Proyecto LATIn
INICIATIVA LATINOAMERICANA DE LIBROS DE TEXTO ABIERTOS

ALFA
FINANCED BY THE EUROPEAN UNION

Inteligencia Artificial

AUTORES

Julio Cesar Ponce Gallegos
Aurora Torres Soto
Fátima Sayuri Quezada Aguilera
Antonio Silva Sprock
Ember Ubeimar Martínez Flor
Ana Casali
Eliana Scheihing
Yván Jesús Túpac Valdivia
Ma. Dolores Torres Soto
Francisco Javier Ornelas Zapata
José Alberto Hernández A.
Crizpín Zavala D.
Nodari Vakhnia
Oswaldo Pedreño

Inteligencia Artificial

1a ed. - Iniciativa Latinoamericana de Libros de Texto Abiertos (LATIn), 2014. 225 pag.

Primera Edición: Marzo 2014
Iniciativa Latinoamericana de Libros de Texto Abiertos (LATIn)
http://www.proyectolatin.org/

Este texto forma parte de la Iniciativa Latinoamericana de Libros de Texto abiertos (LATIn), proyecto financiado por la Unión Europea en el marco de su Programa ALFA III EuropeAid.
El Proyecto LATIn está conformado por: Escuela Superior Politécnica del Litoral, Ecuador (ESPOL); Universidad Autónoma de Aguascalientes, México (UAA), Universidad Católica de San Pablo, Perú (UCSP); Universidade Presbiteriana Mackenzie, Brasil(UPM); Universidad de la República, Uruguay (UdelaR); Universidad Nacional de Rosario, Argentina(UR); Universidad Central de Venezuela, Venezuela (UCV), Universidad Austral de Chile, Chile (UACH), Universidad del Cauca, Colombia (UNICAUCA), Katholieke Universiteit Leuven, Bélgica (KUL), Universidad de Alcalá, España (UAH), Université Paul Sabatier, Francia (UPS).

Índice general

Dedicatoria

Agradecemos a todas las personas involucradas en la escritura del libro asi como a a sus instituciones por su apoyo al proyecto LATIN.

Prólogo

Actualmente la Inteligencia Artificial es un area de la ciencia de gran interes por ser un area multidiciplinaria donde se realizan sistemas que tratan de hacer tareas y resolber problemas como lo hace un humano, asi mismo se trata de simular de manera artificial las formas del pensamiento y como trabaja el cerebro para tomar desiciones. Aunque en la realidad aun no se a podido realizar todo lo que las personas sueñan al conocer esta area o al ver lo que se muestra en la ciencia ficcion es un area que poco a poco va ganando terreno al estar presente en muchas aplicaciones, aparatos, dispositivos que utilizamos de manera cotidiana.

¿Quien utilizará el libro?

Este libro va dirigido a los estudiantes, profesores y profesionales que tienen algun interes o relación con la inteligencia artificial. Con el objetivo de que pueda ser utilizado como un libro de texto que apoye a conocer los fundamentos y aplicaciones que sirvan para la generación de sistemas inteligentes.

Objetivos del libro

Introducir al lector al area de la Inteligencia Artificial y a algunas desus ramas de conocimiento, dandole al usuario una vision general del area y de algunas de las ramas y aplicaciones existentes.

Organización del libro

Este se encuentra dividido en 8 secciones o capitulos, donde cada uno de estos da una introduccion al contenido del mismo menciona los antecedentes necesarios y hace refenecia a la revision bibliografica relacionada.

Abstract

Este libro es una introducción al area de la Inteligencia Artificial y presenta algunas de las aplicaciones que puede tener en la vida real en diversos campos de aplicación, El libro esta compuesto de ocho capítulos los cuales abarcan los antededentes, algunos conceptos importantes para la resolución de problemas como es la representación de conocimiento, el planteamiento de los problemas. Asimismo se menciona la teoría de agentes por un lado y por otro lo que es el aprendizaje computacional. Otra area que se aborta es la computación evolutivo y los algoritmos bioinspirados para la resolución de problemas, dandole enfasis a los problemas de optimizacion. Por ultimo se menciona una nueva tendencia en el area de las ciencias computacionales como es el uso de las GPUs para trabajar de una manera mas rapida al realizar el procesamiento en paralelo.

Abreviaturas

Algoritmos Evolutivos (EA)
Algoritmos Genéticos (AG)
Ingeniería de Software Orientada a Agentes (AOSE)
Inteligencia Artificial (IA)
Solucionador General de Problemas(GPS)
Optimización con Colonias de Hormigas (ACO)
Optimización con Cúmulo de Particulas (PSO)
Problema del Agente Viajero (TSP)
Sistemas Multi-Agentes (MAS)

Glosario

Inteligencia Artificial: Rama de las ciencias computacionales preocupada por la automatización de la conducta inteligente.

Agente Inteligente: Ente capaz de persivir su entorno a traves de sensores y actuar sobre él.

Algoritmos Evolutivos.- Son algoritmos basados en la evolución que puede darse a nivel de un individuo, una población de manera biológica o cultural.

1 — Introducción y Antecedentes de la Inteligencia Artificial

Julio Cesar Ponce Gallegos y Aurora Torres Soto
Universidad Autónoma de Aguascalientes, México

1.1 OBJETIVO

Este capítulo tiene como objetivo establecer un punto de partida para el estudio de la Inteligencia Artificial mediante el conocimiento de sus orígenes y descripción general. El lector hará una revisión histórica de los personajes y las herramientas que sentaron las bases de la Inteligencia Artificial tal como la conocemos el día de hoy.

El conocimiento que adquiera el estudiante podrá ser autoevaluado mediante la aplicación de un cuestionario al final del capítulo.

1.2 RESUMEN DEL CAPÍTULO

A lo largo de este capítulo, se hará una descripción de los orígenes e historia de la inteligencia artificial, así como de algunas contribuciones que esta área ha heredado de otras disciplinas. También se discutirán algunas de las definiciones más comúnmente aceptadas y los diferentes modelos que se emplean para describirla.

También se hará una descripción del conocido como “test de Turing”, que por mucho tiempo se consideró como la prueba que validaría de forma definitiva la existencia de máquinas inteligentes.

Después de haber revisado las principales aplicaciones y herramientas derivadas de la Inteligencia Artificial, en la que se discuten a grandes rasgos los lenguajes de programación desarrollados, algunos sistemas expertos y muy brevemente lo que se entiende por Shell; se hará una revisión de los diferentes tópicos de la Inteligencia Artificial.

1.3 CONOCIMIENTOS PREVIOS

El alumno debe contar con conocimientos de razonamiento lógico, programación, matemáticas, estos conocimientos son la base para el desarrollo de modelos matemáticos que permiten llegar a la solución de un problema, así como la modelación de sistemas que resuelvan problemas como si lo hiciera un humano.

1.4 INTRODUCCION

La Inteligencia Artificial (IA) es una de las ramas de las ciencias de la computación que más interés ha despertado en la actualidad, debido a su enorme campo de aplicación. La búsqueda de mecanismos que nos ayuden a comprender la inteligencia y realizar modelos y simulaciones de estos, es algo que ha motivado a muchos científicos a elegir esta área de investigación.

El origen inmediato del concepto y de los criterios de desarrollo de la "IA" se remonta a la intuición del genio matemático inglés Alan Turing y el apelativo "Inteligencia Artificial" se debe a McCarthy quien organizo una conferencia en el Darmouth College (Estados Unidos) para discutir la posibilidad de construir máquinas "inteligentes"; a esta reunión asistieron científicos investigadores de conocida reputación en el área de las ciencias computacionales como: Marvin Minsky, Nathaniel Rochester, Claude Shannon, Herbert Simon y Allen Newell. Como resultado de esta reunión, se establecieron los primeros lineamientos de la hoy conocida como Inteligencia Artificizal; aunque anteriormente ya existian algunos trabajos relacionados.

Desde su origen, la IA tuvo que lidiar con el conflicto de que no existía una definición clara y única de inteligencia; así es que no es de sorprender que aún en la actualidad, no exista una definición única de ella. Así como la Psicología ha identificado diferentes tipos de inteligencia humana (emocional, interpersonal, musical, lingüística, quinestésica, espacial, etc.), las distintas definiciones de la inteligencia artificial hacen énfasis en diferentes aspectos; aunque existen similitudes entre ellas. A continuación se presentan algunas de las definiciones iniciales de esta área.

- Estudio de la computación que observa que una maquina sea capaz de percibir, razonar y actuar (Winston, 1992).
- Ciencia de la obtención de máquinas que logren hacer cosas que requerirían inteligencia si las hiciesen los humanos (Minsky, 1968).
- Nuevo esfuerzo excitante que logre que la computadora piense... máquinas con mentes, en el sentido completo y literal (Haugeland, 1985).
- Rama de la ciencia computacional preocupada por la automatización de la conducta inteligente (Luger and Stubblefield, 1993).
- Maquina Inteligente es la que realiza el proceso de analizar, organizar, y convertir los datos en conocimiento, donde el conocimiento del sistema es información estructurada adquirida y aplicada para reducir la ignorancia o la incertidumbre sobre una tarea espesifica a realizar por esta (Pajares y Santos, 2006).

Originalmente la Inteligencia Artificial se construyó en base a conocimientos y teorías existentes en otras áreas del conocimiento. Algunas de las principales fuentes de inspiración y conocimientos que nutrieron a esta área son las ciencias de la computación, la filosofía, la lingüística, las matemáticas y la psicología. Cada una de estas ciencias contribuyó no solamente con los conocimientos desarrollados en ellas, sino con sus herramientas y experiencias también; contribuyendo así a la gestación y desarrollo de esta nueva área del conocimiento.

Los filósofos como Sócrates, Platón, Aristóteles, Leibniz desde el año 400 aC, sentaron las bases para la inteligencia artificial al concebir a la mente como una máquina que funciona a partir del conocimiento codificado en un lenguaje interno y al considerar que el pensamiento servía para determinar cuál era la acción correcta que había que emprender. Por ejemplo, Aristóteles quien es considerado como el primero (300aC) en

describir de forma estructurada la forma como el ser humano produce conclusiones racionales a partir de un grupo de premisas; contribuyó con un conjunto de reglas conocidas como silogismos que actualmente son la base de uno de los enfoques de la Inteligencia Artificial.

Sin embargo, la filosofía no es la única ciencia que ha heredado sus frutos a esta área; pues otras contribuciones no menos importantes son las siguientes:

Las matemáticas proveyeron las herramientas para manipular las aseveraciones de certeza lógica así como aquellas en las que existe incertidumbre de tipo probabilista; el cálculo por su lado, bridó las herramientas que nos permiten la modelación de diferentes tipos de fenómenos y fueron también las matemáticas, quienes prepararon el terreno para el manejo del razonamiento con algoritmos.

La Psicología ha reforzado la idea de que los humanos y otros animales pueden ser considerados como máquinas para el procesamiento de información, psicólogos como Piaget y Craik definieron teorías como el conductismo – psicología cognitiva.

Las Ciencias de la Computación -comenzó muy poco antes que la Inteligencia Artificial misma. Las teorías de la IA encuentran un medio para su implementación de artefactos y modelado cognitivo a través de las computadoras. Los programas de inteligencia artificial por general son extensos y no funcionarían sin los grandes avances de velocidad y memoria aportados por la industria de cómputo.

La Lingüística se desarrolló en paralelo con la IA y sirve de base para la representación del conocimiento (Chomsky). La lingüística moderna nació casi a la par que la Inteligencia Artificial y ambas áreas han ido madurando juntas; al grado que existe un área híbrida conocida como lingüística computacional o procesamiento del lenguaje natural. Los lingüistas han demostrado que el problema del entendimiento del lenguaje es mucho más complicado de lo que se había supuesto en 1957.

La economía, como una área experta en la toma de decisiones, debido que éstas implican la pérdida o ganancia del rendimiento; brindó a la IA una serie de teorías (Teoría de la decisión –que combina la Teoría de la Probabilidad y la Teoría de la utilidad; Teoría de juegos – para pequeñas economías; Procesos de decisión de Markov – para procesos secuenciales; entre otras) que la posibilitaron para la toma de “buenas decisiones”.

Finalmente, la Neurociencia, ha contribuido a la IA con los conocimientos recabados hasta la fecha sobre la forma como el cerebro procesa la información.

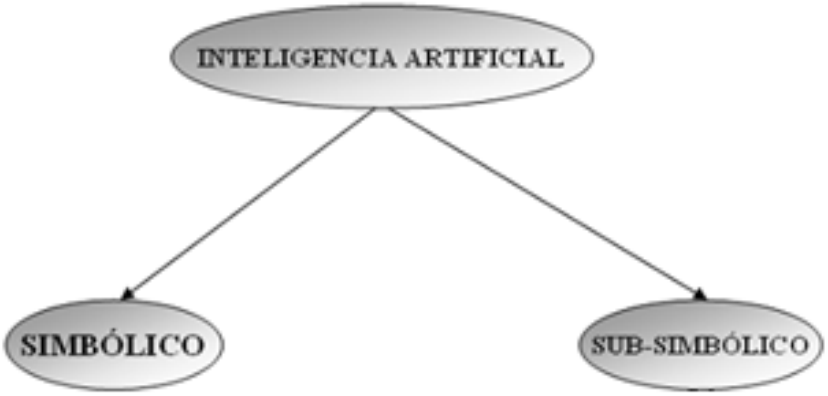
1.4.1 Clasificación de la Inteligencia Artificial

La escuela clásica dentro de la IA, utiliza representaciones simbólicas basadas en un número finito de primitivas y de reglas para la manipulación de símbolos (por ejemplo, redes semánticas, lógica de predicados, etc.), los cuales fueron y siguen siendo parte central de dichos sistemas.

Otro tipo de representación es el llamado sub-simbólico, el cual utiliza representaciones numéricas (o sub-simbólicas) del conocimiento. Aunque la mayor parte de los libros de IA (Hebb, 1949 , Minsky and Papert 1969) sólo enfatizan el trabajo realizado por Rosenblatt y Widrow en la década de los 50’s con redes neuronales en este tipo de representación.

El enfoque sub-simbólico de la IA se caracteriza por crear sistemas con capacidad de aprendizaje. Éste se puede obtener a nivel de individuo imitando el cerebro (Redes Neuronales), a nivel de especie, imitando la evolución. Hasta hace poco era común hablar de Algoritmos Genéticos (AG) en general, en vez de identificar diferentes tipos de AE, ya que el resto de los algoritmos se pueden interpretar como variaciones o mejoras de los AG.

Figura 1.1: Enfoques de clasificación de la Inteligencia Artificial



En la actualidad, la IA empieza a extender sus áreas de investigación en diversas direcciones y trata de integrar diferentes métodos en sistemas a gran escala, en su afán por explotar al máximo las ventajas de cada una de estas en una gran cantidad de áreas del conocimiento ya que se realizan aplicaciones en una gran cantidad de ares del conocimiento como son la medicina, biología, ingeniera, educación, etc..

En la actualidad existen nuevas técnicas que utilizan el enfoque sub-simbólico como son los algoritmos de optimización con colonias de hormigas, sistema inmune, cúmulo de partículas, entre otros, los cuales están inspirados en los comportamientos emergentes de la naturaleza.

1.4.2 Historia de la Inteligencia Artificial

Desde tiempos inmemoriales, el hombre ha buscado la materialización del deseo de crear seres semejantes a él; pasando por la creación de artefactos con aspecto, movimientos y hasta comportamiento similar al que presentamos los seres humanos. El ruso Isaac Asimov (1920-1992), escritor e historiador, narraba sobre objetos y situaciones que en su tiempo eran ciencia-ficción; sin embargo, con el paso del tiempo, muchas de ellas se han ido volviendo realidad. Asimov, en su libro Runaround describió lo que el día de hoy son las tres leyes de la robótica. Su obra literaria serviria como motivación para que los científicos e ingenieros trataran de hacerla realidad.

En los años 50 cuando se logra realizar un sistema que tuvo cierto éxito, se llamó el Perceptrón de Rossenblatt. Este era un sistema visual de reconocimiento de patrones en el cual se aunaron esfuerzos para que se pudieran resolver una gama amplia de problemas, pero estas energías se diluyeron enseguida.

Aproximadamente en ese tiempo, el matematico inglés Alan Turing (1912-1954) propuso una prueba con la finalidad de demostrar la existencia de “inteligencia” en un dispositivo no biológico. Esta prueba conocida como “test de Turing” se fundamenta en la hipótesis

de que si una máquina se comporta en todos aspectos como inteligente, entonces debe ser inteligente (Alan Turing, 1950). Como consecuencia de esta prueba, muchos de los esfuerzos de los investigadores en ese tiempo, se enfocaron en la redacción de sistemas de inteligencia artificial lingüísticos, lo que marcó el nacimiento de los conocidos como "chatbots" (robots de plática). A pesar de que ya se habían realizado investigación sobre el diseño y las capacidades de las entidades no biológicas, el trabajo de Alan Turing de 1950, concentró el interés de la comunidad científica en el desarrollo de las "máquinas inteligentes". Dos de las contribuciones más importantes de Alan Turing son el diseño de la primera computadoracapaz de jugar al ajedrez y el establecimiento de la naturaleza simbólica de la computación (ITAM, 1987).

Posteriormente, en 1957 Alan Newell y Herbert Simon, que trabajaban en la demostración de teoremas y el ajedrez por ordenador logran crear un programa llamado GPS (General Problem Solver). Este era un sistema donde el usuario definía un entorno en función de una serie de objetos y los operadores que se podían aplicar sobre ellos. Este programa fue redactado mediante el uso de IPL (Information Processing Languaje) y es considerado como el primer programa en el que se separó la información relacionada con el problema de la estrategia empleada para darle solución. El GPS se basó en el trabajo previamente desarrollado de sus autores sobre máquinas lógicas y aunque fue capaz de resolver problemas como el de "Las Torres de Hanoi"; no pudo resolver problemas ni del mundo real, ni médicos ni tomar decisiones importantes. El GPS manejaba reglas heurísticas que la conducían hasta el destino deseado mediante el método del ensayo y el error (Newell y Simon ,1961). Varios años más tarde; en los años 70, un equipo de investigadores dirigido por Edward Feigenbaum comenzaría a elaborar un proyecto para resolver problemas de la vida cotidiana (problemas más concretos); dando origen a lo que se conocería como los sistemas expertos.

En 1958 McCarthy desarrolló un lenguaje de programación simbolica cuando estaba trabajando en el MIT; dicho lenguaje es utlilizado aún en la actualidad y es conocido como LISP. El nombre LISP deriva de "LISt Processing" (Procesamiento de LIStas). Las listas encadenadas son una de las estructuras de datos importantes del Lisp.

En el año 1965 Joseph Weizenbaum construyo el primer programa interactivo el cual consistia en que un usuario podia sostener una conversación en ingles con una computadora utilizando una comunicación por escrito, este sistema fue denominado ELIZA.

El primer sistema experto fue el denominado Dendral, un intérprete de espectrograma de masa construido en 1967, pero el más influyente resultaría ser el Mycin de 1974. El Mycin era capaz de diagnosticar trastornos en la sangre y recetar la correspondiente medicación, todo un logro en aquella época que incluso fueron utilizados en hospitales (como el Puff, variante de Mycin de uso común en el Pacific Medical Center de San Francisco, EEUU)

Ya en los años 80, se desarrollaron lenguajes especiales para utilizar con la Inteligencia Artificial, tales como el LISP o el PROLOG. Es en esta época cuando se desarrollan sistemas expertos más refinados, como por ejemplo el EURISKO. Este programa perfecciona su propio cuerpo de reglas heurísticas automáticamente, por inducción.

También podemos destacar la importante intervención de Arthur Samuel, que desarrollo un programa de juego de damas capaz de aprender de su propia experiencia; Selfridge, que estudiaba el reconocimiento visual por computadora.

A partir de este grupo inicial, se formaron dos grandes "escuelas" de I.A.: Newell y Simon

lideraron el equipo de la Universidad de Carnegie-Mellon, proponiéndose desarrollar modelos de comportamiento humano con aparatos cuya estructura se pareciese lo más posible a la del cerebro (lo que derivó en la postura "conexionista" y en las "redes neuronales" artificiales).

McCarthy y Minsky formaron otro equipo en el Instituto Tecnológico de Massachusett (MIT), centrándose más en que los productos del procesamiento tengan el carácter de inteligente, sin preocuparse por que el funcionamiento o la estructura de los componentes sean parecidas a los del ser humano.

Ambos enfoques sin embargo, persiguen los mismos objetivos prioritarios de la I.A.: "entender la inteligencia natural humana, y usar máquinas inteligentes para adquirir conocimientos y resolver problemas considerados como intelectualmente difíciles".

La historia de la IA ha sido testigo de ciclos de éxito, injustificado optimismo y la consecuente desaparición de entusiasmo y apoyos financieros. También han habido ciclos caracterizados por la introducción de nuevos y creativos enfoques y de un sistemático perfeccionamiento de los mejores. Por sus implicaciones con áreas como la medicina, psiclogía, biología, ética y filosofía entre otras, esta rama del conocimiento ha tenido que lidiar con fuertes grupos oponentes y críticas desde sus orígenes; sin embargo, siempre existió un grupo de personas interesadas en el área lo que permitio que se consolidara como un area del conocimiento de gran interes para la investigación científica.

1.4.3 Modelos de Inteligencia

Existe una clasificación de los modelos de inteligencia Artificial que se basa en el objetivo y la forma en que trabaja el sistema, esta clasificación de manera inicial se veía como clases independientes, sin embargo, en la actualidad los sistemas mezclan características de ellas.

1.4.4 Sistemas que Piensan como Humanos

El modelo es el funcionamiento de la mente humana.

