar una inicialización alternativa para la población o introducir alguna solución obtenida por medio de otro algoritmo (seeding).

```
operator.0 = FirstTime
operator.0.parameter.opname = BitFlipMutation
operator.0.parameter.op.parameter.probability = 0.001
```

Figura C.3: El constructor *FirstTime* conteniendo a otro operador.

El constructor *Gap* contiene también un único operador que al igual que ocurría con *FirstTime* se indica mediante las propiedades `opname` y `op.parameter.<propiedad>`. Este operador es aplicado al array de entrada cada cierto número de ejecuciones del constructor. Cuando no se aplica el constructor devuelve el array de entrada. El número de ejecuciones del constructor entre dos aplicaciones del operador contenido se determina mediante la propiedad `gap`. Este operador es usado en combinación con los operadores de migración para determinar la frecuencia con que se realizan las migraciones.

El constructor *Parallel* contiene uno o más operadores (se lanza una excepción cuando es aplicado sin contener operadores). Su labor consiste en aplicar todos esos operadores al array de individuos de entrada, recoger la salida de los operadores y devolver la concatenación de todas estas salidas (Figura C.4). Para especificar los operadores contenidos se emplean las propiedades `opnum`, `op.<i>` y `op.<i>.parameter.<propiedad>`. La primera indica el número de operadores contenidos. Las propiedades de la forma `op.<i>` donde i es un número entre 0 y `opnum` indican el nombre del i-ésimo operador y las propiedades de la forma `op.<i>.parameter.<propiedad>` especifican una propiedad del operador i-ésimo. Este operador ha sido usado principalmente en combinación con el operador de recepción de individuos para añadir los individuos recibidos a la población. En la Figura C.5 podemos ver un fragmento de un fichero de configuración que usa el constructor *Parallel*. Dicho constructor contiene dos operadores: *Dummy* sin propiedades definidas y *Receiver* con dos propiedades: `port` y `max-con`.

El constructor *Sequential* contiene cero o más operadores. Su funcionamiento es como sigue. Aplica el primer operador al array de individuos de entrada. La salida de este operador es la entrada del siguiente y así va usando todos los operadores hasta acabar

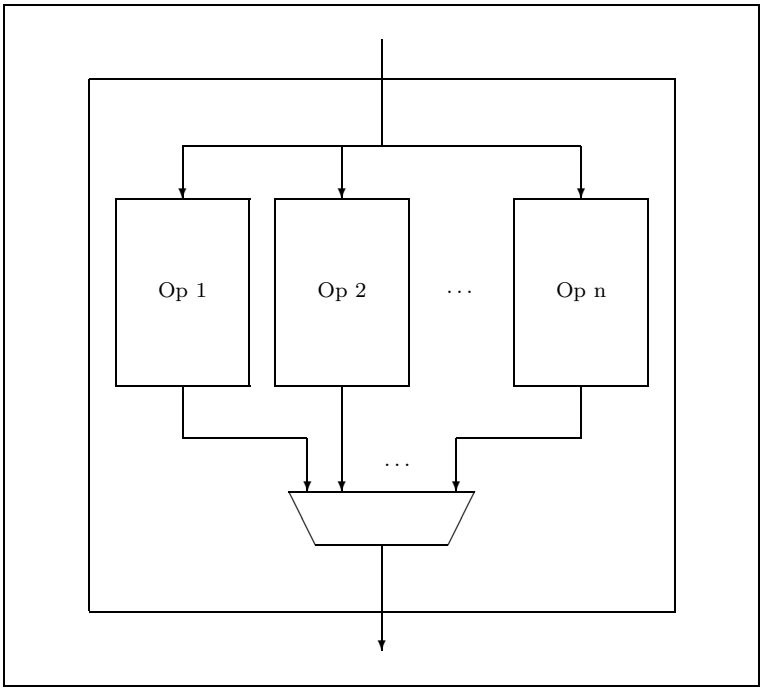

Figura C.4: Funcionamiento del constructor de operadores *Parallel*.

```
operator.4 = Parallel
operator.4.parameter.opnum = 2
operator.4.parameter.op.0 = Dummy
operator.4.parameter.op.1 = Receiver
operator.4.parameter.op.1.parameter.port = 3012
operator.4.parameter.op.1.parameter.max-con = 1
```