Se intenta establecer una teoría sobre el funcionamiento de la mente (experimentación psicológica).

A partir de la teoría se pueden establecer modelos computacionales.

Influencia de las Ciencias Cognitivas.GPS (General Problem Solver, 1963) - Newell & Simon no se preocupaban sobre cómo obtener la respuesta correcta – sino sobre por qué los sistemas llegaban a dar las respuestas que daban.

En los sistemas cognitivos, la mayoría de las investigaciones no son con computadoras sino con humanos y animales.

1.4.5 Sistemas que Actúan como Humanos

El modelo es el hombre; el objetivo es construir un sistema que pase por humano

Prueba de Turing: si un sistema la pasa es inteligente

Capacidades necesarias: Procesamiento del Lenguaje Natural, Representación del Conocimiento, Razonamiento, Aprendizaje

Figura 1.2: Modelos de inteligencia

Pasar la Prueba no es el objetivo primordial de la IA

La interacción de programas con personas hace que sea importante que éstos puedan actuar como humanos

1.4.6 Sistemas que Piensan Racionalmente

Las leyes del pensamiento racional se fundamentan en la lógica (silogismos de Aristóteles)

La lógica formal está en la base de los programas inteligentes (logicismo)

Se presentan dos obstáculos:

Es muy difícil formalizar el conocimiento

Hay un gran salto entre la capacidad teórica de la lógica y su realización práctica

Silogismos de Aristóteles

Lógica como base del esfuerzo

Lógica de Predicados

1.4.7 Sistemas actuantes racionales

Actuar racionalmente significa conseguir unos objetivos dadas unas creencias.

El paradigma es el agente racional, que se aplica, por ejemplo, a muchos sistemas robóticos.

Un agente percibe y actúa, siempre teniendo en cuenta el entorno en el que está situado.

Las capacidades necesarias:

- percepción
- procesamiento del lenguaje natural
- representación del conocimiento
- razonamiento
- aprendizaje automático

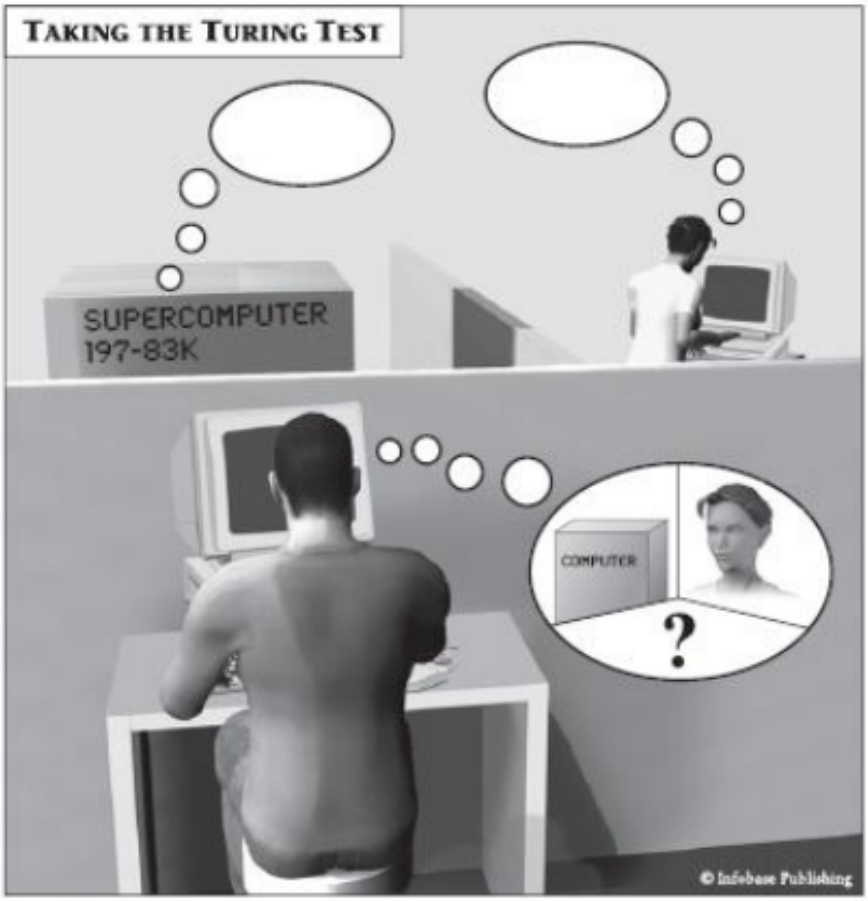
Visión de la actuación general y no centrada en el modelo humano

1.4.8 El test de Turing

La prueba de Turing (Alan Turing ,1950) intenta ofrecer una definición de Inteligencia Artificial que se pueda evaluar. Para que un ser o máquina se considere inteligente debe lograr engañar a un evaluador de que este ser o máquina se trata de un humano evaluando todas las actividades de tipo cognoscitivo que puede realizar el ser humano.

Si el diálogo que ocurra y el número de errores en la solución dada se acerca al número de errores ocurridos en la comunicación con un ser humano, se podrá estimar -según Turing- que estamos ante una máquina "inteligente".

Figura 1.3: "Test de Turing. Una persona debe establecer si sostuvo una charla con una máquina o con otra persona" (imagen tomada de Henderson, 2007)



Hoy por hoy, el trabajo que entraña programar una computadora para pasar la prueba es considerable. La computadora debería se capaz de lo siguiente:

1. Procesar un lenguaje natural: para así poder establecer comunicación satisfactoria, sea en español, inglés o en cualquier otro idioma humano.
2. Representar el conocimiento: para guardar toda la información que se le haya dado antes o durante el interrogatorio. Utilización de Base de Datos para receptar preguntas y luego almacenarlas.
3. Razonar automáticamente: Utiliza la información guardada al responder preguntas y obtener nuevas conclusiones o tomar decisiones.
4. Autoaprendizaje de la máquina: Con el propósito de adaptarse a nuevas circunstancias. El autoaprendizaje conlleva a la autoevaluación.

Para aprobar la prueba total de Turing, es necesario que la computadora esté dotada de:

1. Vista: Capacidad de percibir el objeto que se encuentra en frente suyo.
2. Robótica: Capacidad para mover el objeto que ha sido percibido.

1.4.9 Aplicaciones y herramientas derivadas de la Inteligencia Artificial

A lo largo de la historia de la Inteligencia Artificial sean ido desarrollando diferentes herramientas y aplicaciones entre las que podemos mencionar las siguientes:

- Lenguajes de Programación
- Aplicaciones y Sistemas Expertos
- Ambientes de desarrollo (Shells).

1.4.10 Lenguajes de Programación

Un lenguaje de programación es un lenguaje artificial empleado para dictar instrucciones a una computadora, y aunque es factible hacer uso de cualquier lenguaje computacional para la producción de herramientas de inteligencia artificial; se han desarrollado herramientas cuyo propósito específico es el desarrollo de sistemas dotados con Inteligencia Artificial.

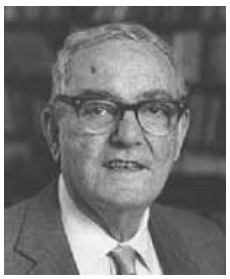
Algunos de los lenguajes más notables son:

IPL-11: Es considerado como el primer lenguaje de programación orientado a la solución de problemas de la Inteligencia Artificial. El IPL es el lenguaje de programación empleado por Newell y Simon para la construcción del GPS (General Solver Problem) (Newell y Simon ,1961). Este lenguaje de programación fue desarrollado en el año 1955 por Herbert Simon, el físico Allen Newell, y J.C. Shaw. Un año más tarde, estos tres científicos desarrollaron lo que sería el antecesor del GPS llamado “Logic Theorist”, el cual era capaz de demostrar una gran variedad de teoremas matemáticos. Este programa se considera como el primer programa creado con la intención de imitar la forma como el ser humano resuelve problemas.

Lisp: Su nombre se deriva de las siglas de LISt Processor. Este lenguaje de programación es uno de los lenguajes más antiguos que continúa siendo empleado; fue especificado por John McCarthy y sus colaboradores en el Instituto Tecnológico de Massachusetts en 1958. Entre otras de las aportaciones de este lenguaje, se puede mencionar la introducción de las estructuras de datos de árbol. Las listas encadenadas son una de las

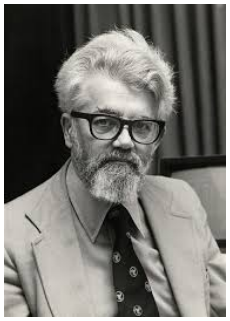
Figura 1.4: "Allen Newell"

Figura 1.5: "Herbert Simon"



estructuras de datos importantes del Lisp, y el código fuente del Lisp en sí mismo está compuesto de listas. Como resultado, los programas de Lisp pueden manipular el código fuente como una estructura de datos, dando lugar a los macro sistemas que permiten a los programadores crear una nueva sintaxis de lenguajes de programación de dominio específico empotrados en el Lisp. El LISP fue el primer lenguaje para procesamiento simbólico.

Figura 1.6: "John McCarthy"[1]



Prolog: Este lenguaje de programación debe su nombre a las siglas de PROgramming in LOGic (PROLOG). A diferencia de otros lenguajes, el Prolog no es un lenguaje de uso general; su uso es exclusivo para la solución de problemas relacionados con el cálculo de predicados, pues Alain Coulmeauer y Philippe Roussell estaban interesados en desarrollar una herramienta para realizar deducciones a partir de texto. La primera descripción detallada de este lenguaje fue en 1975 como manual para el intérprete de Prolog de Marseille (Roussel, 1975). Recientemente, en 1992, los autores del lenguaje escribieron un artículo titulado "El nacimiento de Prolog", en el que se muestra desde una amplia perspectiva el nacimiento de este lenguaje (Colmenauer & Roussel, 1992).

OPS5: Official Production System 5 (OPS5), es un lenguaje para ingeniería cognoscitiva que soporta la representación del conocimiento en forma de reglas. Aunque menos conocido que los otros lenguajes mencionados, este lenguaje de programación fue el pri-

Figura 1.7: “Philippe Roussel”[2]

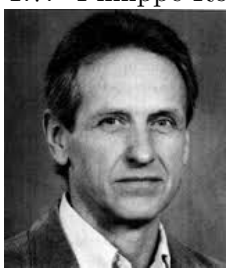


Figura 1.8: “Alain Colmenauer”

mero empleado con éxito en el desarrollo de sistemas expertos. La familia de lenguajes OPS(Oficial Productio System) fue desarrollada a finales de la década de 1970 por el Dr. Charles Forgy. El algoritmo inmerso en este lenguaje de programación es conocido como “Algoritmo Rete”, que es la base de muchos sistas actuales. Este algoritmo fue presentado como resultado de su tesis doctoral en 1979 (Forgy, 1979).

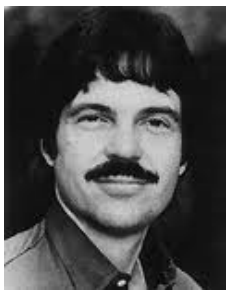
Figura 1.9: “ Dr Charles L. Forgy, creador de la familida de lenguajes OPS”

Small talk: Es el resultado de una investigación para la creación de un sistema informático orientado a la educación. El objetivo era crear un sistema que permitiese expandir la creatividad de sus usuarios, proporcionando un entorno para la experimentación, creación e investigación. Es un lenguaje orientado al trabajo con objetos lidereado por Alan Kay, quien tenía el objetivo de crear una “computadora personal” en toda la extensión de la palabra. El nacimiento del lenguaje se ubica con la tesis doctoral de Kay, como estudiante de la Universidad de Utah en 1969 (Kay, 1969).

Este lenguaje no solamente introdujo un entorno de desarrollo gràfico y amigable, sino que introdujo también el concepto de objetos, cambiando los paradigmas de programación originales. Aunque todo los programacdores compartimos ciertas costumbres y pasos generales para el desarrollo de aplicaciones, el manejo de Smalltalk es una experiencia completamente personal y cada individuo configura el ambiente y hace uso de las herramientas de manera muy diferente. Este lenguaje rompe con el ciclo de redacción/ compilación/ejecución, cambiándolo por un proceso interactivo y creativo.

“El propósito del proyecto Smalltalk es proveer soporte computacional al espíritu crea-

Figura 1.10: "Alan Kay, pionero en el desarrollo de la programación orientada a objetos"



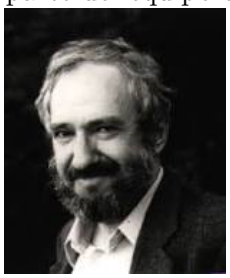
tivo, que anida en cada persona" (Ingalls,1981). Las ideas utilizadas en el desarrollo de Smalltalk son la base de la Programación Orientada a Objetos (POO) actual; aunque ésta fue popularizada varios años después. Smalltalk, también fue el pionero en el desarrollo de las interfaces gràficas de usuario (GUI) actuales.

Algunas de las versiones actuales de Smalltalk, en las que interviene Kay, son los proyectos de código abierto Squeak y Croquet.

Logo: Seymour Papert, un matemático y educador sudafricano que estuvo trabajando con el prominente educador Jean Piaget en la Universidad de Ginebra de 1959 a 1963, se mudó a los Estados Unidos de Norteamérica, donde conoció a Marvin Minsky (uno de los científicos más comprometidos con el desarrollo de la inteligencia artificial en ese momento) y con quien co-fundó el laboratorio de inteligencia artificial del MIT. Como fruto del trabajo de Papert en colaboración con de Bolt, Beranek y Newman, liderados por Wallace Feurzeig, se creó la primera versión de Logo en 1967.

Este lenguaje de programación esta basado en Lisp y ha implementado muchas de las ideas del conocido como construccionismo. Debido a su facilidad de aprendizaje este lenguaje de programación es una de las herramientas preferidas para el trabajo con niños y jóvenes.

Figura 1.11: "Saymour Papert, parte del equipo de desarrollo del lenguaje Logo."



De acuerdo a Harold Abelson, "Logo es el nombre de una filosofía de educación y de una familia en continua evolución de lenguajes de programación que contribuye a su realización". Uno de los objetivos de este lenguaje fue el de proveer una herramienta para la interacción de los seres humanos y las computadoras.

1.4.11 Aplicaciones y Sistemas Expertos

Desde su nacimiento, la inteligencia artifial ha vivido diferentes etapas; algunas de enorme motivación y abundancia de fondos para su investigación, y otras de poca credulidad

sobre sus logros; sin embargo, cuando parece que un camino se cierra, se abren muchos otros para seguir dando frutos. A continuación se comentan algunas de las aplicaciones más conocidas y naturalmente, se discuten también los más prominentes de los primeros sistemas expertos desarrollados; pues con ellos, la inteligencia artificial tuvo un merecido "renacimiento" cuando más parecía necesitarlo. De hecho, los sistemas expertos son actualmente considerados como uno de los productos típicos de la inteligencia artificial.

Algunas de las primeras investigaciones exitosas se realizaron en el área del lenguaje; un ejemplo de reconocimiento mundial es el programa conocido como "Eliza", desarrollado por el profesor Joseph Weizenbaum del Instituto de Tecnología de Massachusetts entre 1964 y 1966. Eliza captó la atención tanto de detractores como de defensores de la Inteligencia Artificial, pues fue uno de los primeros programas que procesaban lenguaje natural. Weizenbaum tenía el objetivo de crear un programa que mantuviera una conversación de texto coherente con un ser humano. Este célebre programa simula al psicólogo Carl Rogers, quien fue partícipe en el desarrollo de la terapia centrada en la persona. Weizembaum también es el autor del libro "Computer Power and Human Reason" (El poder de las computadoras y la razón humana), en el que expone algunas de sus inquietudes con respecto a la Inteligencia Artificial, pues a pesar de ser un área con enormes oportunidades, también podría ser empleada de manera que se perdieran algunas libertades civiles.

Figura 1.12: "Joseph Weizenbaum."

Otras áreas en las que se produjeron aplicaciones de enorme éxito, y que establecieron los principios para el almacenamiento y manipulación de las bases del conocimiento de los sistemas expertos, fueron las ciencias naturales. Un sistema experto es "una aplicación informática que soluciona problemas complicados que de otra manera exigirían ampliamente la pericia humana" (Rolston,1990). Los sistemas expertos simulan el proceso de razonamiento humanos mediante la aplicación de conocimiento e inferencia.

Uno de los primeros sistemas expertos conocido como "Dendral", desarrollado por Edward Feigenbaum en el Instituto de Tecnología de Carnegie (actual Instituto Carnegie de Tecnología). El trabajo de Feigenbaum se vió fuertemente influenciado por el trabajo de otros importantes investigadores como John Von Newman en un principio, y por Herbert Simon y Allen Newell posteriormente. En sus propias palabras Feigenbaum declaró que cuando Newell anunció el desarrollo de los primeros modelos de computadora sobre el pensamiento humano y la toma de desiciones a su clase, él renació; y a partir de ese momento decidió seguir el camino del estudio de los procesos mentales humanos (Henderson, 2007).

En lugar de abordar el proceso de toma de decisiones, Feigenbaum se dedicó al estudio de la memorización; desarrollando un programa conocido como EPAM (Elementary Perceiver and Memorizer). Una de las contribuciones más importantes del trabajo de Feigenbaum a la inteligencia artificial fue el desarrollo de las "redes de discriminación",

que posteriormente formaron parte de la investigación en “redes neuronales”.

A principios de los años 60, Feigenbaum estuvo trabajando en una aplicación concreta relacionada con un espectrómetro de masas, con el que se percató de la necesidad de construir una “base de conocimiento” para que los programas la pudieran usar. El resultado del trabajo de Feigenbaum y su colega Robert K. Lindsay en 1965, fue el desarrollo de “Dendral”; el primer sistema experto exitoso y productivo. Dendral mostro un enorme potencial en la deducción de información sobre estructuras químicas, que de acuerdo a su propio autor, se desprendía de su conocimiento sobre química. (Feigenbaum, 1990).

A pesar de la fuerte crítica que recibió en Dendral en su tiempo, pues algunos investigadores de la época lo consideraron demasiado especializado en química, al grado de que podían aprender poco de él; Feigenbaum no se dejó intimidar; en su lugar, formuló lo que él llamó “El principio del conocimiento”, el cual establece que en la ausencia del conocimiento, el razonamiento es inútil.

Figura 1.13: “Edward Feigenbaum”

Posteriormente, el equipo de Feigenbaum inició un nuevo proyecto que se extendería de 1972 a 1980 en la Universidad de Stanfort. Este sistema experto introdujo el uso de concimiento impreciso y la posibilidad de explicar el proceso de razonamiento de la herramienta. Aunque el proyecto fue inicialmente conducido por Feigenbaum, fue finalizado por Shortliffe y sus colaboradores mediante el uso de Lisp. La relevancia de este sistema radica en la demostración de la eficiencia de su esquema de representación del conocimiento y de sus técnicas de razonamiento; mismas que ejercieron una enorme influencia en el desarrollo de sistemas posteriores basados en reglas, en tanto dominios médicos como no médicos. El propósito de Mycin fue el diagnóstico de enfermedades infecciosas de la sangre.

Figura 1.14: “Edward Shortliffe”

La tabla 1.1 recopila algunos de los primeros sistemas expertos (Rolston, 1990). Los sistemas presentados en esta tabla nos permiten visualizar que alrededor de los años 70‘s se empezó a generar un enorme interés en lo que posteriormente se conoció como

sistemas basados en el conocimiento.

Tabla 1.1. Primeros sistemas expertos

Sistema	Fecha	Autor	Aplicación
Dendral	1965	Stanford	Deducción de información sobre estructuras químicas
Macsyma	1965	MIT	Análisis matemático complejo
HearSay	1965	Carnegie - Mellon	Interpreta en lenguaje natural un subconjunto del idioma
Mycin	1972	Stanford	Diagnóstico de enfermedades de la sangre
Tieresias	1972	Stanford	Herramienta para la transformación de conocimientos
Prospector	1972	Stanford	Exploración mineral y herramientas de identificación
Age	1973	Stanford	Herramienta para generar Sistemas Expertos
OPS5	1974	Carnegie - Mellon	Herramientas para desarrollo de Sistemas Expertos
Caduceus	1975	University of Pittsburg	Herramienta de diagnóstico para medicina interna
Rosie	1978	Rand	Herramienta de desarrollo de Sistemas Expertos
R1	1978	Carnegie - Mellon	Configurador de equipos de computación para DEC

1.4.12 Ambientes de desarrollo

Debido al enorme éxito que tuvieron los sistemas expertos en la década de los 80's, empezaron a surgir desarrollos conocidos como shells. En computación, una Shell es una pieza de software que provee una interfaz para los usuarios.

Un sistema experto posee entre sus componentes las dos herramientas siguientes:

1. Base de conocimientos. El conocimiento relacionado con un problema o fenómeno se codifica según una notación específica que incluye reglas, predicados, redes semánticas y objetos.
2. Motor de inferencia. Es el mecanismo que combina los hechos y las preguntas particulares, utilizando la base de conocimiento, seleccionando los datos y pasos apropiados para presentar los resultados

En el entorno de los sistemas expertos, una Shell es una herramienta diseñada para facilitar su desarrollo e implementación. En otros términos, una Shell es un "sistema experto" que posee una base de conocimientos vacía, pero acompañada de las herramientas necesarias para proveer la base de conocimiento sobre el dominio del discurso de una aplicación específica. Una Shell provee también al ingeniero del conocimiento (encargado de recabar la base de conocimientos) de una herramienta que trae integrado algún mecanismo de representación del conocimiento (ver capítulo 4), un mecanismo

de inferencia (ver capítulo 5), elementos que faciliten la explicación del procedimiento o decisión tomados por el sistema experto (componente explicativo) e incluso, algunas veces, proveen una interfaz de usuario.

El uso de estos ambientes de desarrollo, permiten el desarrollo de sistemas expertos eficientes incluso cuando no se domine algún lenguaje de programación; razón por la que se ha popularizado su uso para la producción de sistemas expertos en todos los dominios del conocimiento.

Algunos ambientes de desarrollo para la construcción de sistemas expertos independientes del dominio o shells clásicos son:

EMYCIN o Essential Mycin: Esta Shell fue construida en la Universidad de Stanford como resultado del desarrollo del sistema experto MYCIN que realizaba el diagnóstico de enfermedades infecciosas de la sangre. El sistema EMYCIN fue la base para la construcción de muchos otros sistemas expertos, tales como el PUFF (que diagnostica enfermedades pulmonares) y el SACON (que trabaja con ingeniería estructural).

OPS5- OPS83: Estas Shell fueron desarrolladas en C. C es el lenguaje que debe ser empleado para la inserción de las reglas base y junto con el uso del encadenamiento hacia adelante, la posibilidad de insertar las reglas en un lenguaje como el C son sus principales aportaciones. Este sistema esta basado en el algoritmo propietario RETE II. El algoritmo RETE es de reconocida eficacia para la comparación de hechos con los patrones de las reglas y la determinación de cuales de ellas satisfacen sus condiciones.

ESDE/VM (Expert System Development Environment). Esta es una herramienta comercial creada por IBM para sus sistemas operativos VM y MVS. Esta herramienta que incluye facilidades gráficas y acceso a bases de datos fue creada para su uso con mainframes. Esta herramienta usa reglas para representar el conocimiento del tipo IF x THEN y AND z. La principal ventaja de este tipo de representación es su facilidad para modificarlas y el hecho de que las reglas son un esquema muy semejante al modelo con el que los seres humanos razonamos o planteamos la solución a un problema.

KEE (Knowledge Engineering Environment): Esta herramienta de la casa comercial IntelliCorp combina el uso de la programación orientada a objetos y el uso de reglas como herramientas de representación del conocimiento. Debido a la mezcla de técnicas, esta Shell es conocida como una herramienta híbrida. El tipo b{asico de representación de KEE esta basada en marcos con jerarquías multipadre y varios mecanismos de herencias, que permiten transferir los atributos de los marcos a través de la jerarquía (Filman, 1992). Esta herramienta también puede codificar el conocimiento mediante LISP. KEE fue desarrollado mediante Common Lisp y usa reglas del tipo IF antecedente THEN consecuente.

S1: De acuerdo s Hayes (1992), esta herramienta para el desarrollo de sistemas expertos basada en reglas, provee toda clase de facilidades para la introducción del conocimiento tanto por parte del ingeniero como del experto sobre un cierto dominio. Este sistema incluye ya el uso del idioma inglés, de manera que traduce automáticamente fragmentos de texto que se asocien con la base de conocimientos. Este sistema además, muestra dinámicamente sus líneas de razonamiento alternativas, conclusiones y reglas heurísticas consideradas.

EXSYS: Considerado por largo tiempo como el líder para el desarrollo de sistemas expertos, es el resultado de más de 28 años de mejora y refinamiento. Esta herramienta

posee también elementos para la producción de sistemas expertos interactivos en la web. Exsys usa reglas del tipo IF/THEN que son resueltas con mecanismos de encadenamiento hacia adelante o hacia a tras de acuerdo al problema que se pretende resolver. En este sistema es factible que las reglas posean múltiples hechos en la parte del IF mediante el uso de conectores lógicos, mientras que la parte del THEN solamente presenta un componente. En este sistema las reglas se agrupan en una estructura arbórea que permite su mejor administración y comprensión.

Naturalmente existe una enorme lista de herramientas que no han sido comentadas, entre las que se pueden mencionar: MED1, NEXPERT OBJECT, GURU, HUGIN SYSTEM, ICARUS, entre otros.

1.4.13 Areas de la Inteligencia Artificial

En la actualidad existe una gran cantidad de areas especificas en las que se trabaja bajo el concepto de inteligenci artificial, cada una de estas cuenta con herramientas y/o tecnicas propias para lograr sus ofjetivos, algunas de estas se enlista en la siguiente tabla:

Tabla 1.2 de topicos de la Inteligencia Artificial

Mineria de Datos	Computación Evolutiva	Algoritmos Bio-inspirados
Reconocimiento de Imagenes	Reconocimiento de Patrones	IA Distribuidad y Sistemas Multiagentes
Sistemas Expertos y Sistemas Basados en Conocimiento	Representación y Administracion del Conocimiento	Procesamiento del Lenguaje Natural
Ontologias	Interfaces Inteligentes	Redes Neuronales
Lógica Difusa	Algoritmos Genéticos	Aprendizaje Máquina
Vida Artificial	Programación Lógica	Sistemas Hibridos Inteligentes
Sistemas Tutores Inteligentes	Razonamiento Basado en Casos	Realidad Aumentada
Programación Evolutiva	Optimización Multiobjetivo	Teoria de Automatas

1.5 ACTIVIDADES DE APRENDIZAJE

Después de haber haber concluido la lectura del capítulo “Introducción y Antecedentes de la Inteligencia Artificial”, responda a las siguientes preguntas:

1. En que lenguaje de programación se construyó el sistema conocido como “Logic Theorist”, cuyo propósito era el de resolver
2. ¿Quién fue Saymour Papert?
3. Mencione un evento de enorme interés para la Inteligencia Artificial que tuvo lugar en 1956.
4. ¿Quién es el creador del lenguaje conocido como Smalltalk?
5. ¿A quién se le atribuye el desarrollo de la familia de lenguajes OPS?
6. ¿Qué lenguaje de programación se empleó para el desarrollo de Mycin?
7. ¿Qué es Dendral?
8. ¿Cómo describirías al software ELIZA?

1.6 MATERIAL DE REFERENCIAS A CONSULTAR *OPCIONAL

1.6.1 LECTURAS ADICIONALES.

Libros:

Herken, R. "The Universal Turing Machine". Segunda Edición. Londres: Oxford University Press, 1988. (Incluye una descripción de la computadora universal de Turing).

McCorduck, Pamela. Machines Who Think. Natick, Mass.: A. K. Peters, 2004. (Revisión del trabajo de John McCarthy y otros investigadores pioneros de la Inteligencia Artificial)

Shasha, Dennis, and Cathy Lazere. Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists. New York: Springer-Verlag, 1995. (Incluye entre otros grandes pensadores de las ciencias de la computación, información de McCarthy y Feigenbaum).

Alonso, Angeles. Representación y manejo de información semántica heterogénea en interacción hombre-máquina. Tesis doctoral en Ciencias de la Computación. Instituto Politécnico Nacional. Noviembre, 2006. México.

Sitios web:

Hodges. A, "The Alan Turing Home Page." Disponible en línea. URL: http://www.turing.org.uk/turing/Turing.html Accesado en Diciembre de 2013.

Home Page de "John McCarthy." Disponible en línea. URL: http://www.formal.stanford.edu/jmc Accesado en Diciembre de 2013.

Home Page de "Edward Sshortliffe". Disponible en línea. URL: http://www.shortliffe.net/ Accesado en Diciembre de 2013.

1.6.2 REFERENCIAS

Asimov I. (1950). I, Robot. New York, NY: Gnome Press.

Colmenauer A, & Roussel P. "Birth of Prolog" 1992 ACM

ITAM (1987). Breve historia de la inteligencia artificial. URL: http://biblioteca.itam.mx/estudios/estudio/estudio10/sec_16.html, Recuperado en Noviembre de 2013.

McCarthy J. (1960). Recursive Functions of Symbolic Expressions and their Computation by Machine. Part 1. CACM. Vol. 3, No. 4, pp. 184-195.

A. Newell y H. A. Simon (1961). GPS, a program that simulates human thought, en E. Feigenbaum y J. Feldmann, Hrsg. (1995) Computers and Thought, ISBN 0262560925

Charles Forgy, "On the efficient implementation of production systems." Ph.D. Thesis, Carnegie-Mellon University, 1979.

Hayes. F. "Rule based systems". Encyclopedia of Artifitial Intelligence. Volume 2. John Wiley & Sons. Pp. 1417-1426. 1992.

Henderson H., Artificial Intelligence. Mirrors for the mind. Chelsea house publishers.

2007.

[Ingalls1981] INGALLS, Daniel H. H., Design Principles Behind Smalltalk, Byte Magazine, The McGraw-Hill Companies, agosto de 1981, Url: http://users.ipa.net/~dwighth/smalltalk/byte_aug81/design_principles_behind_smalltalk.html

[Kay1969] KAY, Alan C., THE REACTIVE ENGINE (Extractos), Url: http://www.mprove.de/diplom/gui/kay69.html

Pajares G. y Santos M. (2006). Inteligencia Artificial e Ingeniería del Conocimiento. Alfaomega. ISBN: 970-15-1166-2. Mexico.

Rusell & Norving (2003). Inteligencia Artificial. Un enfoque moderno, Segunda Edición. Prentice Hall.

(Roussel, 1975)

Turing A. (1950). Computing Machinery and Intelligence. Mind. Vol. 59, pp 433-460.

Rolston W. D. Principios de Inteligencia Artificial y Sistemas Expertos. Editorial McGraw Hill. ISBN 958-600-047-7, 1990.

Feigenbaum, Edward. "The Age of Intelligent Machines: Knowledge Processing—From File Servers to Knowledge Servers." KurzweilAI. net. Available online. URL: http://www.kurzweilai.net/the-age-of-intelligent-machines-knowledge-processing-from-file-serve
Visitado en Diciembre de 2013.

Filman. R.(1992) " KEE". Encyclopedia of Artificial Intelligence, Volume 1. John Wiley & Sons. Pp 718.

2 — Planteamiento del Problema

Fatima Sayuri Quezada Aguilera
Universidad Autónoma de Aguascalientes, Mexico

2.1 INTRODUCCION

El ser humano constantemente se hace preguntas referentes a las actividades que realiza cotidianamente como los juegos dominados por el azar, la forma en como se solucionan los problemas mediante el razonamiento, las labores en el trabajo, la solución de problemas numéricos, las relaciones interpersonales y sus efectos, el lenguaje y sus reglas, entre otros. Así mismo, también se pregunta sobre los fenómenos o sistemas que observa en el medio ambiente en que se desenvuelve, sobre si mismo y los seres que lo rodean.

Estas preguntas nos conlleva a un "problema" (o más) del cuál se desea encontrar la solución, si esta existe, y no sólo una solución sino la mejor solución que se pueda obtener e implementar tomando en cuenta todos los recursos disponibles. Un sistema informático es el producto final de la solución que se proponga, siendo así la síntesis del problema.

Es por esto que el hombre lleva a cabo dos tareas o procesos principales: Analizar y Sintetizar (Abstraer). La primera es para comprender, identificar elementos y medir el fenómeno, actividad o sistema bajo estudio. Seguido de esto viene la síntesis, que permite abstraer todos o la mayoría de los elementos y detalles más relevantes (desde la perspectiva de quien o quienes llevan a cabo esta tarea) en forma de especificaciones. En esta última tarea generalmente existen varios niveles de abstracción, los que pueden variar de a cuerdo al problema y el nivel de detalle al que se desee o se pueda llegar.

Estos problemas los encontramos en todas las áreas de la ciencia y de la vida del ser humano, por lo cual se puede decir que hay una cantidad infinita de problemas a resolver. Ahora bien, ¿Es posible resolver todos estos problema? ¿Se cuenta con las herramientas matemáticas o de otro tipo para resolverlos?

Los investigadores del área de la Inteligencia Artificial han desarrollado herramientas para resolver cierto tipo de problemas que con otros enfoque clásicos ha sido difícil encontrar la mejor solución, obteniendo muy buenos resultados.

Las búsquedas que mediante agentes inteligentes realizan el procedimiento con o sin información, para llegar a la solución, son una buena forma de resolver problemas usando las herramientas de la Inteligencia Artificial.

Para hacer el uso adecuado de las técnicas de la Inteligencia Artificial, es importante definir si el problema puede ser resuelto mediante las mismas y plantear el problema de

la manera adecuada.

2.2 Clasificación de los problemas

Los problemas se clasifican según la complejidad para resolverlos tanto en tiempo como en espacio, ya que son los recursos críticos en la solución de los diferentes problemas.

Por su complejidad para resolver los problemas, se definen dos clases principales que son:

P- Viene del inglés “Polinomial”, que quiere decir que el algoritmo requiere un tiempo polinomial o polinómico en función del tamaño de la instancia del problema para resolverlo.

NP – Del inglés “Non deterministic Polinomial”, que significa que el algoritmo requiere un tiempo que tiende a ser exponencial en función del tamaño de la instancia del problema para resolverlo.

En la clase NP, existe otra clases de problemas llamados NP- Completos (NPC), por lo cual muchos científicos creen que P NP. Los problemas NP – Completos tienen la característica de que si se encuentra un algoritmo polinomial para resolver cualquiera de los problemas NP- Completos, entonces toda esa clase de problemas puede ser resuelta por un algoritmo polinomial. Hasta el momento no se ha encontrado ningún algoritmo polinomial para resolver estos problemas.

A continuación se muestran gráficamente las relaciones que hay entre las clases de problemas P y NP y las clases P, NP y NPC.

Figura 2.1: Esta es la relación que más científicos creen que existe entre estas clases de problemas

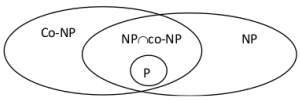
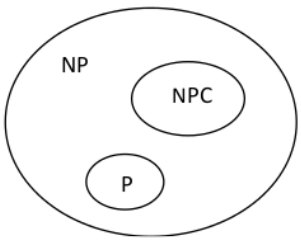


Figura 2.2: Esta es la relación que muchos investigadores creen que hay entre estas clases de problemas



En el libro “” (Garey y Johnson) los autores dedican un apéndice al compendio de los problemas NP y los cuales son agrupados de a cuerdo a sus características. Estos son problemas base o tipo, es decir, son una generalización en la cual se define formalmente (matemáticamente) el problema, considerando las variables y las restricciones además de otros elementos. De esta manera, si encontramos un problema en cualquier área de la ciencia y/o en la vida real que se plantee como cualquiera de estos problemas, entonces

se tendrá una idea más clara de que tipo de herramienta o técnica se deberá usar para solucionarlo.

2.3 ¿Cómo plantear un problema?

El planteamiento del problema inicia con la descripción del mismo en lenguaje natural, donde se establece la meta (solución que se quiere obtener), se delimita el problema y los elementos que intervienen como los son las variables, restricciones, función objetivo o a optimizar, punto de partida, todo esto de manera un tanto informal. Aquí es donde se identifica el nivel de conocimiento que se tiene del problema.

Luego de obtener la descripción en lenguaje natural, se sigue un proceso de definición formal y estructurada que se representará en algún lenguaje, lo cual nos lleva a otro nivel de abstracción o síntesis. En este nivel es necesario definir las entidades que van a interactuar y eliminar ciertos detalles que se consideren no relevantes. Aquí es donde se puede determinar que parte de la descripción del problema puede ser validada y representada simbólicamente. Parte del proceso requiere que se determinen las entradas, las salidas y el proceso (cálculos y acciones) que realizará cada entidad de cada nivel y la forma en como se comunicaran entre ellas.

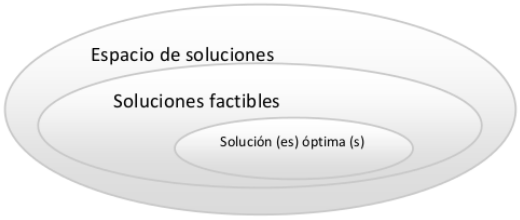
De ser posible, es importante formalizar aspectos del problema como:

- Función objetivo o a optimizar
- Variables / Parámetros que intervienen
- Restricciones

La forma de la solución. Si sólo se desea obtener un valor final o la serie de pasos para llegar a ella.

En este proceso de formalización del problema a resolver, es importante verificar si este es similar a los problemas tipo ya definidos en la literatura, como los problemas NP, de los cuales ya hay una definición muy específica, pudiendo así concentrarse en los aspectos como ciertas restricciones que puedan variar un poco con respecto al problema tipo (pero que no lo hacen diferente del problema a resolver).

Figura 2.3: Representación del Espació de Soluciones



También se debe determinar el espacio de soluciones, de las cuales se van a considerar sólo la soluciones factibles, es decir las soluciones que cumplan con las restricciones previamente establecidas, de las cuales una o más pueden ser la solución optima que resuelve el problema. Además de lo anterior, se requiere la información o datos con los que se trabajará, que representarán la instancia del problema a resolver.

Una vez determinado esto, se podrá elegir la estrategia a seguir para obtener la solución del problema, que en este caso será la búsqueda mediante agentes.

2.4 Planteamiento del problema para ser resuelto mediante la búsqueda

Como se mencionó, la solución de un problema se puede llevar a cabo mediante métodos de búsqueda, para lo cuál se requiere abstraer los elementos, las entidades y la forma en que van a ser manipulados por un agente que se conocerá como solucionador del problema.

- Función objetivo o a optimizar
- Variables / Parámetros que intervienen
- Restricciones

La forma de la solución. Si sólo se desea obtener un valor final o la serie de pasos para llegar a ella.

Tabla 2.1 Elementos que se deben identificar de un problema

Elementos para solucionar un problema	**Elementos para solucionar un problema mediante búsqueda**
Función Objetivo	Meta
Variables / Parámetros	Elementos
Valores de la variables	Valores / Estados
Restricciones	Cambiar de un estado a otro
Forma de la solución	Solución
Instancia del problema	Datos
Estrategia	Algoritmo de Búsqueda
Representación del problema de a cuerdo a la estrategia seleccionada	Grafo

El proceso de identificación – representación/abstracción – obtención de datos - estrategia y solución del problema, puede ser llevado a cabo por entes que tengan un nivel de inteligencia suficiente para llevar a cabo dicho proceso, por lo cual es un tema básico en la Inteligencia Artificial. Este proceso de resolución de problemas requiere de un conjunto de datos iniciales que utilizando un conjunto de procedimientos seleccionados a priori es capaz de determinar el conjunto de pasos o elementos que nos llevan a lo que denominaremos una solución.

Cada problema que se pueda plantear es diferente, sin embargo, todos tienen ciertas características comunes que nos permiten en primera instancia clasificarlos y estructurarlos. En segunda instancia esta estructuración, la cual se debe llevar a cabo de una manera formal, nos puede permitir resolverlos de manera automática, utilizando la representación adecuada. Dicha representación por necesidad de uniformidad y estandarización deberá utilizar un lenguaje común para ser utilizado por los entes inteligentes.

Si abstraemos los elementos que describen un problema podemos identificar elementos como:

1. Un punto de partida, los elementos que definen las características del problema.
2. Un objetivo a alcanzar, qué queremos obtener con la resolución.
3. Acciones a nuestra disposición para resolver el problema, de qué herramientas disponemos para manipular los elementos del problema.
4. Restricciones sobre el objetivo, qué características debe tener la solución

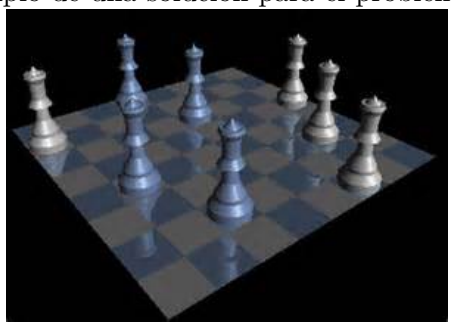
5. Elementos que son relevantes en el problema definidos por el dominio concreto, que conocimiento tenemos que nos puede ayudar a resolver el problema de una manera eficiente.

Espacio de estados: Se trata de la aproximación más general, un problema se divide en un conjunto de pasos de resolución que enlazan los elementos iniciales con los elementos que describen la solución, donde en cada paso podemos ver como se transforma el problema.

El conjunto de estados que el agente debe recorrer, generalmente se representa mediante un grafo, aunque en algunos casos concretos se pueden utilizar árboles. Donde cada nodo del grafo representará a uno de los estados.