Figura C.5: El constructor *Parallel* conteniendo a dos operadores.

con el último. Entonces toma la salida de este último y la devuelve. Si no hay operadores devuelve la entrada (Figura C.6). La forma de especificar los operadores contenidos es igual que en *Parallel*. Este constructor encadena los operadores de la misma forma que lo hace el funcionamiento propio del algoritmo evolutivo. De hecho, este constructor no sería necesario de no ser por la presencia del resto de los constructores. Usado en combinación con *FirstTime* o *Gap* permite aumentar el número de operadores que se pueden aplicar en cada caso (que en esos constructores está restringido a uno). En la Figura C.7 vemos un fragmento de un fichero de configuración que usa este constructor. Ahí vemos al con-

structor *FirstTime* que tiene un operador compuesto contenido. Este operador compuesto es el constructor *Sequential* con tres operadores simples. El primero es *VRPIntroSol* y tiene una propiedad definida: `individual`. El segundo y tercer operadores son *Evaluate* y *ElitistReplacement* respectivamente.

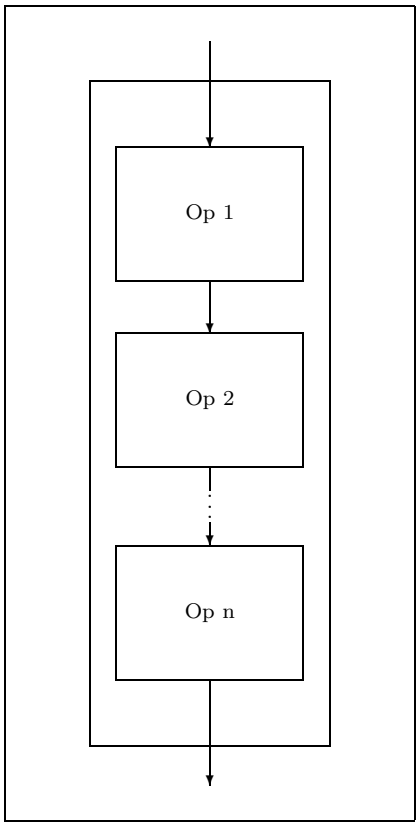

Figura C.6: Funcionamiento del constructor de operadores *Sequential*.

```
operator.0 = FirstTime
operator.0.parameter.opname = Sequential
operator.0.parameter.op.parameter.opnum = 3
operator.0.parameter.op.parameter.op.0 = VRPIntroSol
operator.0.parameter.op.parameter.op.0.parameter.individual = VRP
operator.0.parameter.op.parameter.op.1 = Evaluate
operator.0.parameter.op.parameter.op.2 = ElitistReplacement
```

Figura C.7: El constructor *Sequential* conteniendo a tres operadores.

Ya hemos visto los cuatro operadores compuestos que hemos implementado. Estos operadores deben sobreescribir los métodos `setPopulation(Population pop)`, `waitForReady()`, `waitForEnd()` y `endOperation()` de **EAOperator**. Puesto que el nuevo código de estos algoritmos es común y además común a cualquier operador compuesto que se implemente en el futuro, se han implementado dichos métodos en una clase llamada **ContainerOperator** que es subclase de **EAOperator** y superclase de los operadores compuestos.

C.8 Operadores de Inicialización

Lo primero que hace el algoritmo evolutivo es inicializar la población. Cada individuo requiere una inicialización distinta y por ese motivo existe un método abstracto para tal fin en la clase **Individual**. Cada tipo de individuo debe implementar este método de acuerdo a sus características. A veces nos puede interesar la inicialización por defecto de los individuos y otras veces no. En nuestro caso, para inicializar los individuos *GABin*, *GABinNN* y *GAInt* se elige para cada posición de la cadena un 0 o un 1 aleatoriamente con igual probabilidad. Los individuos *ES* y *ESNN* inicializan el vector de variables y desviaciones típicas con un valor real en el intervalo $[0, 1]$ y los ángulos con un valor real en el intervalo $[0, 2\pi]$. Para inicializar *VRP* se crean tantas rutas como clientes haya consistiendo cada ruta en un viaje del almacén al cliente y regreso al almacén.

Los operadores que presentamos aquí han sido desarrollados para inicializar la población de forma alternativa (como es el caso de *PInitialization*) o para incluir en ésta soluciones obtenidas al aplicar un algoritmo heurístico específico del problema (como ocurre con *VRPIntroSol* y *TAIntroSol*).

Los individuos *GAInt* han sido usados en las pruebas para representar permutaciones, sin embargo, su inicialización por defecto no es adecuada para este fin. El operador *PInitialization* implementado en la clase **PermInitialization** inicializa la población de individuos *GAInt* de forma que los individuos representen permutaciones. Este operador debe aplicarse sólo una vez al comienzo del algoritmo, por ello debe ser usado en combinación con *FirstTime*. Para inicializar un individuo este operador asigna a cada elemento del array de enteros su posición en dicho array y posteriormente realiza una serie de intercambios aleatorios de elementos del array. El número de intercambios viene determinado

por la propiedad `swaps`.

El operador *VRPIntroSol* implementado en la clase **VRPIntroSolOp** introduce una solución obtenida mediante el algoritmo Savings (Sección 3.4). Si el array de entrada es nulo o está vacío devuelve un array con un solo individuo que representa la solución de Savings. Si el array de entrada no está vacío devuelve el mismo array pero el primer individuo es sustituido por la solución del algoritmo.

El operador *TAIntroSol* implementado en la clase **TAGreedyAlgorithm** aplica el algoritmo greedy para TA (Sección 3.7) para obtener varias soluciones e introducirlas en la población o devolverlas a la salida. Posee las siguiente propiedades:

- `instance`: Nombre del fichero que contiene la instancia del TA.
- `affect-pop`: Toma un valor de verdad (*yes* o *no*) indicando si el algoritmo introduce las soluciones en la población o en el array de entrada para devolverlos a la salida.
- `feasible-only`: Toma un valor de verdad (*yes* o *no*) indicando si el algoritmo sólo genera soluciones válidas o puede generar soluciones no válidas.
- `fraction`: Indica la fracción de individuos generados sobre el total. Si los individuos van a ser introducidos en la población el total se refiere al tamaño de la población. Si los individuos van a ser introducidos en el array de entrada el total se refiere al número de individuos de ese array.

C.9 Operadores de Migración

Al afrontar la implementación de los operadores de migración se optó por no hacerlos dependientes de la topología de procesos. Como resultado, los operadores de migración implementados permiten emplear cualquier grafo dirigido como topología de comunicación. Para conseguir esto son necesarios dos operadores: *Receiver* y *Sender*. El primero se encuentra implementado en la clase **MigrationReceiver** y el segundo en **MigrationSender**. Llamaremos *emisor* al operador *Sender* y *receptor* a *Receiver*

El operador *Receiver* una vez inicializado se pone a la escucha de conexiones. Los operadores *Sender* inician esas conexiones. Un *Receiver* puede aceptar varias conexiones

y un *Sender* puede realizar conexiones a varios *Receiver*. Durante el funcionamiento del algoritmo, cada vez que se aplica el operador *Receiver* se devuelven los individuos que han llegado procedentes de los otros subalgoritmos y que son almacenados temporalmente en el operador hasta que es aplicado. Los individuos que haya en el array de entrada pasan a dsposición del heap. Cuando se aplica el operador *Sender*, los individuos del array de entrada son enviados a cada uno de los subalgoritmos a los que el operador está conectado y además son devueltos a la salida.