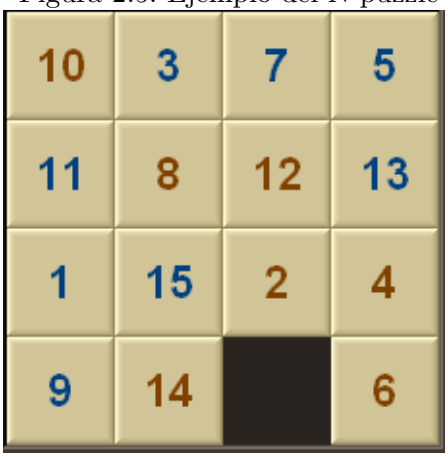
El problema de las N reinas es un problema clásico, en este caso el problema se trata de colocar N reinas en un tablero de ajedrez de N $\times$ N de manera que no se maten entre si. En este problema no nos interesa la manera de hallar la solución, sino el estado final. En la figura tenemos representada la solución para un problema con dimensión 8:

Figura 2.4: Ejemplo de una solución para el problema de las 8-Reinas



- Espacio de estados: Configuraciones de 0 a n reinas en el tablero con sólo una por fila y columna
- Estado inicial: Configuración sin reinas en el tablero
- Estado final: Configuración N reinas en el tablero en la que ninguna reina se mata entre si.
- Operadores: Colocar una reina en una fila y columna
- Condiciones: La reina no es matada por ninguna otra ya colocada
- Transformación: Colocar una reina más en el tablero en una fila y columna determinada
- Solución: Una solución, pero no nos importan los pasos

Figura 2.5: Ejemplo del N-puzzle



El N-puzzle es un problema clásico que consiste en un tablero de M posiciones dispuesto como una matriz de N × N en el que hay ((NxN) -1) posiciones ocupadas por fichas numeradas de manera consecutiva empezando del uno y una posición vacía. Las fichas se pueden mover ocupando la posición vacía, si la tienen como adyacente. El objetivo es partir de una disposición cualquiera de las fichas, para obtener una disposición de éstas en un orden específico. Tenemos una representación del problema en la siguiente figura:

La definición de los elementos del problema para plantearlo en el espacio de estados sería la siguiente:

Espacio de estados: Configuraciones de 8 fichas en el tablero
Estado inicial: Cualquier configuración
Estado final: Fichas en un orden específico
Operadores: Mover el hueco Condiciones: El movimiento está dentro del tablero
Transformación: Intercambio entre el hueco y la ficha en la posición del movimiento
Solución: Que movimientos hay que hacer para llegar al estado final, posiblemente nos interesa la solución con el menor número de pasos

2.5 Bibliografía

C. Papadimitriou and K. Steiglitz. Combinatorial Optimization: Algorithms and Complexity. Dover Publications, 1998.
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Introductiion to Algorithms. The MIT Press, 2000.
Steven S. Skiena. The Algorithm Design Manual. Springer. Segunda Edición. 2008.
Stuart J. Russell, Peter Norvig. Inteligencia Artificial Un enfoque Moderno. Pearson Prentice Hall. Segunda Edición. 2004

3 — Representación del Conocimiento

Antonio Silva Sprock y Ember Martínez Flor
Universidad Central de Venezuela, Venezuela
Universidad de ¿?,Colombia

3.1 INTRODUCCION

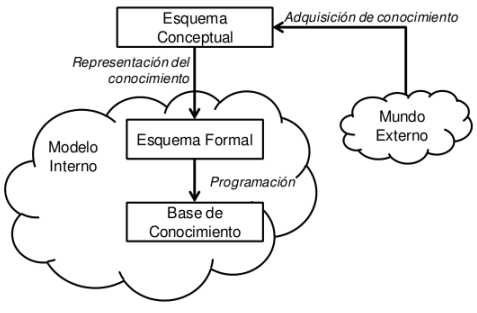
La mayoría de los programas y softwares que trabajan con IA, manipulan símbolos y piezas de información sobre el mundo, representado a partir de los conocimientos adquiridos, provenientes de múltiples fuentes de conocimiento, como: documentos, expertos, etc.

Para que la máquina se comporte de manera inteligente debe poseer conocimiento almacenado en una Base de Conocimiento y debe ser capaz de utilizarlo, es decir, no basta con introducir el conocimiento en la máquina o en el software, hay que proveer unos mecanismos que le permitan razonar con el conocimiento previamente almacenado y esto es expresar de una manera formal conceptos y las relaciones surgidas.

Lograr tal razonamiento sobre una Base de Conocimiento implica la necesidad de plasmarlos según algún modelo de representación, que en cualquier caso debe ser aquel que mejor se adecúe a los conocimientos adquiridos. Posteriormente, se concreta la elección particular de la herramienta y se formaliza de acuerdo a ella.

El área de la IA que estudia cómo representar el conocimiento adquirido del mundo en una computadora, se llama Representación del Conocimiento, y a las distintas formas, estructuras o técnicas que permiten expresar el esquema conceptual como un esquema formal, se denominan Formalismos o Técnicas de Representación de Conocimiento. La figura 3.1 muestra el proceso modelización del Mundo Externo hacia una Base de Conocimiento.

Figura 3.1: proceso modelización del Mundo Externo hacia una Base de Conocimiento



3.2 CARACTERÍSTICAS DESEABLES DE LOS FORMALISMOS DE REPRESENTACIÓN DE CONOCIMIENTO.

Rich y Hnight (1994) indican que un buen sistema de representación de conocimiento en un dominio particular debe poseer características de suficiencia y de eficiencia, como son:
Suficiencia de la representación: la capacidad de representar todos los tipos de conocimiento necesarios en el dominio.
Suficiencia deductiva: la capacidad para manipular las estructuras de la representación con el fin de obtener nuevas estructuras que se correspondan con un nuevo conocimiento deducido a partir del antiguo.
Eficiencia deductiva: capacidad de incorporar información adicional en las estructuras de conocimiento con el fin de que los mecanismos de inferencia puedan seguir las direcciones más prometedoras.
Eficiencia en la adquisición: la capacidad de adquirir nueva información con facilidad.
Como se observa, las características mencionadas son aún deseables, ya que no puede encontrarse un formalismo que garantice tales características, sin embargo si se puede exigir que un buen formalismo posea:
Exactitud: referida a si el modelo utilizado es lo suficientemente fiel a la realidad como para no distorsionarla.
Adecuación: referida al poder expresivo y la eficacia del formalismo, la primera trata de todo lo que se puede decir con la notación, es decir, si la representación permite realizar o evitar distinciones sutiles. La eficacia se refiere a las estructura concretas que se utilizan y al impacto de dichas estructuras en las operaciones del sistema.
Estas últimas exigencias permiten poder escoger de un conjunto heterogéneo de formalismos imperfectos, que logran adecuarse en alguna medida al conocimiento del dominio que se desea representar.

3.3 TIPOS DE CONOCIMIENTO

La técnica o formalismo de representación de conocimiento depende en gran medida del conocimiento. Los distintos formalismos permiten representar cualquier conocimiento del mundo exterior, sin embargo, unos formalismos son más adecuados que otros y esta adecuación depende del tipo de conocimiento.

Conocimiento Declarativo versus Conocimiento Procedimental

Rich y Hnight (1994) llaman al conocimiento declarativo, conocimiento relacional simple, al afirmar que pueden almacenarse en estructuras del mismo tipo que las utilizadas en los sistemas de bases de datos. Igualmente afirman que es simple por la escasa capacidad deductiva que ofrece. La tabla 3.1 muestra un ejemplo de conocimiento descriptivo.

Figura 3.2: tabla almacenando conocimiento descriptivo

Estudiante	Fecha de Nacimiento	Calificaci n	Preferencia
Jos L pez	18/09/1992	17	Base de Datos
Mar a Herrera	24/06/1996	15	Ingenier a de Software
Carmen Mar n	01/03/1994	16	Tecnolog a de Internet
Mart n Urrutia	30/08/1993	14	Inteligencia Artificial

Por otra parte, existe otro conocimiento operacional o procedimental, que especifica que hacer cuando se da una determinada situación, por ejemplo pensemos en la necesidad de seleccionar el mejor estudiante por cada opción de la carrera, considerando la preferencia y la calificación, o incluir como criterio al más joven, donde requerimos algunos elementos más que los almacenados estáticamente mostrados en la tabla 1.

Conocimiento heredable

Una de las formas más útiles de inferencia en la herencia de propiedades, donde los elementos de una clase heredad los atributos y los valores de otras clases más generales en las que están incluidos.

3.4 TÉCNICAS DE REPRESENTACIÓN DE CONOCIMIENTOS

Las diversas técnicas de representación de conocimientos se pueden agrupar en tres familias, dependiendo de la adecuación del formalismo para representar conceptos, relaciones o acciones:

3.4.1 Formalismos basados en conceptos

Representan las principales entidades del dominio, así como los posibles valores que pueda tomar cada propiedad. Los formalismos basados en conceptos más utilizados son: Objeto-Atributo-Valor y los Marcos.

Marcos: propuestos por Marvin Minsky en 1975. Es una estructura de datos que representa situaciones estereotipadas, que posee gran capacidad para representar aspectos semánticos sobre el conocimiento declarativo del dominio. Los Marcos están formada por un nombre y un conjunto de propiedades llamadas ranuras o nichos, en inglés Slots. Cada nicho tiene unas propiedades, que reciben el nombre de facetas, estas facetas describen el tipo de valores que puede tomar. Los marcos se relacionan con otros marcos, de esta forma se puede establecer una jerarquía entre ellos.

En la práctica, los conceptos Marcos y Objetos tienden a asemejarse mucho según las utilidades que presentan; en algunas herramientas Shell, ambos términos representan lo mismo.

Composición de los marcos:

- Tipos: marcos clases y marcos instanciados. Ver Figura 3.2.
- Relaciones: estándares (subclase e instancia y superclase y elementos de la clase) (ver figura 3.3), no estándares (fraternal, disjunto, no disjunto y cualquier otra “ad hoc” (ver figura 3.4). Las relaciones permiten expresar las dependencias entre los conceptos y permite desarrollar un sistema basado en Marcos.
- Propiedades: de clase (propias), de instancia (miembros)
- Ranuras: contiene al concepto de propiedad y de relación
- Facetas por tipo de conocimiento: facetas declarativas (tipo, cardinalidad, multivaluado, factor de certeza, valores permitidos, valores por omisión), facetas procedimentales (precondición, si necesito, si modifico, si añado, si borro) (ver figura 3.5).
- Facetas por tipo de propiedades: facetas de clase y de instancia (tipo, cardinalidad, multivaluado, factor de certeza), de Instancia (valores permitidos, valores por omisión, precondición, si necesito, si modifico, si añado, si borro).

Figura 3.3: jerarquía de marcos mostrando la jerarquía

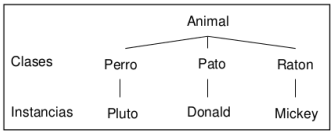


Figura 3.4: ejemplo de relaciones estándares

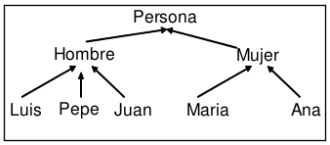


Figura 3.5: ejemplo de relaciones estándares

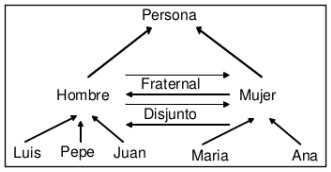


Figura 3.6: ejemplo de propiedades

Figura 3.7: ejemplo del Marco llamado Hombre

Marco Hombre		Nexo	Valor Permitido	Valor por defecto	Valor por omisi n	Condici n	Si necesito	Si modifico	Si a ado
Relaciones	Subclase	persona							
	fraternal	mujer							
	casado	mujer							
Prop. Propias	Nombre								
Prop.Miembros									

3.5 Formalismos basados en relaciones

Capaces de representar las relaciones existentes entre entidades y conceptos. Los más importantes son: Lógica, Redes semánticas y la Teoría de Dependencia Conceptual y las Ontologías.

Lógica: estos formalismos pueden ser la Lógica Proposicional y Lógica de Predicados o de Primer Orden.

Lógica Proposicional: una proposición es un enunciado, frase o expresión que tiene un significado determinado y que mediante un criterio es susceptible de ser calificada como verdadero o falso. Son ejemplos de proposiciones los

siguientes enunciados:

- Brasil es campeón mundial de futbol.
- Colombia es un país cafetero.
- El calor dilata los cuerpos.
- El sol es un planeta

No toda frase o expresión es una proposición por ejemplo:

- No cruces la calle si hay mucho tráfico
- ¿Cuál es su número telefónico?
- ¿Qué hora es?
- ¡Estoy triste!

La "Brasil es campeón mundial de futbol" representa una proposición porque puede ser calificada de verdadera o falsa, en este caso particular Brasil ganó los mundiales del año 1958, 1962, 1970, 1994 y 2002 por la que puede ser calificada como verdadera, mientras que la frase "No cruces la calle si hay mucho tráfico" no es proposición porque es una oración imperativa que expresa una sugerencia o consejo, a la cual no se le puede asignar calificación de verdadera o falsa.

Las proposiciones son los elementos u objetos básicos en la lógica de proposiciones, donde son representadas por letras minúsculas del alfabeto p,q,r,s,t , etc. Cada una de estas letras recibe el nombre de símbolos proposicionales o átomos.

Una expresión formada por un solo símbolo proposicionales se llama proposición atómica o simple. Las proposiciones compuestas relaciona dos o más proposiciones por medio de un conector lógico.

Las proposiciones "El sol es un planeta" y "El calor dilata los cuerpos" son proposiciones simples, mientras que expresiones como "el número 15 es divisible por 3 y por 5" y "Colombia es un país cafetero y bananero" son proposiciones compuesta debido a que relacionan dos o más proposiciones. En el primer caso se pueden identificar la proposición "el número 15 es divisible por 3" y "el número 15 es divisible por 5" unidas por una conjunción (conector lógico y).

En lógica proposicional las sentencias o fórmulas bien formadas se construyen usando los símbolos proposicionales (p, q, r, s, t, etc), conectores lógicos y símbolos auxiliares. Los símbolos primitivos en lógica de proposiciones son:

Símbolos: "p","q","r","s",...

Conectores lógicos: conjunción ($\wedge$), disyunción($\vee$), negación($\neg$), implicación o condicional($\rightarrow$), bicondicional($\leftrightarrow$).

Símbolos auxiliares: "(","(")","[","]","{","}"

Una expresión o fórmula bien formada cumple con los siguientes criterios:

- Todos los símbolos proposicionales son una formula bien formada.
- Si A es una expresión bien formada, la negación de la expresión es una formula bien formada ($\neg$ A)
- Si A y B son expresiones bien formadas entonces, la expresión formada por la composición de A y B a través de un conector lógico es una expresión bien formada.
- A es una expresión bien formada si es resultado de la aplicación de las reglas anteriores un número finito de veces.

Por ejemplo, si se consideran las siguientes proposiciones:

- p: La Inflación es alta

- q: El tasa de desempleo es alta
- r: El índice de pobreza crece

Se pueden construir las siguientes formulas bien formadas:

- p ∧ q → $r\neg(p\wedge$ q) ¬r

Dado los valores de verdad de las proposiciones se puede obtener el valor de verdad de una expresión o formula bien formada, utilizando la semántica de los conectivos lógicos, como se muestra a continuación

P	q	p ∧ q	p ∨ q	¬ p	p → q	p ↔ q
V	V	V	V	F	V	V
V	F	F	V	F	F	F
F	V	F	V	V	V	F
F	F	F	F	V	V	V

En la lógica proposicional se puede determinar el valor de verdad de una proposición sin necesidad de tener en cuenta la semántica de las de expresión, es suficiente con conocer su estructura. Por ejemplo se puede calcular el valor de verdad para la expresión pq pq de la siguiente forma:

P	Q	p ∧ q	p ∨ q	p ∧ q → p ∨ q
V	V	V	V	V
V	F	F	V	V
F	V	F	V	V
F	F	F	F	V

Lógica de Predicados: en el lenguaje natural existen argumentos que son intuitivamente válidos, pero cuya validez no puede ser probada por la lógica proposicional. Esto se debe a la falta de un mecanismo que le permita indagar la estructura interna de las variables proposicionales, generalizar y diferenciar un objeto de una propiedad. Por ejemplo, considérese las siguientes expresiones:

- Las personas que practican algún deporte son más saludables
- Toda persona es de sexo masculino o femenino.
- Algunos alumnos aprobaran la materia.
- Las ciudades cálidas son más apetecidas por los turistas.

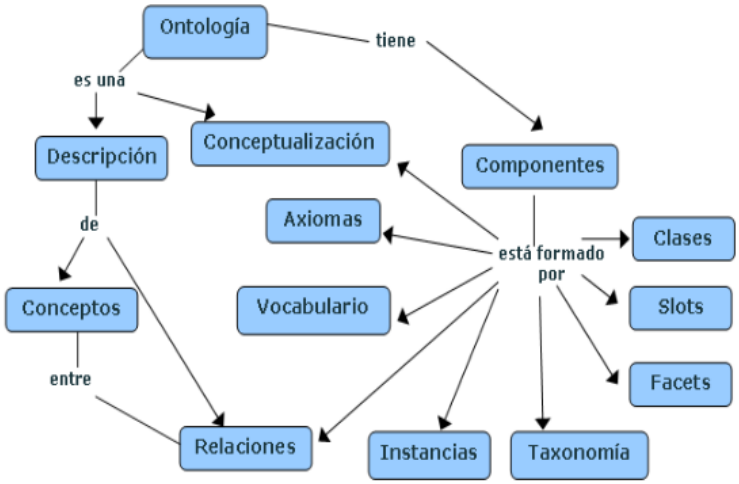
La lógica de predicados permite enunciar algo sobre los objetos, es decir definir atributos y relaciones entre elementos a través de objetos, atributos y relaciones se puede expresar prácticamente cualquier tipo de expresión, inclusive como las listadas anteriormente.

Redes Semánticas: son grafos orientados formados por nodos y por arcos unidireccionales, los nodos representan los conceptos, objetos, atributos, características o situaciones y los arcos entre los nodos, representan las relaciones entre los conceptos.

Los arcos pueden ser estructurales, donde la semántica es independiente del conocimiento que se está representado y pueden ser de: generalización (relaciona una clase con otra más general y permite formar una red de nodos por especialización de conceptos. Ejemplo: Subclase_de, es_un), de instanciación (relaciona un objeto concreto con su tipo o clase genérica. Ejemplo: Instancia_de, es_un), de agregación (relaciona un objeto o concepto con sus componentes. Ejemplo: Tiene_parte). Los arcos descriptivos pueden ser

de propiedades (relaciona un objeto con alguna propiedad específica. Ejemplo: marca_carro, calificación, sabe_cantar) y los arcos relacionadas a otras relaciones (permiten definir otras relaciones no estructurales. Ejemplo: esposo_de, hermano_de). Los arcos descriptivos sirven para describir los conceptos mediante atributos propios del dominio. La figura 3.8 muestra una Red Semántica.

Figura 3.8: ejemplo de Red Semántica



Ontologías: entre muchas definiciones (Ceccaroni, 2001; Gómez-Pérez, Fernández-López y Corcho, 2004; Tramullas, 1999), la ofrecida por (Gruber, 1993), es la más aceptada: "Es una especificación formal y explícita de una conceptualización compartida".
Las Ontologías son:

- Formales: por ser una organización teórica de términos y relaciones usados como herramienta para el análisis de los conceptos de un dominio.
- Compartidas: por capturar conocimiento consensual que es aceptado por una comunidad.
- Explícitas: por estar referida a la especificación de los conceptos y a las restricciones sobre estos.

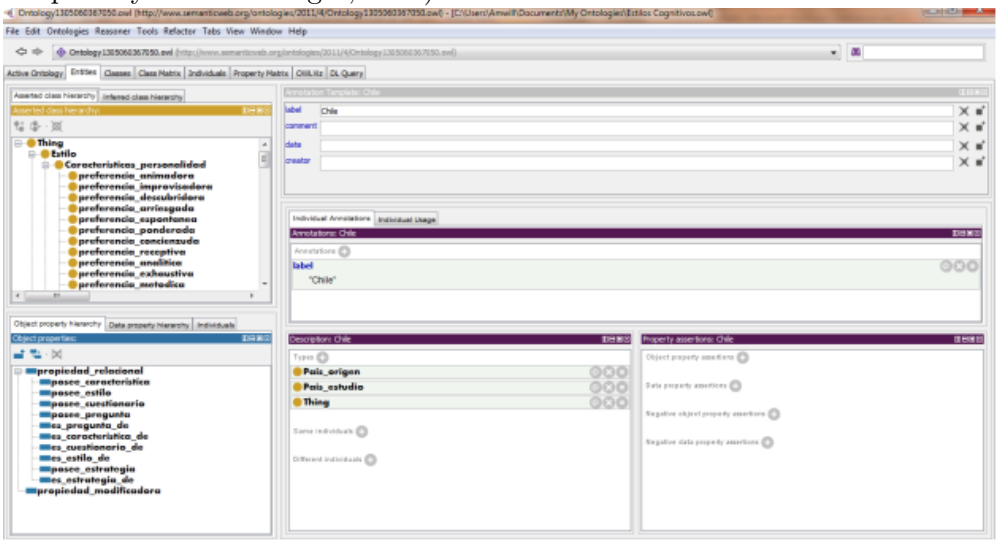
Y agregando más sentido a la definición, se puede adicionar que una ontología debe ser legible y libre de ambigüedades.
Componentes de las Ontologías: varían de acuerdo al dominio de interés y a las necesidades de los desarrolladores. (Lozano, 2002; Gómez-Pérez y otros, 2004), indican que por lo general entre los componentes se encuentran los siguientes:

- Clases: las clases son la base de la descripción del conocimiento en las ontologías ya que describen los conceptos (ideas básicas que se intentan formalizar) del dominio. Las clases usualmente se organizan en taxonomías a las que por lo general se les aplican mecanismos de herencia.
- Relaciones: representan las interacciones entre los conceptos del dominio. Las ontologías por lo general contienen relaciones binarias; el primer argumento de la relación se conoce como el dominio y el segundo como el rango.

- Funciones: son un tipo concreto de relación donde se identifica un elemento mediante el cálculo de una función que considera varios elementos de una ontología.
- Instancias: representan objetos determinados de un concepto.
- Taxonomía: conjunto de conceptos organizados jerárquicamente. Las taxonomías definen las relaciones entre los conceptos pero no los atributos de éstos.
- Axiomas: usados para modelar sentencias que son siempre ciertas. Los axiomas permiten, junto con la herencia de conceptos, inferir conocimiento que no esté indicado explícitamente en la taxonomía de conceptos. Los axiomas definidos en una ontología pueden ser estructurales o no estructurales: un axioma estructural establece condiciones relacionadas con la jerarquía de la ontología, conceptos y atributos definidos; un axioma no estructural establece relaciones entre atributos de un concepto y son específicos de un dominio. Los axiomas se utilizan también para verificar la consistencia de la ontología.
- Propiedades (Slots): son las características o atributos que describen a los conceptos. Las especificaciones, rangos y restricciones sobre los valores de las propiedades se denominan "facets". Para un concepto dado, las propiedades y las restricciones sobre éstos son heredadas por las subclases y las instancias de la clase.