La migración de individuos se produce siempre de un emisor a un receptor. Para modificar la frecuencia con que los individuos son migrados se emplea el constructor *Gap* en combinación con *Sender*. A continuación describimos las propiedades de estos dos operadores de migración.

El operador *Sender* posee las siguientes propiedades:

- `destinations`: Número de receptores a los que se conecta.
- `destination.<i>.host`: Host donde se encuentra el i-ésimo receptor.
- `destination.<i>.port`: Puerto donde escucha el i-ésimo receptor.
- `destination.<i>.try-time`: Tiempo en milisegundos durante el que intenta realizar la conexión con el i-ésimo receptor antes de desistir. Si este valor es negativo, intenta conectar indefinidamente.
- `local-host`: Esta propiedad en combinación con la siguiente se usa en máquinas con varias direcciones IP para elegir la dirección IP y el puerto desde la que se realiza la conexión.
- `local-port`: Se usa con la propiedad anterior para elegir el puerto desde donde se realiza la conexión.

La últimas dos propiedades son opcionales y son útiles cuando realizamos las pruebas con una máquina que dispone de varios interfaces de red con direcciones distintas y deseamos elegir un interfaz concreto. Si está presente la propiedad `local-host` debe estar también `local-port`.

El operador *Receiver* posee las siguientes propiedades:

- `ip`: Dirección IP en la que atiende conexiones. Si esta propiedad no aparece atenderá las conexiones que lleguen a todas las IPs de la máquina. Esta propiedad es útil para elegir un interfaz de red concreto en el caso de máquinas con varios interfaces.
- `port`: Puerto donde admite conexiones.
- `max-con`: Número máximo de conexiones permitidas.

El receptor actúa como un servidor oyendo conexiones de un puerto indicado.

La topología de un algoritmo evolutivo paralelo viene especificada mediante un grafo dirigido donde los vértices representan subalgoritmos y los arcos representan las migraciones que se pueden hacer entre esos subalgoritmos. Por ejemplo, si el arco (i, j) pertenece al grafo, entonces se producen migraciones de individuos desde el subalgoritmo i al j. Aquellos subalgoritmos a los que llegue al menos un arco deben tener un receptor mientras que aquellos de los que sale al menos un arco deben tener un emisor. Pero con los operadores presentados en esta sección se puede ir más allá. Nada impide que en un mismo subalgoritmo se empleen varios receptores y/o varios emisores, dando mucha más flexibilidad al usuario para experimentar. El orden en que se inician los subalgoritmos no es relevante para crear satisfactoriamente la infraestructura de conexiones necesarias. No puede darse una situación de interbloqueo por muy caprichosa que sea la topología. La clave de esto se encuentra en que cuando un subalgoritmo es iniciado, todos sus operadores comienzan su proceso de inicialización y no tienen que esperar a que otro operador del mismo subalgoritmo acabe de inicializarse, lo que podría llevar a interbloqueo. En particular, todos los receptores se ponen a escuchar inmediatamente y los emisores no tienen problema para establecer la concexión con los receptores. Cuando todos los operadores de todos los subalgoritmos han sido inicializados comienza la ejecución de los subalgoritmos. Los emisores acaban su inicialización cuando han establecido la conexión. Por tanto, la ejecución comienza cuando todas las conexiones han sido realizadas. Esto significa que no hay que preocuparse de lanzar los subalgoritmos en un orden determinado. Tan sólo hay que preocuparse de que las direcciones de los destinos que se indican en el emisor son correctas (hay un receptor escuchando en esa dirección) y de dejar el tiempo adecuado para establecer la conexión (`try-time`). Las conexiones se realizan usando Sockets Java sobre TCP.

C.10 Otros Operadores

Incluimos en esta sección tres operadores que no podemos encuadrar en ninguna de las categorías anteriores. Se trata de *Dummy*, *Print* y *Evaluate*. Se encuentran implementados en las clases **DummyOperator**, **PrintOperator** y **EvaluateOperator** respectivamente. El operador *Dummy* simplemente devuelve el array de individuos recibido. El operador *Print* escribe por pantalla los individuos que recibe y los devuelve igual que *Dummy*. Por último el operador *Evaluate* se encarga de evaluar a los individuos que recibe. Para ello emplea el método `evaluate()` de los individuos. Luego devuelve el array de individuos igual que hace *Dummy*. Los tres operadores funcionan con cualquier número de individuos e incluso con un array nulo.

Bibliografía

[ASW94] Faris N. Abuali, Dale A. Schoenefeld, and Roger L. Wainwright. Terminal assignment in a communnications network using genetic algorithms. In *Proceedings of the 22nd Annual Computer Science Conference*, pages 74–81, Phoenix, March 1994. AZ.

[AT99] E. Alba and J. M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.

[AT00] E. Alba and J. M. Troya. Influence of the migration policy in parallel distributed GAs with structured and panmictic populations. *Applied Intelligence*, 12(3):163–181, 2000.

[AT01] E. Alba and J. M. Troya. Gaining new fields of application for OOP: the parallel evolutionary algorithm case. *Journal of Object Oriented Programming*, December (web version only) 2001.

[AVZ01] Erik Agrell, Alexander Vardy, and Kenneth Zeger. A table of upper bounds for binary codes. *IEEE Transactions on Information Theory*, 47(7):3004–3006, 2001.

[AW92] N.P. Archer and S. Wang. Application of the back propagation neural network algorithm with monotonicity constraints for two-group classification problems. *Decision Sciences*, 24(1):60–75, 1992.

[Bac96] T. Back. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, New York, 1996.

[BFM97] T. Bäck, D.B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York NY, 1997.

[BM92] K. P. Bennett and O. L. Mangasarian. Robust linear programming discrimination of two linearly inseparable sets. *Optimization Methods and Software*, 1:23–34, 1992.

[CCZ01] Carl K. Chang, Mark J. Christensen, and Tao Zhang. Genetis algorithms for project management. *Annals of Software Engineering*, 11:107–139, 2001.

[CFW98] H. Chen, N.S. Flann, and D.W. Watson. Parallel genetic simulated annealing: A massively parellel simd algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):126–136, 1998.

[CMT79] N. Christofides, A. Mingozzi, and P. Toth. The vehicle routing problem. In N. Christofides, A. Mingozzi, P. Toth, and C. Sandi, editors, *Combinatorial Optimization*, Wiley, Chichester, 1979.

[COS00] D.W. Corne, M.J. Oates, and G.D. Smith. *Telecommunications Optimization: Heuristics and Adaptative Techniques*. Wiley, 2000.

[CW64] G. Clarke and J.W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–581, 1964.

[DJ90] K. Dontas and K. De Jong. Discovery of aximal distance codes using genetic algorithms. In *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence*, pages 905–811, Herndon, VA, 1990. IEEE Computer Society Press, Los Alamitos, CA.

[FOW66] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.

[HAFF99] T. Haupt, E. Akarsu, G. Fox, and W. Furnanski. Web based meta-computing. http://www.npac.syr.edu/users/haupt/WebFlow, 1999.

[Hay94] Simon Haykin. *Neural Networks. A Comprehensive Foundation*. Prentice Hall, 1994.

[HM94] Martin T. Hagan and Mohammad B. Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE Transactions on Neural Networks*, 5(6), November 1994.

[HMS92] J.V. Hensen, J.B. McDonald, and J.D. Stice. Artificial intelligence and generalized qualitative-response models: an empirical test on two audit decision-making domains. *Decision Sciences*, 23:708–723, 1992.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[Hol01] S. Holzner. *XML Complete.* McGraw-Hill, 2001.

[KC97] S. Khuri and T. Chiu. Heuristic algorithms for the terminal assignment problem. In *Proceedings of the 1997 ACM Symposium on Applied Computing*, pages 247–251. ACM Press, 1997.

[KWR93] J.W. Kim, H.R. Weistroffer, and R.T. Redmond. Expert systems for bond rating: a comparative analysis of statistical, rulebased, and neural network systems. *Expert Systems*, 10(3):167–171, 1993.