La figura 3.9 muestra una implementación de una Ontología de Estilos de Aprendizaje, en el Software de desarrollo de Ontologías Protégé.

Figura 3.9: Implementación en Ptotege, de una Ontología de Estilos de Aprendizaje (Silva Sprock y Ponce Gallegos, 2013).



Clasificación de las Ontologías: las ontologías se pueden clasificar tomando en cuenta diferentes criterios, como por ejemplo, la intención de uso o tarea en particular, la formalidad del lenguaje utilizado para su construcción, la generalidad, entre otros. Sin embargo, no existe una taxonomía estándar de las ontologías, que permitan manejar una clasificación exacta de las ontologías, es decir múltiples autores presentan múltiples clasificaciones (Gómez-Pérez y otros, 2004; Uschold y Grüninger, 1996; FIPA, 2000).
Uschold (Uschold y Grüninger, 1996) presenta tres dimensiones sobre las cua-

les varían los tipos de ontologías:

- Formalidad: para referirse al grado de formalismo del lenguaje usado para expresar la conceptualización.
- Propósito: para referirse a la intención de uso de la ontología.
- Materia: para expresar la naturaleza de los objetos que la ontología caracteriza.

Ontología según su Formalidad: los tipos de ontologías según el grado de formalidad del lenguaje usado, son los siguientes:

- Ontología altamente informal: expresada en lenguaje natural (Glosario de términos).
- Ontología informal estructurada: utiliza lenguaje natural estructurado y restringido, que permite reducción de la ambigüedad.
- Ontología semi-formal: usa un lenguaje de definición formal, como ontolingua.
- Ontología rigurosamente formal: la definición de términos se lleva a cabo de manera meticulosa usando semántica formal y teoremas TOVE (Toronto Virtual Enterprise).

Ontología según su Propósito: los tipos de ontologías según el propósito o uso que se les vaya a dar son las siguientes:

- Ontologías para comunicación entre personas: una ontología informal puede ser suficiente.
- Ontologías para inter-operabilidad entre sistemas: para llevar a cabo traducciones entre diferentes métodos, lenguajes, software, etc. En estos casos la ontología se usa como un formato de intercambio de conocimiento.
- Ontologías para beneficiar la ingeniería de sistemas; en cuanto a:
- Reusabilidad: la ontología es la base para una codificación formal de entidades importantes, atributos, procesos y sus interrelaciones en el dominio de interés. Esta representación formal podría ser un componente reusable y/o compartido en un sistema de software.
- Adquisición de Conocimiento: la velocidad y la confiabilidad podría ser incrementada usando una ontología existente como punto de partida y como base para la adquisición de conocimiento cuando se construye un sistema basado en conocimiento.
- Fiabilidad: una representación formal hace posible que la automatización del chequeo de consistencia resulte en software más confiable.
- Especificación: la ontología puede asistir al proceso de identificación de requerimientos y definición de una especificación para sistemas de tecnologías de información.

Ontología según su Materia: según los objetos o problemas que se caractericen en las ontologías, éstas pueden ser:

- Ontologías de dominio: caracterizan objetos específicos, tales como medicina, finanzas, química, biología, etc.
- Ontologías para resolver problemas: conceptualizan el problema o tarea a resolver en un dominio.
- Meta-Ontologías: el objeto que se caracteriza es un lenguaje de representación de conocimiento.

3.6 Formalismos basados en acciones

Estos formalismos permiten representar los conocimientos del dominio mediante acciones básicas. Los más importantes son: Reglas de Producción y los Guiones.

Reglas de Producción: la representación basada en reglas parte del principio que la relación lógica entre un conjunto de objetos pueden ser representadas mediante una regla, cada regla tiene la forma "si P entonces Q", donde P es la es el antecedente, premisa, condición o situación de la regla y Q es el consecuente, conclusión, acción o respuesta. El antecedente y la conclusión de la regla son expresiones lógicas que contiene una o más proposiciones simples o compuestas.

Una proposición como "Si el fuego es producido por un líquido combustible inflamable entonces incendio es de clase B", puede ser simbolizada en lógica de la siguiente forma:

P $\rightarrow Q$, si fuego producido por un líquido combustible inflamable, entonces incendio es de clase b

$\forall$ x (F(x) $\rightarrow$ C(b)), para todo x, si x es fuego producido por un líquido combustible inflamable, entonces fuego clase b.

Algunas representaciones imponen restricciones a las reglas, por ejemplo, el antecedente debe ser una expresión lógica que solo admiten conectores conjuntivos y el consecuente solo admite expresiones lógicas simples. Estas restricciones se toman para facilitar la manipulación y procesamiento computacional. También se aplican este tipo de restricciones debido a que se puede remplazar cualquier expresión lógica a través de las reglas de sustitución o equivalencia sin pérdida de generalidad.

Otra forma más general de estructurar Reglas de Producción es formando el antecedente, de la forma SI, y un consecuente, de la forma ENTONCES, cuando los hechos descritos en el antecedente son ciertos se tienen los hechos que forman el consecuente.

- **Si** persona_corre **entonces** persona_suda
- **Si** persona_suda **entonces** persona_se_deshidrata
- **Si** persona_se_deshidrata **entonces** bebe_agua

3.7 Referencias

Ceccaroni, L. (2001). Onto WEDSS-An ontology-Based environmental decisión-support system for the management of wastewater treatment plants. Tesis Doctoral. Universidad Politécnica de Cataluña, España.

Corcho, O., Fernández-López, M., Gómez-Pérez, A. y López, A. (2005). Building legal ontologies with METHONTOLOGY and WebODE. Law and the Semantic Web. Legal Ontologies, Methodologies, Legal Information Retrieval, and Applications. Springer-Verlag, LNAI 3369. Marzo de 2005.

Foundation for Intelligent Physical Agents. (2000). FIPA Ontology Service Specification. Número de documento: XC00086C. Descargado el 10 de julio de 2008 de http://www.fipa.org/specs/fipa00086/XC00086C.pdf.

Gómez-Pérez, A., Fernández-López, M. y Corcho, O. (2004). Ontological Engineering, Springer Verlag, 2004.

Gruber, T. (1993). Toward Principles for the Design of Ontologies Used for Knowledge Sharing. Available as Technical Report KSL 93-04, Knowledge Systems Laboratory, Stanford University. Descargado el 10 de Julio de 2008 de http://citeseer.ist.psu.edu/gruber93toward.html.

Lozano, A. (2002). Métrica de idoneidad de ontologías. Tesis Doctoral. Escuela Politécnica de Cáceres. Departamento de Informática. Universidad de Extremadura. España. ISBN: 84-7723-537-6.

Rich, E. y Knight, K. Inteligencia artificial (segunda edición) (McGraw-Hill Interamericana, 1994).

Silva Sprock, A., Ponce Gallegos, J. (2013). Reingeniería de una Ontología de Estilos de Aprendizaje para la Creación de Objetos de Aprendizaje. Revista de Tecnología de Información y Comunicación en Educación Eduweb. ISSN: 1856-7576. Vol 7 Nº 2. Julio-Diciembre 2013.

Tramullas, J. (1999). Agentes y ontologías para el tratamiento de información: clasificación y recuperación en Internet. Actas del IV Congreso ISKO España. 22-24 abril de 1999: Granada. p.247-252.

Uschold, M. y Grüninger, M. (1996). Ontologies: Principles, Methods and Applications. AIAI-TR-191. Knowledge Engineering Review. Vol 11 Number 2.

4 — Agentes Inteligentes

Ana Casali
Universidad Nacional de Rosario, Argentina

Dentro de la IA un área que ha ganado relevancia en las últimas dos décadas es el desarrollo de agentes. Esto ha surgido por un lado, por la necesidad de abordar problemas complejos que son muchas veces difíciles de tratar por un sistema monolítico o que son de naturaleza distribuida, ya sea porque los datos o los procesos están distribuidos. Por otro lado, hay muchos autores donde consideran que ciertos comportamientos inteligentes más que ser modelados como conductas complejas individuales, pueden surgir de la interacción de conductas simples. Esto ha llevado a revisar muchas de las lineas principales de la IA con este enfoque orientado a agentes, como es el reconocido libro de Inteligencia Artificial de Russell & Norvig (2009).
Dada la relevancia del tema, en este libro se incorpora este capítulo para dar una introducción a los Agentes Inteligentes. Luego de dar algunas motivaciones a la noción de agencia, en primer lugar se presenta el enfoque individual de agente. En esta dirección se plantean un modelo teórico y distintas arquitecturas, haciendo énfasis en las que se basan en los sistemas intencionales, las cuales consideran que las acciones que realizan los agentes son consecuencia de ciertos "estados mentales" como sus creencias (información que posee de él y de su entorno), deseos (objetivos ideales) e intenciones (objetivos que decide alcanzar). Luego, se presenta una introducción al lenguaje JASON como ejemplo de un lenguaje que permite implementar agentes concretos de razonamiento práctico (sistemas PRS). En la última sección se presentan los tópicos relevantes a los aspectos sociales de los agentes en los sistemas multiagentes, como las características a considerar para que la performance de estos agentes en sociedad sea adecuada. Se plantea la necesidad de un protocolo de comunicación e interacción y alguna de las formas de interacción más relevantes, destacando la negociación. Se acompaña este capítulo con la bibliografía que se ha citado en las distintas secciones y que servirá al lector para ampliar y profundizar la cobertura de los temas que se han desarrollado.

4.1 Introducción

En los últimos años se ha incrementado el diseño e implementación de sistemas multiagentes (MAS) para abordar el desarrollo de sistemas distribuidos complejos. Estos sistemas están compuestos por agentes autónomos que interactúan entre sí. Aplicaciones orientadas a los agentes se han utilizado en distintos dominios como el comercio electrónico, sistemas de gestión de pro-

cesos distribuidos, sistemas de control de tráfico, etc. En gran parte, esto se debe a que el paradigma de sistemas multiagentes ofrece un conjunto de conceptos y técnicas para modelar, diseñar e implementar sistemas distribuidos complejos.
Hay distintas razones que sostienen la importancia de un enfoque de IA distribuida: muchos de los problemas están físicamente distribuidos, el mundo está compuesto por entidades autónomas que interactúan entre sí y con el entorno. Uno de los principales objetivos de este enfoque de la IA es resolver problemas que son muy complejos para abordarlos con un sistema monolítico y además, entender los principios subyacentes al comportamiento de múltiples entidades del mundo denominadas agentes, sus interacciones y cómo producen un comportamiento general del sistema multiagente (MAS).
La noción de agencia surge como ilustra la Figura, del aporte y confluencia de distintas lineas de trabajo. Como el eje principal de la Inteligencia Artificial Distribuida, pero también puede considerarse como un nuevo enfoque de la Ingenieria de Software Orientada a Agentes (AOSE: Agent Oriented Software Engineering). Por otra parte, puede verse como una evolución de la Programación Orientada a Objetos, donde estos objetos van a tener mayor autonomía decidiendo qué método ejecutar y cuándo hacerlo. Todo estos enfoques han podido concretarse a su vez, dado los avances del soporte que le da el área de Sistemas Distribuidos y Redes.

Cuando se hace referencia a la noción de agente surgen dos ámbitos de trabajo a similitud con lo que sucede a nivel humano: el agente en sí mismo (aspectos personales) y los conjuntos de agentes (aspectos sociales). En muchos escenarios, los agentes que conforman el sistema necesitan interactuar de diferentes maneras con el fin de cumplir sus metas o mejorar su desempeño. El objetivo de estas interacciones puede ser de diferente naturaleza: intercambiar información u objetos, coordinar acciones, negociar, colaborar, competir, etc. En este capítulo se tratarán principalmente los aspectos individuales de la agencia y luego se verá como los agentes conforman los llamados sistemas multiagentes.

4.2 Qué es un agente?

Un agente es una entidad física o virtual que posee ciertas características generales: es capaz de percibir el entorno y tener una representación parcial del mismo; es capaz de actuar sobre el entorno; puede comunicarse con los

otros agentes (pueden ser humanos o no) que comparten su hábitat; tiene un conjunto de objetivos que gobiernan su comportamiento y posee recursos propios. Para ver que elementos distinguen a un agente de software (softbot) de otro tipo de programa presentaremos algunas definiciones.
Según Norvig & Russel (Russel & Norvig 2009) "*un agente inteligente es aquél que puede percibir su ambiente mediante sensores y actuar sobre ese mundo mediante efectores (o actuadores)*". Estos autores definen como meta de la IA: diseñar un agente inteligente o racional que opere o actúe adecuadamente en sus ambientes. Esto significa que el agente debe hacer siempre lo correcto de acuerdo a sus percepciones y que emprenderá la mejor acción posible en una situación dada. La racionalidad dependerá de: la secuencia de percepciones –todo lo que el agente ha percibido hasta ahora; la medida de éxito elegida; cuánto conoce el agente del ambiente en que opera y las acciones que el agente esté en condiciones de realizar.
Un ejemplo presentado en (Russel & Norvig 2009) es considerar un Taxi con piloto automático (taximetrero reemplazado por un agente inteligente), en este ejemplo podemos distinguir los siguientes elementos en relación al agente:

- *Percepciones:* video, acelerómetro, instrumental del tablero, sensores del motor.
- *Metas:* seguridad, llegar a destino, maximizar ganancias, obedecer las leyes, satisfacción del cliente.
- *Ambiente:* calles urbanas, avenidas, tráfico, peatones, clima, tipo de cliente.

Si ahora se considera un agente recomendador de turismo en la Web, en este caso se tendría:

- *Percepciones:* sitios Web visitados por ususarios, consultas realizadas, compra de pasajes y servicios, etc.
- *Metas:* vender productos turísticos, promover un destino, satifacer al usuario etc.
- *Ambiente:* la Web.

Wooldridge & Jennings (1995) definen a un agente como: "*un sistema de computación situado en algún entorno, que es capaz de una acción autónoma y flexible para alcanzar sus objetivos de diseño.*" De esta definición se desprende que es un sistema de software (hardware) con las siguientes propiedades fundamentales:

- **Autonomía** (actuar sin intervención, control): Un sistema autónomo es capaz de actuar independientemente, exhibiendo control sobre su estado interno.

- **Habilidad Social** (lenguaje de comunicación): es la capacidad de interacción con otros agentes (posiblemente humanos) a través de algún tipo de *lenguaje de comunicación de agentes.*

- **Reactividad** (percepción-acción): La mayoría de los entornos interesantes son dinámicos. Un sistema reactivo es aquél que mantiene una interacción continua con el entorno y responde a los cambios que se producen en él, en tiempo de respuesta adecuado.

- **Proactividad** (dirigido a la meta, toma iniciativa): Generalmente se espera que un agente haga cosas para nosotros. Un sistema proactivo es aquél que genera e intenta alcanzar metas, no es dirigido sólo por eventos, toma iniciativa y reconoce oportunidades.

4.3 Modelos abstractos de agentes

Siguiendo a (Weiss, 1999; Wooldridge, 2009) se puede formalizar a continuación, un modelo abstracto de agente que con ciertas variantes puede modelizar distintos tipos de agentes. Primero se asume que el entorno del agente puede caracterizarse por un conjunto numerable de estados del entorno: $S = \{s_1, s_2, ...\}$. Las acciones que el agente puede realizar están representada por el siguiente conjunto de acciones $A = \{a_1, a_2, ...\}$. Luego un agente estandar puede definirse mediante una función que mapea las secuencias de estados a acciones:
$acción : S^* \rightarrow A$
Intuitivamente puede verse que este tipo de agente decide que acción realizar en base a su historia (experiencias pasadas). El comportamiento no-determinista del entorno puede modelarse mediante la función:
$env : S \times A \rightarrow \wp(P)$
que toma el estado actual del entorno s y una acción a que ejecuta el agente, y los mapea a un conjunto de estados del entorno *env(s,a).* Si todos estos conjuntos tienen un sólo estado, el entorno es *determinista.*
Se puede representar la interacción del Agente con su entorno mediante su *historia*, la cual se define como una secuencia *estado-acción-estado*, a partir del estado inicial S_0:
$h : S_0 \rightarrow^{a_0} S_1 \rightarrow^{a_1} ... \rightarrow^{a_{n-1}} S_n$
Dos agentes serán equivalentes respecto a un entorno env, si sus historias son iguales.
Hay agentes que no toman sus decisiones respecto a su historia, sino que se basan sólo en el estado actual del entorno. Estos agentes suelen denominarse *puramente reactivos* y pueden representarse mediante una función que va de cada estado del entorno a una acción:
$acción : S \rightarrow A$
Un ejemplo simple de este tipo de agente es un termostato de un tanque al que debe mantenerse en cierta temperatura T, si la temperatura percibida es menor a T el agente activará el calentador, en caso contrario, no hará nada. Este modelo abstracto de agente no nos conduce a su implementación ya que no se especifica como desarrollar la función acción. Se deberá considerar un refinamiento de este modelo y tomar decisiones de diseño para poder desarrollar agentes concretos.
Otra aspecto que hay que tener en cuenta, es la habilidad del agente de obtener información de su entorno, esto se puede representar mediante la función *ver,* que va de los estados del entorno al conjunto P de percepciones:
$ver : S \rightarrow P$
En los agentes que tengan alguna estructura interna de datos, *agentes basados en estados*, se considera que la decisión de la acción a tomar, dependerá al menos parcialmente, de estos estados. Sea I el conjunto de los posibles estados internos, la función selección de la acción, puede definirse para estos agentes de la siguiente forma:

$acción : I \rightarrow A$
Además, otra función *next* mapeará las transiciones de un estado y las percepciones recibidas, a un nuevo estado:
$next : I \times P \rightarrow I$
Se pueden citar ejemplos de agentes triviales como un Termostato o demonio en UNIX (biff). Actualmente hay muchas implementaciones de agentes inteligentes entre las cuales podemos mencionar agentes planificadores de vuelos, de interfaz, recomendadores, agentes para comercio electrónico, entre otros.
Russel & Norvig (2009) presentan distintos tipos básicos de agentes, que concuerdan con estos modelos planteados:
(i) agentes reactivos (de actitud refleja) y (ii) agentes con estado interno (informados de lo que pasa). También plantean modelos de razonamiemto más complejo, pero que tienen un comportamiento más flexible: (iii) agentes basados en Metas o en una medida de Utilidad.
Wooldridge & Jennings (1995) presentan también una noción más fuerte de agente, donde además de las propiedades de un agente mencionadas anteriormente, se agregan nociones mentales como: Actitudes de información (Conocimiento, Creencias, etc.) y pro-actitudes (Deseos, Intenciones, Obligaciones, etc.). Esta noción de agente se verá representada claramente en algunas de las arquitecturas que se describirán en la próxima sección.

4.4 Arquitecturas de Agentes

Con el fin de dar a los sistemas multiagente un soporte formal, se han propuesto diversas teorías y arquitecturas de agentes. Las teorías de agentes son esencialmente especificaciones del comportamiento de los agentes expresadas como propiedades que los agentes deben satisfacer. Una representación formal de estas propiedades ayuda al diseñador a razonar sobre el comportamiento esperado del sistema. Las arquitecturas de agentes, representan un punto medio entre la especificación y la implementación. En ellas se identifican las principales funciones que determinan el comportamiento del agente y se definen las interdependencias que existen entre ellas. Luego esas arquitecturas se implementarán en un lenguaje de programación adecuado, con el fin de desarrollar agentes concretos para la aplicación deseada.
Las teorías del agente basadas en un enfoque intencional son las más utilizadas y estudiadas. Estos enfoques se basan en una psicología popular mediante el cual se predice y se explica el comportamiento humano a través de la atribución de actitudes. Por ejemplo, a la hora de explicar la conducta humana, a menudo es útil y común hacer declaraciones como la siguiente:

- *Jorge tomó su abrigo porque creía que iba a ser frío.*
- *Pedro trabajó duro porque quería recibirse.*

En estos ejemplos, los comportamientos de Jorge y Pedro se pueden explicar en términos de sus actitudes, como *creer y querer*. Se ha utilizado el término de sistema intencional para describir entidades cuyo comportamiento puede ser predicho por el método de atribuir ciertas actitudes mentales tales como creencia, deseo y la visión racional (Dennet, 1987).
Cuando el proceso del sistema subyacente es bien conocido y comprendido, no hay razón para tomar una postura intencional, pero no es lo que ocurre en muchas aplicaciones. Las nociones intencionales son herramientas de abstracción, que proporcionan una forma familiar de describir, explicar y predecir

el comportamiento de un sistema complejo. Considerando que un agente es un sistema que está convenientemente descrito por un sistema intencional, vale la pena evaluar cuáles son las actitudes apropiadas para representar a un agente. Las dos categorías más importantes son las *actitudes de información* (por ej., conocimiento y creencias) y *pro-actitudes* (deseos, intenciones, la obligación, etc.).
Las actitudes de información están relacionadas con el conocimiento o creencias que el agente tiene sobre el mundo, mientras que las pro-actitudes son las que de alguna manera orientan las acciones de los agentes. Las actitudes de ambas categorías están estrechamente relacionadas y gran parte del trabajo de las teorías de agentes consiste en esclarecer las relaciones entre ellas. Parece razonable que un agente debe estar representado en términos de al menos, una actitud de información y una pro-actitud. Se necesita un marco lógico capaz de establecer cómo los atributos de agencia están relacionados, cómo se actualiza el estado cognitivo de un agente, cómo afecta el ambiente a las creencias del agente y, cómo la información y pro-actitudes del agente lo conducen a realizar acciones.
Considerando ahora el área de arquitecturas de agentes, este campo representa el paso desde la especificación hasta la implementación. Se abordan cuestiones tales como las siguientes: ¿Cómo se va a construir sistemas informáticos que satisfagan las propiedades especificadas por una particular teoría de agentes? ¿Cómo el agente se puede descomponer en la construcción de un conjunto de módulos componentes y cómo estos módulos deben interactuar? ¿Qué estructuras de software / hardware son apropiadas? Maes (1991) definió una arquitectura de agentes como: "*Una metodología específica para la construcción de los agentes. En él se especifica la forma en la que el agente se puede descomponer en la construcción de un conjunto de módulos componentes y cómo estos módulos deben interactuar. El conjunto de los módulos y sus interacciones tiene que dar una respuesta a la cuestión de cómo los datos de los sensores y el estado mental actual del agente determinan las acciones ... y el futuro estado mental del agente.* "

4.4.1 Distintas Arquitecturas de Agentes

Existen diferentes propuestas de clasificación de las distintas arquitecturas de agentes, siguiendo (Wooldridge, 2009) se presentan los lineamientos de cuatro clases de arquitecturas, las cuales no son ni exaustivas ni excluyentes y luego, se detallan la arquitectura PRS y los modelos de agentes BDI.

- **Arquitecturas basadas en la lógica - Agentes deductivos**

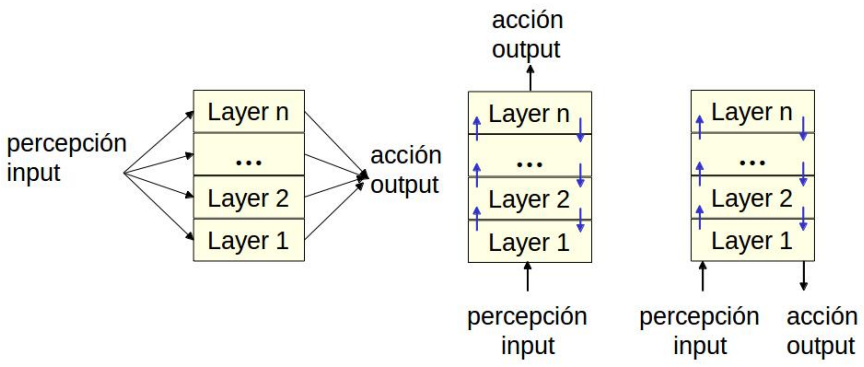
Una forma clásica de construir agentes sigue la linea tradicional de IA de desarrollar sistemas inteligentes basados en una representación simbólica de su entorno y de sus deseos, que permita realizar una manipulación de esta representación. Estos agentes toman decisiones a través del razonamiento lógico, basado en la coincidencia de patrones y la manipulación simbólica. La idea de desarrollar agentes deductivos como demostradores de teoremas es atractiva y se ha implementado en algunos agentes sin embargo, hay algunos problemas que deben resolverse, por ejemplo: cómo trasladar el mundo real a una descripción simbólica certera y adecuada, en tiempo adecuado y la complejidad computacional de los demostradores de teoremas, lo que pone en duda la efectividad de estos mecanismos en entornos de tiempos restringidos.

- **Arquitecturas reactivas - Agentes reactivos**

Estas arquitecturas plantean diferentes alternativas al paradigma simbólico y proponen una arquitectura que actúa siguiendo un enfoque conductista, con un modelo estímulo-respuesta. Las arquitecturas reactivas no tienen un modelo del mundo simbólico como elemento central de la representación del conocimiento y no utilizan razonamiento simbólico complejo. La inteligencia y el comportamiento racional son el producto de la interacción que el agente mantiene con el medio y la conducta inteligente emerge de la interacción de varios comportamientos simples. Una de las arquitecturas alternativas más conocida es la arquitectura de subsunción (Brooks [23]).

- **Arquitecturas en capas - Agentes híbridos**

Muchos investigadores han argumentado que ni un enfoque completamente reactivo o deliberativo es adecuado para la construcción de agentes. Teniendo en cuenta el requisito de que un agente debe ser capaz de un comportamiento reactivo y proactivo, un enfoque interesante consiste en la creación de subsistemas separados para hacer frente a estos tipos de comportamientos. Una clase de arquitecturas en las que los subsistemas definidos se organizan en capas jerárquicas y que interactúan, implementa esta idea. En este enfoque, un agente se define en términos de dos o más capas, para hacer frente a las conductas reactivas y proactivas, respectivamente. Ejemplos de las arquitecturas de capas son la Arquitectura de Ferguson TouringMachines de capas horizontales donde todas las capas tienen acceso a los datos del entorno y a realizar acciones en el entorno (Ferguson, 1992), y de Muller InteRRaP capas verticales de dos pasadas, donde una capa tiene el acceso a los datos del entorno y a realizar acciones en él (Muller, 1997)



(a) Capas Horizontales (b) Capas Verticales (control one-pass) (c) Capas Verticales (control two-pass)

- **Arquitecturas de razonamiento práctico - Agentes PRS y Agentes BDI**

Es un modelo de agente que decide momento a momento que acción seguir en orden a satisfacer los objetivos. Esta arquitectura incluye dos importan-

tes procesos: *deliberación* -decide en cada caso que objetivos perseguir- y *razonamiento de medios-fines* (means-ends) -determina la forma en que se alcanzarán dichos objetivos. Luego de la generación de los objetivos el agente debe elegir uno y comprometerse a alcanzarlo. Los objetivos que decide alcanzar son sus intenciones, las cuales juegan un papel crucial en el proceso de razonamiento práctico.
La arquitectura BDI (Belief, Desire, Intention) es una de las más relevantes dentro de esta línea y fue originada por el trabajo Rao & George (Rao & Georgeff 1995). Esta arquitectura está caracterizada porque los agentes están dotados de los estados mentales que representan las Creencias, Deseos e Intenciones, incluyendo las correspondientes estructuras de datos:
- *Creencias (Beliefs)*: representan el conocimiento que el agente tiene sobre sí mismo y sobre el entorno.
- *Deseos (Desires)*: son los objetivos que el agente desea cumplir.
- *Intenciones (Intentions)*: se puede considerar como un subconjunto de deseos consistentes entre sí que el agente decide alcanzar. Las intenciones derivan en las acciones que ejecutará el agente en cada momento.
El proceso de un agente BDI se ilustra en el esquema presentado en la siguiente Figura (Weiss, 1999). En dicha figura además de los tres componentes principales del modelo (B, D e I) aparecen otros componentes que se describen a continuación:

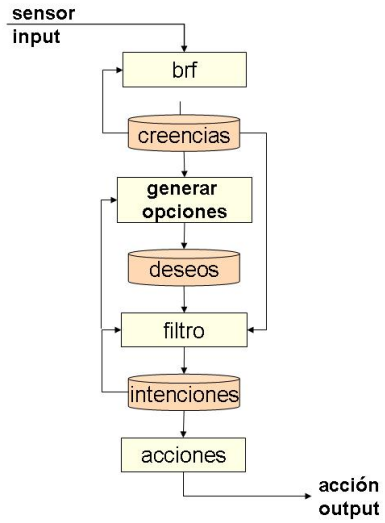
- Un conjunto de creencias actuales: Bel
- Una función de revisión de creencias (brf), que toma la entrada sensada y el estado actual de creencias del agente para determinar un nuevo conjunto de creencias: (donde $\wp$ representa el conjunto de partes de un conjunto):
  $brf : \wp(Bel) \times P \rightarrow \wp(Bel)$
- Una función generadora de opciones (generate options), que determina qué opciones están disponibles para el agente (sus deseos), en base a sus creencias e intenciones actuales:
  $opciones : \wp(Bel) \times \wp(Int) \rightarrow \wp(Des)$
- Una función de filtrado (filter), que representa el proceso de deliberación del agente en la que el agente determina sus intenciones, basado en sus actuales creencias, deseos e intenciones:
  $filtrado : \wp(Bel) \times \wp(Des) \times \wp(Int) \rightarrow \wp(Int)$
- Una función selectora de acciones (actions), la cual determina en cada caso qué acción llevar a cabo en base a las intenciones vigentes:
  $ejecutar : \wp(Int) \rightarrow A$

El proceso resultante de un agente BDI es una función acción: P $\rightarrow$ A, la cuál para cada percepción (elemento del conjunto P) decide una acción (del conjunto A), y puede definirse por el siguiente pseudocódigo:
function accion (p:P):A
begin

- *B:= brf(B,p)*
- *D:= opciones(B,I)*
- *I:= filtrado(B,D,I)*

return ejecutar(I)
end function accion

Figura 4.1: Diagrama de una arquitectura genérica de un agente BDI



Dentro de los modelos de agentes, el modelo BDI es uno de los más difundidos y estudiados y hay varias razones que contribuyen a su importancia. El lector puede ver un análisis de su evolución en (Georgeff et al, 1999). Por un lado, esta arquitectura está provista de componentes esenciales que le permiten abordar aplicaciones del complejo mundo real ya que suelen estar situados en entornos dinámicos e inciertos. Por otra parte, el modelo BDI es también interesante porque se ha realizado sobre él un gran esfuerzo para su formalización a través de las lógicas BDI (Rao&Georgef, 1995). Además, la importancia del modelo no se suscribe al marco teórico, sino que se han presentado diferentes plataformas que implementan esta arquitectura. Entre las implementaciones podemos mencionar por ejemplo a IRMA (Intelligent Resourcebounded Machine Architecture)(Bratman et al., 1988), dMARS(distribute Multi-Agent Reasoning System) (D'Inverno et al., 1998), entre otras.
En el modelo BDI, sus distintas variantes y muchas de sus implementaciones están construidas sobre lógicas bi-valuadas. Estos presentan limitaciones al momento de representar mundos inciertos, ya que carece de una forma de representar una estimación acerca de la incertidumbre (posibilidad, probabilidad, etc.) de que el mundo planteado sea el real. Esto ha llevado a (Casali et al., 2011) a proponer en un modelo graduado que mejore la performance del agente utilizando grados en las actitudes mentales (B,D,I).

4.4.2 Agentes de Razonamiento Procedural (PRS)

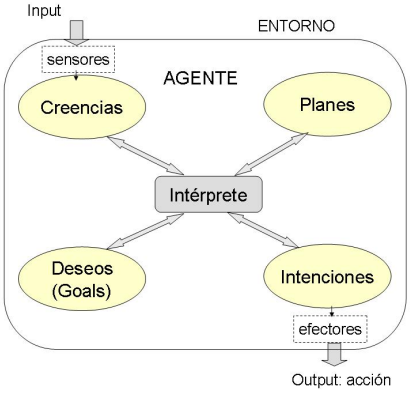
La arquitectura PRS –Procedural Reasoning Systems– (Georgeff&Lansky, 1987) fue quizás la primera arquitectura basada en el paradigma BDI. Esta se ha convertido en una de las más conocidas y ha sido utilizada en distintas aplicaciones. Un agente con arquitectura PRS trata de alcanzar cualquier meta que se proponga basándose en sus creencias sobre el mundo (entorno). También puede simultáneamente reaccionar ante la ocurrencia de algún nuevo evento. De esta forma, en un agente PRS los comportamientos de tipo dirigido

por los objetivos (*goal-directed)* y dirigido por los eventos (*event-directed)* pueden ser fácilmente integrados. Estos sistemas consisten en un conjunto de herramientas y métodos, para la representación y ejecución de planes. Estos planes o procedimientos son secuencias condicionales de acciones, las cuales pueden ejecutarse para alcanzar ciertos objetivos, o reaccionar en situaciones particulares. Una arquitectura PRS consiste básicamente de los siguientes componentes:

- *base de creencias* (beliefs) actuales sobre su entorno, la cual es actualizada para reflejar cambios;
- *conjunto de objetivos o deseo*s actuales (goals) a alcanzar;
- *librería de planes o procedimientos*, que describen secuencias particulares de acciones y pruebas que pueden realizarse para alcanzar ciertos objetivos o para reaccionar ante ciertas situaciones; y
- *estructura de intenciones*, que consiste de un conjunto ordenado (parcialmente) de todos los planes elegidos para ejecución.

Un intérprete (mecanismo de inferencia) manipula estos componentes como se ilustra en la Figura.

Figura 4.2: Arquitectura de un agente PRS



En cada ciclo de este intérprete: recibe nuevos eventos y objetivos internos; selecciona un plan (procedimiento) apropiado teniendo en cuenta los nuevos eventos, objetivos y creencias; ubica el plan dentro de la estructura de intenciones (grafo); elije un plan (intención) en la estructura y finalmente ejecuta un paso del plan activo. Esto puede resultar en una acción primitiva o en la formulación de un nuevo objetivo. El sistema interactúa con el entorno a través de su base de conocimientos y de las acciones básicas o primitivas. Las estructuras básicas de un agente PRS pueden tener distintas representaciones, dependiendo de la plataforma elegida. En general, la lógica utilizada para la representación del conocimiento es muy similar para todas las implementaciones PRS.

Lenguajes de Agentes

Luego de que se ha definido una arquitectura de agente, de cara a su implementación surgen las siguientes preguntas: Cómo deben programarse estos agentes? Cuáles deben ser las primitivas para esta tarea? Cómo es posible hacer que estos lenguajes provean un marco efectivo? Se han planteado distintos lenguajes que permite programar sistemas computacionales en

términos de los de conceptos desarrollados en los distimtos modelos formales de agentes. Si bien se puede programar agentes utilizando lenguajes de programación de propósito general, podemos destacar los siguientes lenguajes orientados a agentes: Agent0 (Shoham, 1990), Concurrent METATEM (Fisher, 1994), de la familia PRS hay una variedad de lenguajes entre los que se pueden mencionar: JASON (http://jason.sourceforge.net/), JACK (http://aosgrp.com/products/jack/), OPENPRS y la familia APL -2APL, 3APL- (http://apapl.sourceforge.net/). En la próxima sección se presentarán los componentes principales de JASON como ejemplo de lenguaje para especificar agentes.

4.5 Un Lenguaje para Desarrollar Agentes: Introducción a JASON

En esta sección se hará una reseña de los conceptos más importantes de la herramienta JASON (http://jason.sourceforge.net/) como un ejemplo de plataforma que permite diseñar agentes PRS. Si bien JASON puede usarse mediante su integración con sistemas multiagentes como por ejemplo, utilizando JADE (http://jade.tilab.com/), este trabajo se centrará en el comportamiento individual de los agentes. El lenguaje interpretado por JASON es una extensión del lenguaje para especificar agentes AgentSpeak(L) (Georgeff&Lansky, 1987) el cual es una extensión eficiente de la programación lógica para sistemas BDI. Los tipos de agentes especificados con AgentSpeak (L) son referidos a veces como sistemas de planificación reactivos.
JASON, a comparación de otros sistemas para desarrollar agentes BDI, posee la ventaja de ser multiplataforma al estar desarrollado en el lenguaje JAVA, está disponible su código Open Source y es distribuido bajo la licencia GNU LGPL. Además de interpretar el lenguaje AgentSpeak (L) original, posee las siguientes características:

- Negation fuerte de fórmulas,
- Manejo de planes de falla,
- Comunicación entre agentes,
- Anotaciones en las creencias,
- Soporte del desarrollo de entornos (Programados en JAVA),
- Anotaciones en las etiquetas o labels de los planes que pueden ser usadas por funciones selectoras modificadas o extendidas.
- Posibilidad de correr un sistema multiagente distribuido sobre la red.
- Extensible (funciones selectoras, funciones de verdad y la arquitectura completa).
- Librería de acciones internas esenciales.
- Acciones internas extensibles por el usuario, programadas en JAVA.

4.5.1 Arquitectura de un Agente en AgentSpeak

Un agente en AgentSpeak (L) es creado especificando un conjunto de creencias (beliefs) y un conjunto de planes (plans). Otros elementos relevantes son los objetivos (goals) del agente y los eventos disparadores (trigger events) que sirven para representar la parte reactiva de un agente.
***Creencias*:** Representarán las creencias del agente respecto a su entorno. Un átomo de creencia (belief atom) es un predicado de primer orden. Los atomos de creencias y sus negaciones son literales de creencia (belief literals). Un conjunto inicial de creencias es tan sólo una colección de átomos de creencia.

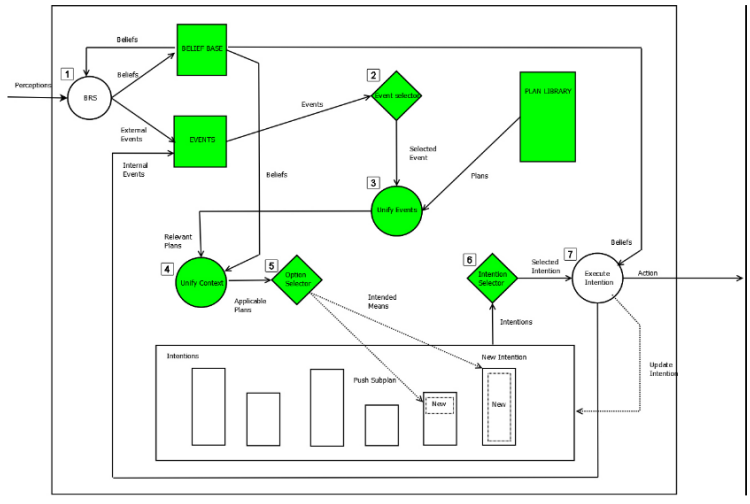
***Objetivos*:** Representarán los objetivos del agente, AgentSpeak (L) distingue sólo dos tipos de objetivos (goals): *achivement goals* y *test goals.* Estos dos son predicados prefijados por los operadores '!' y '?' respectivamente.
(i) *achivement goal*: denota que el agente quiere alcanzar un estado en el mundo donde el predicado asociado sea verdadero (en la práctica, esto lleva a la ejecución de subplanes) y
(ii) *test goal*: devuelve una unificación asociada con un predicado en el conjunto de creencias del agente, si no hay asociación simplemente falla.
Evento disparador (trigger events) define que eventos pueden iniciar la ejecución de un plan. Un evento puede ser interno, cuando un sub-objetivo tiene que ser logrado, o externo, cuando es generado por actualizaciones de creencias debido a una percepción del ambiente. Hay dos tipos de eventos disparadores, relacionados con el hecho de que agregan o quitan actitudes mentales (creencias u objetivos). Estos son prefijados por '+' y '-' respectivamente.
Planes son acciones básicas que un agente puede realizar sobre su ambiente. Estas acciones están también definidas como predicados de primer orden, pero con símbolos especiales, llamados símbolos de acción, usados para distinguirlos de otros predicados. Un plan está formado por un evento disparador (denotando el propósito del plan), seguido por una conjunción de literales de creencia representando un contexto. Para que el plan sea aplicable, el contexto debe ser una consecuencia lógica de las actuales creencias del agente. El resto del plan es una secuencia de acciones básicas o subobjetivos que el agente tiene que lograr (o testear) cuando el plan es elegido para su ejecución.
Ejemplo de un plan en AgentSpeak (L):
+concierto = (A,V) : Gusta (A)
<- !reservar_tickets (A,V).
+!reservar_tickets\= (A,V) : ˜ ocupado (tel)
<- llamar (V).
. . .
<- !escoger_asientos (A,V).
Este ejemplo representa que cuando el agente obtiene el conocimiento de que un concierto A se va a desarrollar en el lugar V (representado por *concierto(A,V)*) y si le gusta al agente el concierto A (*Gusta (A)*), entonces tendrá como nuevo objetivo reservar tickets para este evento (*!reservar_tickets (A,V)*). Luego, cuando el objetivo es agregado a las creencias del agente, el plan que se activará es el siguiente: si el teléfono no está ocupado (*˜ ocupado (tel)*), *llamar a V* (lugar de reserva), más otras acciones (denotadas por "..."), finalizadas por el subobjetivo de escoger los asientos *(!escoger_asientos (A,V))*.
En AgentSpeak(L) un agente ag es especificado como un conjunto de creencias bs (la base de creencia inicial) y un conjunto de planes *ps* (la librería de planes del agente). Las fórmulas atómicas *at* del lenguaje son predicados $P(t_1,...,t_n)$ donde P es un símbolo de predicado, A es un símbolo de acción, y $t_1,...,t_n$ son términos estándar de lógica de primer orden.
Un *plan* está definido por $p ::= te : ct < -h$, donde *te* es un evento disparador, *ct* es el contexto del plan y h es una secuencia de acciones, objetivos, o actualizaciones de creencias; *te:ct* es la cabeza del plan y h es el cuerpo. Un conjunto de planes está dado por *ps* como una lista de planes.

Un evento disparador puede ser el agregado o eliminado de una creencia ($+at$ y $-at$ respectivamente), o el agregado o quitado de un objetivo ($+g$ y $-g$ respectivamente). Finalmente, el cuerpo del plan h puede estar constituido por una acción $A(t_1,...,t_n)$, un goal g o la actualización de las creencias $+at$ y $-at$.
Para ver detalles de la sintaxis del AgentSpeak(L) se puede consultar (Bordini&Hübner 2007).