[LGPS99] G. Laporte, M. Gendreau, J.Y. Potvin, and F. Semet. Classical and modern heuristics for the vehicle routing problem. Technical Report G-99-21, Les Cahiers du GERAD, March 1999. Revised in October, 1999.

[LPS99] G. Laporte, M.G. Jean-Yves Potvin, and F. Semet. Classical and modern heuristics for the vehicle routing problem. Technical report, Les Cahiers du GERARD, March 1999.

[MF98] Z. Michalewicz and D.B. Fogel. *How to Solve It: Modern Heuristics.* Springer Verlag, Berlin Heidelberg, 1998.

[MSW90] O. L. Mangasarian, R. Setiono, and W.H. Wolberg. Pattern recognition via linear programming: Theory and application to medical diagnosis in: "large-scale numerical optimization". pages 22–30. SIAM Publications, Philadelphia, 1990.

[PHH93] E. Patuwo, M.Y. Hu, and M.S. Hung. Two-group classification using neural networks. *Decision Sciences*, 24(4):825–845, 1993.

[Pla01] D.S. Platt. *Introducing Microsoft®.NET*. Microsoft Press, 2001.

[Pre94] Lutz Prechelt. PROBEN1 — a set of neural network benchmark problems and benchmarking rules. Technical Report 21, Fakultät für Informatik Universität Karlsruhe, 76128 Karlsruhe, Germany, September 1994.

[PRS00] C.A. Pena-Reyes and M. Sipper. Evolutionary computation in medicine: An overview. *Artificial Intelligence in Medicine*, 19(1):1–23, 2000.

[Rec73] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog Verlag, Stuttgart, 1973.

[RSORS96] V.J. Rayward-Smith, I.H. Osman, C.R. Reeves, and G.D. Smith. *Modern Heuristic Search Methods*. John Wiley & Sons, Chichester, 1996.

[SCL92] L.M. Salchenberger, E.M. Cinar, and N.A. Lash. Neural networks: a new tool for predicting thrift failures. *Decision Sciences*, 23(4):889–916, 1992.

[SD00] Randall S. Sexton and Robert E. Dorsey. Reliable classification using neural networks: a genetic algorithm and backpropagation comparison. *Decision Support Systems*, 30:11–22, 2000.

[SHH93] V. Subramanian, M.S. Hung, and M.Y. Hu. An experimental evaluation of neural networks for classification. *Computers and Operation Research*, 20(7):769–782, 1993.

[TK92] K.Y. Tam and M.Y. Kiang. Managerial applications of neural networks: the case of bank failure prediction. *Management Science*, 38(7):926–947, 1992.

[TLZ99] K.C. Tan, L.H. Lee, and K.Q. Zhu. Heuristic methods for vehicle routing problem with time windows. September 1999.

[Whi00] Darrel Whitley. *Permutations*, volume 1, chapter 33.3, pages 274–284. IOP Publishing Lt, 2000.

[WM90] William H. Wolberg and O.L. Mangasarian. Multisurface method of pattern separation for medical diagnosis applied to breast cytology. In *Proceedings of the National Academy of Sciences*, volume 87, pages 9193–9196, U.S.A, December 1990.

[WM97] W.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evoutionary Computation*, 1(1):67–82, 1997.

[Wol90] W. H. Wolberg. Cancer diagnosis via linear programming. *SIAM News*, 23(5):1 & 18, September 1990.

[YL97] X. Yao and Y. Liu. A new evolutionary system for evolving artificial neural networks. *IEEE Transactions on Neural Networks*, 8:694–713, 1997.

[YSM93] Y. Yoon, G. Swales, and T.M. Margavio. A comparison of discriminant analysis versus artificial neural networks. *Journal of the Operations Research Society*, 44(1):51–60, 1993.

[ZU01] Ahmed K. Zaman and C.E. Umrysh. *Developing Enterprise Java Applications with J2EE^{TM} and UML*. Addison-Wesley, 2001.