Semántica Informal

El intérprete de AgentSpeak (L) también maneja un conjunto de eventos y un conjunto de intenciones que será la lista de acciones que el agente decide ejecutar. Su funcionamiento requiere tres funciones de selección. La función de selección de eventos (SE), que selecciona un sólo evento del conjunto de eventos. Una función de selección de opciones (SO), que selecciona un plan del conjunto de planes aplicables. Por último, una función de selección de intenciones (SI). Estas funciones selectoras pueden definirse según el comportamiento de cada agente que las utilice.
Las intenciones son cursos de acciones que un agente se compromete a hacer para manejar cierto evento. Cuando un evento es interno, consecuencia de una acción ejecutada por un agente, éste es acompañado por la intención que lo generó. Al estar el agente interactuando con el ambiente a través de sus percepciones recibirá nuevas creencias que serán eventos externos. Estos eventos externos crean nuevas intenciones, representando focos de atención separados. Los eventos internos son generados por el comportamiento interno del sistema o ejecución del mismo y los externos, son los eventos percibidos por los agentes.
La Figura describe cómo trabaja un intérprete de AgentSpeak (L):

Figura 4.3: Máquina de estados de un agente PRS (Bordini & Hübner, 2007)



En cada ciclo de interpretación de un agente, se actualiza la creencia del agente y con ello la lista de eventos, que pudo ser generado por una percepción del ambiente (evento externo) o por la ejecución de intenciones (acciones), lo cual produce un evento interno (ver en la figura (1)). Se asume que las creencias son actualizadas por las percepciones. La función de revisión de

creencias no es parte del intérprete pero es un componente necesario de la arquitectura del agente.
Después de que el selector de eventos (SE) haya seleccionado el evento (2), el agente tiene que unificar este evento con eventos disparadores en la cabeza de los planes (3). Esto genera un conjunto de planes relevantes con sus respectivos contextos, se eliminan aquellos cuyos contextos no se satisfagan con la base de creencias del agente, para formar un conjunto de planes aplicables (4). Este conjunto de planes se denominan "opciones". Estas opciones se generaron a partir de un evento externo o interno. Entonces el selector de opciones SO elige uno de estos planes (5), luego pone este plan con alguna intención existente (si el evento fue interno) o crea una nueva intención (si el evento fue externo).
Todo lo que queda al agente es seleccionar una intención (6) para ser ejecutada en el ciclo (7). La función de selección de intenciones (SI) se encarga de esto. La intención incluye un plan y la primer fórmula es tomada para su ejecución.
Jason presenta mejoras respecto a la sintaxis de AgentSpeak (L), siempre se deberá tener en cuenta que el estado de un agente, a lo largo de su ciclo de vida, está representado por un conjunto de creencias (beliefs) y un conjunto de planes (plans) y su comportamiento estará reflejado en el comportamiento de las funciones selectoras. Una de las diferencias m ás importante que distingue a JASON de la sintaxis de AgentSpeak (L) es la inclusión de las anotaciones tanto en las creencias como en los planes. Las anotaciones en las creencias permiten introducir por ejemplo, información sobre la fuente de la información, grado de creencia, etc. Por ejemplo se puede crear una creencia que sea X[MIANOTACION]. Las anotaciones en los planes pueden ser usadas para dar la relevancia del plan, de modo de ordenar la ejecución de los planes aplicables de acuerdo a su relevancia.
También son válidas en JASON fórmulas atómicas $p(t_1, ..., t_n)$, y $\not p(t_1, ..., t_n)$ donde $\not$ denota una negación fuerte. La negación por falla es usada en los contextos de los planes y se denota por 'not' precediendo un literal. El contexto es por lo tanto una conjunción de literales. Los términos en JASON pueden ser variables, listas (estilo PROLOG), como también números enteros, de punto flotante o strings. Como en PROLOG, los operadores relacionales infijos son permitidos en los contextos de un plan y '_' es usada como variable anónima.
En JASON las fórmulas atómicas pueden contener anotaciones, representados por una lista de términos encerradas en corchetes inmediatamente siguiendo una fórmula. Dentro de una base de creencias (beliefs), las anotaciones son usadas, por ejemplo, para registrar las fuentes de la información, o para representar un valor asociado al grado de creencia (belief degree) de que esa fórmula es cierta.
Debido a las anotaciones, las unificaciones se vuelven más complicadas. Cuando se intenta unificar una formula atómica con anotaciones at_1 $[s_{11}, \ldots, s_{1N}]$ con otra fórmula atómica at_2 $[s_{21}, \ldots, S_{2M}]$ no solamente at_1 debe unificar con at_2 sino que $\{s_{11}, \ldots, s_{1N}\}$ debe estar incluido $\{s_{21}, \ldots, S_{2M}\}$. Si por ejemplo, at_1 aparece en un contexto de un plan y at_2 en la base de belief. No solamente at_1 debe unificar con at_2 sino que todas las listas de términos de la anotación asociada a at_1 deben ser corroboradas por la anotación asociada a at_2.

Como caso más general, además de que un agente esté representado por un conjunto de beliefs iniciales y un conjunto de planes, los agentes pueden contener goals iniciales que pueden aparecer entre los beliefs iniciales de un agente. Los goals iniciales se representan como beliefs con un prefijo '!' y no pueden contener variables.
Otro cambio sustancial de JASON es que la base de belief puede contener reglas estilo PROLOG y la misma sintaxis usada en el cuerpo de una regla puede ser usada en el contexto de los planes.
Los operadores "+" y "-" en los cuerpos de los planes son usadas para poner o sacar creencias que trabajan como "notas mentales" del agente mismo. El operador "+" agrega una creencia después de remover, si existe, la primer ocurrencia de la misma en la base de belief. Por ejemplo , *+a(X+1)* primero remueve *a(_)* y agrega *a(X+1)*.
Las variables pueden ser usadas en lugares donde una formula atómica es esperada, incluyendo eventos disparadores, contextos y cuerpos de los planes. En el contexto o el cuerpo de un plan también se puede ubicar una variable en vez de una formula atómica pero manteniendo las anotaciones explícitas, observar los siguientes ejemplos de unificación:
Sea X una variable y p una constante:
X [a,b,c] = p [a,b,c,d] (unifican y X es p)
p [a,b] = X [a,b,c] (unifican y X es p)
X [a,b] = p [a] (no unifican) pues [a,b] no está incluido en [a]

Los planes tienen etiquetas representados por [@$atomic\_formula$]. Una etiqueta de un plan puede ser cualquier fórmula atómica, incluyendo anotaciones. Las anotaciones dentro de las etiquetas o labels de un plan pueden ser utilizadas para la implementación de funciones selectoras para planes más complejos.
Finalmente, las acciones internas pueden ser usadas tanto en el contexto como en el cuerpo de un plan. Cualquier símbolo de acción empezando con "." o conteniendo "." denota una acción interna. Las mismas son acciones definidas por el usuario las cuales son ejecutadas internamente por el agente. Se denominan internas para marcar una distinción con las acciones que aparecen en el cuerpo de un plan que denotan las acciones que un agente puede realizar para cambiar el entorno compartido.

4.5.2 Caso de Estudio

Se presenta un caso de estudio que muestra el potencial de esta plataforma para el desarrollo de agentes.
Se desea modelar un agente supervisor de un horno rotativo utilizado para la fundición de metales. El horno además posee un motor que le permite girar lentamente alrededor de su eje principal. Este tipo de hornos es utilizado para la fundición de cobre, latón, bronce y aluminio. El agente deberá analizar constantemente tres variables fundamentales en la operación del horno:

- Temperatura del horno
- Presión de los gases en el interior del horno
- Vibración del motor de rotación

Para esta tarea se cuenta con tres sensores estratégicamente situados. En este contexto se implementará un agente de supervisión que contará con las

lecturas de los tres sensores y a partir de las mismas deberá tomar acciones preventivas. Por lo tanto, los sensores abastecerán al agente de creencias sobre las lecturas y sus grados de certeza asociados los cuales dependerán de las precisiones de los instrumentos utilizados. Estas precisiones pueden variar de acuerdo al rango de valores de las variables en los cuales el horno esta trabajando. En base a las lecturas de los sensores y las acciones preventivas se modelarán los planes del agente. En un momento de tiempo t1, el horno se encuentra trabajando a una temperatura de 700 grados (creencia B1), a una presión de 80 bars (B2) y a un nivel de vibración de 8 (B3). Los hechos sensados ingresarán de acuerdo a la sincronización de los lectores.
El agente además posee acciones de supervisión de acuerdo a las lecturas de los sensores:
- Si la lectura de temperatura supera los 300 grados se deberá tomar la medida de encender el ventilador para que el horno comience a enfriarse, esta acción se encuentra modelada en el plan P1 (alerta temperatura).
- Si la temperatura supera los 600 grados, el horno comienza a operar en un estado crítico de temperatura por lo cual se debe apagar el horno para lograr un enfriamiento total, esta acción se encuentra modelada en el plan P2 (urgencia de temperatura) y posee mayor relevancia que el plan P1.
- Para el caso de la variable presión, si la lectura supera los 35 bars se deberá cerrar una válvula de inyección del horno (plan P3) y si la presión supera los 70 bars se deberá encender una alarma general, dado el peligro de explosión en el horno y se deberán tomar medidas de precaución (plan P4). Las fallas de presión son las causas más frecuentes de accidentes por lo cual el control de esta variable es la más relevante para este sistema de horno.
- También se deberá supervisar el trabajo del motor de rotación del horno, a cierta cantidad de tiempo de trabajo empieza a sufrir el desgaste de sus piezas por lo cual comienza a producir un nivel elevado de vibración. Para evitar una rotura inminente del motor, a un nivel superior de vibración 8, se procede con el frenado del mismo (plan P5).
El *agente supervisor* se especificará en JASON mediante una base de creencias y un conjunto de planes iniciales.

Creencias:

B1= sensor_termico(700).
B2= sensor_presion(80).
B3= sensor_vibracion(8).

Planes:

P1= @alerta_temperatura + sensor_termico(T EM P) : TEM P $>$ 300
$<-$ prende_ventilador.
P2= @urgencia_temperatura + sensor_termico(T EM P) : TEM P $>$ 600
$<-$ apagar_horno.
P3= @alerta_presion + sensor_presion (P RES) : PRES $>$ 35
$<-$ cierra_valvula.
P4= @urgencia_presion + sensor_presion(P RES) : PRES $>$ 70
$<-$ enciende_alarma
P5= @manejo_vibracion + sensor_vibracion(N V L) : N V L $>$ 5
$<-$ frena_motor

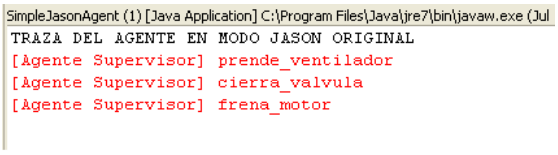
Las creencias del agente generan el conjunto de eventos y los planes iniciales forman la librería de planes del agente. Los eventos se evalúan en el orden que ingresan en el sistema. Los principales pasos de la ejecución para este ejemplo se resumen en:

1. Se tomará el evento generado a partir de B1, dado que la selectora de eventos toma el primero.
2. El evento unifica con los eventos disparadores de los planes P1 y P2 con lo cual los convierte a los mismos en relevantes.
3. Los planes P1 y P2 unifican el contexto contra B1, con lo cual se generan dos opciones a partir de los planes.
4. La selectora de opciones seleccionará la primera opción que es la generada por P1 y se crea una intención.
5. Se ejecuta la acción de la intención generada a partir de P1: **prende_ventilador.**
6. Se selecciona el segundo evento que es el generado a partir de B2.
7. El evento unifica con los eventos disparadores de los planes P3 y P4 con lo cual los convierte en relevantes.
8. Los planes P3 y P4 unifican el contexto contra B2, con lo cual se generan dos opciones a partir de los planes.
9. La selectora seleccionara la opción de P3 por ser la primera y se crea una intención.
10. Se ejecuta la acción de la intención generada a partir de P3: **cierra_valvula.**
11. Se tomará el evento generado a partir de B3.
12. El evento unifica con el evento disparador de P5 y lo convierte en relevante.
13. El plan P5 unifica el contexto contra B3 y se agrega a los planes aplicables.
14. La selectora de opciones toma la primera entre las opciones: P5.
15. Se ejecuta la acción de la intención generada en base a P5: **frena_motor.**

Luego, para este caso, el orden de las acciones resultantes a ejecutar por el agente es:

- **prende_ventilador**
- **cierra_valvula**
- **frena_motor**

Dado el orden de sincronización de los sensores, se ejecutó la acción relacionada a la variable "'temperatura"' primero, por ser la que primero ingresó al conjunto de creencias del agente, luego se trató el sensor depresión y por último el de vibración. Además, se ejecutó antes el plan para el horno P1 sobre P3 siguiendo el orden de ejecución de los planes en el comportamiento de JASON que por defecto respeta el orden de los planes en la librería. Con otras estrategias se puede alterar este orden (por ej, utilizando grados de relevancia de los planes en las anotaciones). A continuación se muestra la traza de la ejecución de este ejemplo en JASON.



4.6 Sistemas Multiagentes

Los agentes operan en algún entorno que suele ser tanto computacional como físico, por ejemplo, un agente recomendador en la Web o un robot que se desplaza en un hospital, etc. El entorno puede ser abierto o cerrado y suele tener más de un agente. Si bien hay situaciones donde un agente puede operar solo, un creciente número de sistemas están siendo vistos en término de sistemas multiagentes, compuestos por agentes autónomos que interactúan entre sí. Aplicaciones orientadas a los agentes se han utilizado en distintos dominios como sistemas de comercio electrónico, sistemas de gestión de procesos distribuidos, sistemas de control de tráfico, etc. En gran parte, esto se debe a que el paradigma de sistemas multiagentes ofrece un conjunto de conceptos y técnicas para modelar, diseñar e implementar sistemas distribuidos complejos. Por otra, se debe a que los sistemas de computos operan cada vez más en forma distribuida y en múltiples dispositivos además, la información también suele estar distribuida.
Siguiendo el paradigma de la Ingeniería de Software Orientada a Agentes –AOSE: Agent Oriented Software Engineering– (Jennings, 2000) estos sistemas pueden desarrollarse como organizaciones de agentes denominados sistemas multiagentes (MAS: Multi-Agent Systems) donde cada agente tendrá su funcionalidad e interactuarán unos con otros y con el entorno en el cuál habitan. Es necesario entonces, analizar propiedades del entorno para que los agentes puedan operar efectivamente e interactuar unos con otros convenientemente. Los entornos proveen una infraestructura computacional para que las interacciones se puedan realizar. Hay que considerar los protocolos de comunicación y de interacción que utilizarán los agentes.
Los protocolos de comunicación permiten a los agentes intercambiar y entender mensajes. Actualmente, el lenguaje de comunicación entre agentes FIPA ACL es considerado un estandar. Por otra parte, los protocolos de interacción permiten a los agentes tener conversaciones, que serán intercambios de mensajes estructurados. Para dar un ejemplo concreto de estos conceptos, un protocolo de comunicación puede especificar que los siguienetes tipos de mensajes pueden ser intercambiados entre dos agentes:

- proponer un curso de acción
- aceptar una acción
- rechazar una acción
- retractar una acción
- estar en desacuerdo con una acción
- hacer una contrapropuesta de acción

A partir de estos tipos de mensajes, la siguiente conversación puede darse entre dos agentes Ag1 y Ag2 (como una instancia de una interacción para la negociación de agentes):

- *Ag1 le propone a Ag2 un curso de acción, Ag2 evalúa la propuesta y*
- *responde Ag2 a Ag1 con:*
  - *envía una aceptación o*
  - *envía una contrapropuesta o*
  - *envía un desacuerdo o*
  - *envía un rechazo*

4.6.1 Características de los Sistemas Multiagentes

Algunas de las características más relevantes a considerar en los MAS son las siguientes:

- Los sistemas multiagentes proveen una infraestructura que especifica los protocolos de comunicación y de interacción entre agentes.
- Los entornos multiagentes suelen ser abiertos y suelen no tener un diseño centralizado.
- Contienen agentes autónomos y que pueden tener intereses propios o ser cooperativos. Pueden tener *una meta común o metas independientes,* si un grupo de agentes comparte una meta global, suelen denominarse *colaborativos* (por ej., un equipo de futbol) en cambio si tienen distintas metas (generalmente opuestas) suelen ser *competitivos* (por ej. agentes de dos equipos de futbol que se enfrentan).
- Pueden *compartir conocimientos sobre el problema* y las posibles soluciones, el *conocimiento global que cada agente tiene* puede incluir control global, consistencia global, metas globales, etc.
- Para que un conjunto de agentes pueda desarrollar una actividad conjunta en un entorno compartido *debe existir algún tipo de coordinación*, la cual puede ser muy compleja. En un grupo de agentes cooperativos puede utilizarse planificación de tareas y en un escenario competitivo, pueden negociarse propuestas.

Además de estas características, para que un agente o sociedad de agentes pueda tener una buena performance, es necesario analizar el entorno en que se encuentra. Un análisis detallado de las distintas propiedades del entorno a tener en cuenta se presenta en (Russell & Norvig, 2009) y una breve descripción es la siguiente:
- *accesibles/inaccesibles:* en un entorno accesible los sensores proporcionan toda la información relevante del entorno para el agente.
- *deterministas/no deterministas*: en un entorno determinista el estado siguiente puede obtenerse a partir del actual y de las acciones del agente.
- episódicos/no episódicos: en un entorno episódico la experiencia del agente está dividida en episodios independientes.
- estáticos/dinámicos: un entorno dinámico puede sufrir cambios mientras el agente está razonando.
- discretos vs continuos: en un entorno discreto existe un número concreto de percepciones y acciones claramente definidos.

4.6.2 Comunicación

Si dos agentes van a comunicarse sobre algún tema en particular, es necesario que ellos acuerden sobre la terminología de dicho dominio. La alternativa que se plantea es utilizar ontologías como una definición formal de un cuerpo de conocimiento. Conformadas por especificaciones de un conjunto de términos con el fin de proveer una base común de entendimiento sobre un dominio en particular.
Por otra parte, la comunicación es un tema de relevante importancia en ciencias de la computación y muchos formalismos se han desarrollado para representar las propiedades de comunicación en sistemas concurrentes, donde es necesario sincronizar distintos procesos, como son los agentes. En general los agentes no pueden forzar a otros agentes a realizar acciones, lo que ellos

pueden realizar son acciones de comunicación (*speach acts*) con la intención de influenciar a otros apropiadamente. Por ejemplo si un agente le comunica a otro que "es un día cálido en Buenos Aires" querrá influenciar las creencias del otro agente de modo que él incorpore ese conocimiento a su base.
La teoría de *actos del habla -speech acts-* trata a la comunicación como acciones y que al igual que otras acciones, dependen de las intenciones de los agentes (Cohen & Levesque 1985). KQML es un lenguaje basado en mensajes para la comunicación de agentes. Un mensaje puede verse como un objeto donde cada mensaje tiene una performativa (que puede pensarse como la clase de un objeto) y un número de parámetros (pares atributo/valor, que pueden verse como instancias de variables. La aceptación de este lenguaje por parte de la comunidad de MAS fue importante, sin embargo tuvo algunos problemas que llevó al consorcio FIPA (Foundation for Intelligent Physical Systems: http://www.fipa.org/) al desarrollo de un lenguaje muy cercano, FIPA Agent Communication Language (ACL). La principal diferencia entre el lenguaje FIPA ACL y KQML es el conjunto de performativas que proveen y que está provisto de una semántica formal. Un ejemplo de mensaje en este lenguaje es el siguiente:

(inform
: sender agent1
: receiver agent2
: content (price good2 150)
: language sl
: ontology hpl-auction
)

Para facilitar el rápido desarrollo de un sistema multiagente usando FIPA ACL, se han generado varias plataformas que soportan este protocolo de comunicación. La mas conocida y utilizada es JADE –Java Agent Development Environment– (http://jade.tilab.com/). Esta plataforma provee paquetes y herramientas para que desarrolladores JAVA puedan crear sistemas de agentes en concordancia con FIPA.

4.6.3 Coordinación

Para que los agentes que integran un MAS puedan realizar alguna actividad en un entorno compartido es necesaria algún tipo de coordinación de sus comportamientos y actividades. En agentes no-antagonistas, la coordinación suele ser una cooperación que hace necesaria una planificación (distribuida o centralizada), mientras que entre agentes competitivos o que tienen intereses individuales, suele existir una negociación. La negociación también tiene lugar en agentes colaborativos que tienen que intercambiar recursos para lograr sus objetivos. De las distintas formas de interacción se analizará a continuación la negociación ya que ha ganado gran interés en este último tiempo por ser un aspecto fundamental en distintos sistemas multiagentes.

Negociación

En (Jennings et al., 2001) se describe a la negociación como una "forma de interacción en la que dos o más agentes que tienen diferentes intereses intentan encontrar un compromiso o consenso sobre algún asunto". Los autores consideran que la investigación en el área debe tratar con los siguientes

tres temas fundamentales: el Objeto de Negociación, que es asunto por el cual se negocia (por ej. el precio de un producto, la fecha o el lugar de una reunión, etc.), el Protocolo de negociación, que está conformado por las reglas que gobiernan la interacción (por ej. subastas holandesas, inglesas, etc.) y los modelos de toma de decisiones, que describen el comportamiento de los participantes. También afirman que la negociación puede ser vista como una búsqueda distribuida sobre un espacio de potenciales acuerdos entre las partes. La dimensión y la topología de este espacio son determinadas por la estructura del objeto de negociación y del protocolo utilizado. La negociación automática tiene impacto directo en distintas aplicaciones entre las que se destaca el comercio electrónico: desde subastas donde se negocian asuntos simples (por ej. el precio de un producto) hasta negociaciones más complejas donde se discuten varios asuntos (por ej. garantía, forma de pago, tiempos, etc.). Otra aplicación relevante de la negociación es en problemas de planificación donde se deben asignar tareas y/o recursos. Un ejemplo de sistemas que negocia es Cognitor (http://www.cognicor.com/), un sistema de reclamos que automatiza las negociaciones con clientes disconformes que realizan quejas a una empresa.

En la literatura que trata la negociación, la noción de recursos es muy general. Los recursos pueden ser de distinto tipo, por ejemplo: un cuadro valioso, el derecho de explorar un área en busca de un mineral, una cantidad de ciclos del procesador de una PC, el tiempo de entrega de una mercadería, el conocimiento para llevar adelante una tarea, etc. Una característica a tener en cuenta respecto a los recursos es que estos suelen ser escasos y demandados por uno o más agentes. Los principales enfoques de negociación automatizada en los sistemas multiagente pueden agruparse en las siguientes clases principales según su enfoque [Jennings 2001]: basados en Teoría de Juegos, basados en Heurísticas y basados en Argumentación. A continuación se describen sus características fundamentales:

Negociación basada en Teoría de juegos: La teoría de juegos es una rama de la economía que estudia las estrategias e interacciones entre agentes que intentan maximizar su propia utilidad en base a los movimientos que pueden realizar y la utilidad que reciben luego de que se realizan los movimientos. En teoría de juegos, se intenta determinar la estrategia óptima analizando la interacción de un juego con participantes idénticos. Dado un escenario de negociación particular que involucre a agentes de negociación, las técnicas de teoría de juegos pueden ser aplicadas para resolver dos problemas claves: el diseño de un protocolo apropiado que gobernará las interacciones entre los agentes participantes y el diseño de las estrategias (modelo de decisión del agente) que utilizarán los agentes durante el proceso de negociación. Se tienen dos desventajas al aplicar este enfoque. Por un lado, las estrategias que dan mejores resultados en la teoría tienden a ser computacionalmente intratables y por lo tanto no pueden aplicarse al desarrollo de agentes concretos. Por otra parte, la teoría considera que el espacio de resultados es completamente conocido, lo que suele no ocurrir en escenarios realistas ya que mucha información se puede obtener durante el proceso de negociación. Ejemplos de este enfoque se puede ver en (Rosenschein, 1994).

Negociación basada en Heurísticas: Con el objetivo de atacar las limitaciones computacionales, han emergido los enfoques de negociación basados en heu-

rísticas. Las heurísticas son reglas generales que producen buenos resultados en contextos donde las suposiciones son relajadas. En este tipo de enfoque las estrategias están determinadas por heurísticas que permiten realizar una búsqueda no exhaustiva en el espacio de negociación. Mientras los métodos heurísticos pueden superar algunas de las deficiencias de los enfoques basados en teoría de juegos, sufren de algunas desventajas. La primera es que las soluciones son subóptimas y la segunda es que los modelos necesitan una evaluación empírica (Faratin, 2000).

Negociación basada en Argumentación (ABN): Si bien los enfoques anteriores son adecuados para resolver un gran rango de problemas, en estos modelos de negociación los agentes sólo pueden intercambiar potenciales acuerdos (propuestas) y no pueden intercambiar otra información adicional. Esto puede ser problemático cuando el agente tiene información parcial del entorno o sus decisiones dependen de las decisiones de los otros agentes. Además, otra limitación que presentan los enfoques anteriores, es que las utilidades o preferencias de los agentes se suponen completamente caracterizadas antes de la interacción y no cambian durante la negociación. Situaciones más complejas donde la información es incompleta no responden a esta caracterización. Los enfoques basados en argumentación permiten superar estas deficiencias.

En el contexto de la negociación basada en argumentación, un argumento es una pieza de información que permite a un agente justificar su postura de negociación o influenciar la postura de otro agente. Los argumentos permiten que las propuestas, además de ser simplemente aceptadas o rechazadas, también puedan ser criticadas. De esta manera la negociación puede ser más eficiente ya que la contraparte puede tener una mejor posición para realizar una oferta diferente. Si bien no existe un consenso sobre los elementos que caracteriza a un framework de negociación basado en argumentación, en (Rahwan, 2003) se menciona que deben considerarse algunos aspectos externos e internos de los agentes. Entre los *elementos externos a los agentes* se destacan los siguientes: (i) el *lenguaje de comunicación* que permite expresar propuestas, críticas, creencias, preferencias, objetivos etc. (ii) el *lenguaje de dominio*, que permite expresar información del dominio relevante (color, garantía, forma de pago, etc.). (iv) el *protocolo de negociación* compuesto por las reglas que regulan la comunicación. (v) las *bases de información, repositorio* donde se mantiene información importante para la negociación (historial del diálogo, reputación de los participantes, etc.).

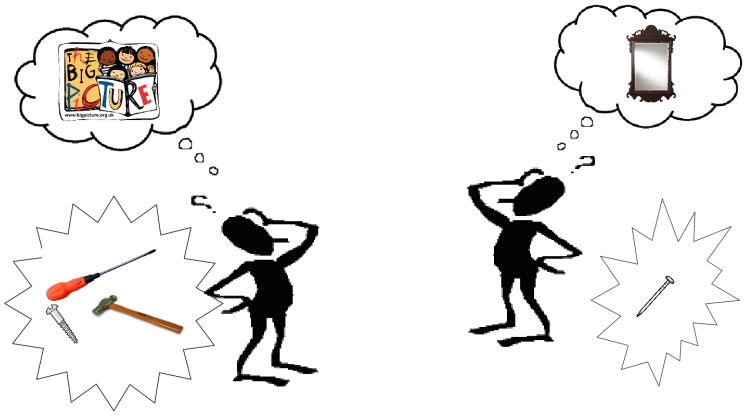
Por otro lado, *los elementos internos a los agentes* corresponden a las funcionalidades que deben realizar los agentes basados en argumentación a través de una arquitectura adecuada. Las funcionalidades que estos autores citan son: interpretar las locuciones entrantes, evaluar los argumentos, actualizar el estado mental, generar argumentos candidatos, seleccionar argumentos candidatos y generar la locución de salida.

Un escenario de negociación cooperativa

Un escenario tıpico de negociación involucra dos agentes cooperativos que poseen recursos y tienen la necesidad de realizar un intercambio para alcanzar sus objetivos. Además, los agentes pueden tener creencias incompletas o erróneas sobre el oponente. Para acordar cuales recursos trocar, establecen una comunicación en donde se ofrecen propuestas de posibles o intercambios y se responden críticas y contrapropuestas. El rumbo de la comunicación va

cambiando en la medida de que los agentes revelan los recursos que poseen y las características de estos. Un modelo de negociación automática basada en argumentación para agentes intencionales que quieren negociar en este tipo de escenarios se presenta en (Pilotti et al. 2012).
A continuación mostraremos un ejemplo de diálogo entre dos agentes que negocian siguiendo una variante de un ejemplo de la literatura.

Ejemplo:
Sean dos agentes benevolentes Ag1 y Ag2, los cuales se ilustran en la siguiente Figura. El agente Ag1 tiene como objetivo colgar un cuadro y tiene como recursos un tornillo, un destornillador y un martillo. El tiene el conocimiento de como colgar el cuadro con un clavo y un martillo y además, conoce como colgar un espejo con un tornillo y destornillador.
Por su parte, el Ag2 tiene como objetivo colgar un espejo, tiene un clavo y el conocimiento de cómo colgar un espejo con un clavo y un martillo.



Ninguno de los dos agentes puede lograr su objetivo con sus propios recursos y conocimento, necesitan intercambiar elementos y/o conocimiento para lograrlo. Utilizando agentes intencionales que siguen un modelo de negociación que utiliza revisión de creencias, se puede obtener este diálogo entre los agentes a través del cuál alcanzan a negociar un intercambio de recursos. Las reglas *r1 & r2 => O* representan el "knowhow" de cómo con ciertos recursos (por ej. *r1 y r2*) se puede lograr *el objetivo O*.
Ag1: Te propongo que me proveas de [clavo] porque
If uso [martillo, clavo & martillo => colgarCuadro] then
puedo lograr [corgarCuadro] a cambio de [tornillo]
Ag2: Te propongo que me proveas de [colgarEspejo] porque
if uso [] then puedo lograr [colgarEspejo] a cambio de [clavo]
Ag1: Te propongo que me proveas de [clavo] porque
if uso [martillo, clavo & martillo => colgarCuadro] then puedo lograr [colgarCuadro] a cambio de [tornillo, destornillador, tornillo & destornillador => colgarEspejo]
Ag2: Acepto, te doy un [clavo] y me das
[tornillo, destornillador, tornillo & destornillador => colgarEspejo]
Encontrar distintos modelos de negociación colaborativa o competitiva, que tengan un sustento teórico, pero puedan llevarse a un modelo computacional eficiente para ser aplicados en sistemas multiagentes, es un campo abierto de

investigación.

4.7 Bibliografía

A continuación se presenta la bibliografía que se ha referenciado en el capítulo y otras citas donde los lectores pueden ampliar algunos de los temas desarrollados.

Bergenti F., Gleizes M. P. and Zambonelli F. (Eds.), Methodologies and Software Engineering For Agent Systems: The Handbook of Agent-oriented Software Engineering, Kluwer Academic Publishing (New York, NY), July 2004.

Bordini, R. H., Hübner, J. F., and Wooldridge M., Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley and Sons, 2007.

Bratman M. E., Israel D. J. and Pollack M. E., Plans and resource-bounded practical reasoning. Computational Intelligence, 4: 349-355, 1988.

Casali, A., Ll. Godo & C. Sierra. "A graded BDI agent model to represent and reason about preferences." Artificial Intelligence, Special Issue on Preferences Artificial Intelligence 175:1468–1478, 2011.

Cohen P. and Levesque P. Speech acts and rationality, Proceeding ACL '85 Proceedings of the 23rd annual meeting on Association for Computational Linguistics, pp. 49-60, Stroudsburg, PA, USA, 1985.

Dennet D. C., The Intentional Stance. MIT Press, Cambridge, MA, 1987.

D'Inverno, M., D. Kinny, M. Luck & M. Wooldridge. A formal specification of dMARS. In Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages, ed. M.P. Singh, A.S. Rao & M. Wooldridge. Montreal: Springer-Verlag, 1998.

Faratin P., Sierra C. and Jennings N., Using Similarity Criteria to Make Negotiation Trade-Offs *International Conference on Multiagent Systems (ICMAS-2000), Boston, MA.*, 119-126, 2000.

Ferguson I. A., TouringMachines: an Architectures for Dynamic, Rational , Mobile Agents. PhD Thesis, Clare Hall, University of Cambrage, UK, 1992.

Georgeff M. P. and Lansky A. L., Reactive reasoning and planning. In Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), 677682, Seattle, WA, 1987.

Georgeff M., Pell B., Pollack M., Tambe M. and Wooldridge M., 1999. The Belief-Desire-Intention Model of Agency. In J. P. Muller, M. Singh, and A. Rao (Eds.), Intelligent Agents V - LNAI, Volume 1365, Springer-Verlag, 1999.

Ingrand, F. "OPRS development environment", 2004.

Ingrand F. F., Georgeff M. P. and Rao A. S., An architecture for real time reasoning and system control. IEEE Expert, 7(6), 1992.

Jennings N. R., Faratin P., Lomuscio A. R. ,Parsons S. , Sierra C. and Wooldridge M. Automated Negotiation: Prospects, Methods and Challenges, International Journal of Group Decision and Negotiation 2001, 10, (2), 199-215, 2001.

Jennings N. R., Sycara K. and Wooldridge M., A roadmap of agent research and development. Autonomous Agents and Multi-Agent Systems. 1(1), 7-38, 1998.

Maes P., The Agent netword architecture (ANA). SIGART Bulletin, 2(4) 115-120, 1991.

Luck M., McBurney P. and Preist C., Agent Technology: Enabling Next Ge-

neration Computing. AgentLink, 2003, ISBN 0854 327886. (http://www.agentlink.org/roadmap/
Muller J.P., A cooperation Model for Autonomous Agent. In J. P. Muller, M. Wooldridgde and N. R. Jennings (Eds.), Intelligent Agents, III, LNAI Volume 1193, 245-260, Springer, Berlin, 1997.
Pilotti P., Casali A. and Chesñevar C. A Belief Revision Approach for Argumentation-based Negotiation with Cooperative Agents. In Proceedings of Argumentation in Multi-Agent Systems Workshop (ArgMAS 2012), Peter McBurney, Simon Parsons and Iyad Rahwan (Eds) pp. 129-147, Valencia, España, junio de 2012.
Rahwan I., Ramchurn S. D. , Jennings N. R., Mcburney P. , Parsons S. and L. Sonenberg. Argumentation-based negotiation. *Knowl. Eng. Rev. Vol 18, No 4, p 343-375,* 2003.
Rao, A. and Georgeff M., BDI agents: From theory to practice. In proceedings of the 1st International Conference on Multi-Agents Systems, pp 312-319, 1995.
Rosenschein J. S. and Zlotkin G. Rules of Encounter MIT Press, 1994 .
Jennings N.R., On Agent-Based Software Engineering. Artificial Intelligence 117(2), 277-296, 2000.
Russell, S. and Norvig P., Artificial Intelligence: a modern approach, 3rd edition Prentice Hall. Englewoods Cliifs, NJ, 2009.
Weiss G., In Weiss G. (Ed.), Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence, The MIT Press, 1999.
Wooldridge M and Jennings N. R., Intelligent Agents: theory and practice. The Knowledge Engineering Review, 10(2), 115-152, 1995.
Wooldridge, M., Introduction to Multiagent Systems, 2nd Edition, John Wiley and Sons Ltd., 2009.

5 — Introduccion al Aprendizaje

Eliana Scheihing
Universidad Autral de Chile, Chile

5.1 CONCEPTO DE APRENDIZAJE

En la actualidad, gracias a los avances de la informática, es posible almacenar y procesar grandes cantidades de datos, así como acceder a datos ubicados físicamente en otras localidades geográficas a través de las redes computacionales.

En este contexto aparece el concepto de aprendizaje en informática, denominado también aprendizaje de máquina o automático, que corresponde a programas computacionales que buscan optimizar los parámetros de un modelo usando datos previos o datos de entrenamiento. Los modelos pueden ser inductivos, cuando permiten hacer predicciones sobre el futuro o bien descriptivos cuando permiten generar conocimiento a partir de los datos.

El aprendizaje automático usa la teoría estadística para construir modelos matemáticos, pues de esta manera es posible hacer inferencias a partir de una muestra. La ciencia de la computación es requerida en la fase de entrenamiento para la implementación de algoritmos de optimización eficientes, además de ser necesaria en las tareas de almacenamiento y procesamiento de grandes volúmenes de datos. Una vez que un modelo es ajustado, se requiere también eficiencia en su representación y solución algorítmica para la fase de inferencia.

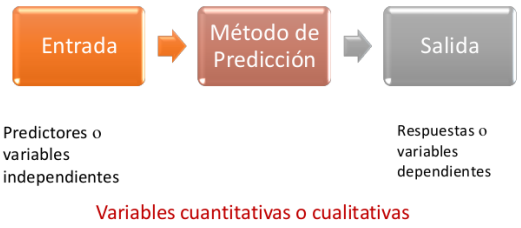
La Minería de datos es el concepto acuñado en el mundo del negocio para la aplicación del aprendizaje automático en grandes volúmenes de datos con el objeto de extraer información de los mismos. Aplicaciones de Minería de Datos se han extendido a diversas áreas, como por ejemplo el estudio del comportamiento de consumo de los clientes de un supermercado. En finanzas bancarias los datos históricos se utilizan para construir modelos de riesgo de créditos, detección de fraudes entre otros. En el área de manufactura, los modelos de aprendizaje se utilizan para optimización, control y resolución de problemas. En medicina, existen aplicaciones para el diagnóstico médico. En telecomunicaciones, los patrones de llamadas son analizados para optimización y maximización de la calidad de servicio. En ciencia, grandes volúmenes de datos se pueden analizar también con estas técnicas.

Los algoritmos de reconocimiento de patrones son otro ejemplo de aplicación del aprendizaje automático para solucionar problemas de visión, reconocimiento de discurso y robótica.
Los algoritmos de aprendizaje automático se pueden clasificar en supervisados y no supervisados. El aprendizaje supervisado corresponde a la situación en que se tiene una variable de salida, ya sea cuantitativa o cualitativa, que se desea predecir basándose en un conjunto de características. Se establece un modelo que permite relacionar las características con la variable de salida. Luego se considera un conjunto de datos de entrenamiento en los cuales se observan tanto los valores de la variable de salida como de las características para determinados individuos (personas u otros). Usando tales datos se ajustan los parámetros del modelo, con lo cual es posible predecir valores de la variable de salida para nuevos individuos. Este proceso de ajuste del modelo se denomina aprendizaje supervisado, puesto que es un proceso de aprendizaje guiado por los valores de la variable de salida.

El aprendizaje no supervisado corresponde a la situación en que existe un conjunto de datos que contienen diversas características de determinados individuos, sin que ninguna de ellas se considere una variable de salida que se desee predecir. En este caso la tarea de aprendizaje es describir como están organizados los datos, posibles asociaciones entre ellos o agrupamientos.

5.2 APRENDIZAJE SUPERVISADO

El objetivo del aprendizaje supervisado es la predicción: Dado el valor de un vector de entrada X, generar una buena predicción $\hat{Y}$ de la salida Y



5.2.1 Un primer ejemplo: la regresión lineal

Dado el vector de entradas:

X=(X1, X2,. . . ,Xp)
Predecir la salida Y con el modelo

$$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j$$

$\hat{\beta}_0$ sesgo

Si se incluye la constante 1 en X, el modelo queda:

$$\hat{Y} = X^T \hat{\beta}$$

Aquí Y es escalar. Si Y es un vector de dimensión K, entonces X sería una matriz de dimensión pxK.

En el espacio entrada-salida (p+1)-dimensional, (X, $\hat{Y}$) representa un hiperplano

Si la constante se incluye en X, entonces el hiperplano pasa por el origen

$f(X) = X^T\beta$: es una función lineal, y en consecuencia: $f'(X) = \beta$: es el vector que apunta en dirección perpendicular al plano

5.2.2 Procedimiento de entrenamiento:

Método de mínimos cuadrados

N = #observaciones

Minimizar la suma de los cuadrados de los errores

$RSS(\beta) = \sum_{i=1}^{N}(y_i - x_i^T\beta)^2$

O de manera equivalente, en notación vectorial

$RSS(\beta) = (y - X\beta)^T(y - X\beta)$

Donde

$$y = (y_1, ...y_N) \text{ y } X = \begin{pmatrix} x_{11} & ... & x_{1p} \\ \vdots & \ddots & \vdots \\ x_{N1} & ... & x_{Np} \end{pmatrix}$$

con y_i la salida de la observación i, cuyos datos de entrada son $x_{i1}...x_{ip}$

Esta función cuadrática siempre tiene mínimo global, aunque no necesariamente único. Derivando con respecto a β se construyen las ecuaciones normales

$X^T(y - X\beta) = 0$

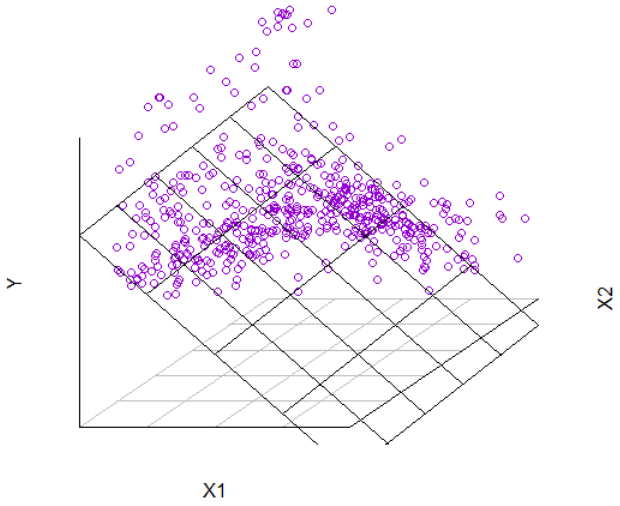
Si X^TX es no-singular, entonces la solución única es:

$\beta = (X^TX)^1X^Ty$

El valor ajustado dada una entrada x es:

$\hat{y}(x) = x^T\hat{\beta}$

La superficie es caracterizada por la siguiente figura.



5.2.3 El problema de la Clasificación

Uno de los problemas clásicos que abordan los algoritmos de aprendizaje supervisado es la clasificación. Consideremos el siguiente ejemplo:

Supongamos que se disponen de datos que consisten de dos variables de entrada X1 y X2 y una variable de salida con dos valores 0 ó 1.

Clasificador Lineal

Una estrategia de aprendizaje sería ajustar una regresión con los datos de entrenamiento:

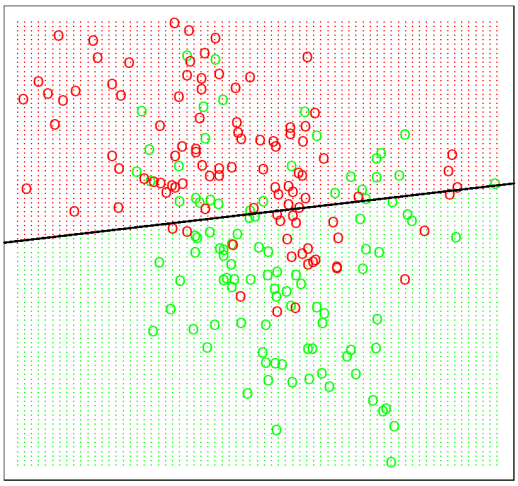
$$\hat{y}(x) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2$$

y luego definir como criterio de clasificación:

$$c(x) = \begin{cases} 0si & \hat{x} < 0{,}5 \\ 1si & \hat{x} \geqslant 0{,}5 \end{cases}$$

En la Figura 5.1 se ha considerado el ajuste de una regresión para datos de la naturaleza del ejemplo, en dónde se ha codificado en rojo los datos cuya variable de salida toman el valor 1 y verde cuando el valor es 0. La línea recta corresponde al criterio de clasificación. Para valores de entrada que se ubiquen en el plano sobre la línea recta, el clasificador predice una salida de valor 1 y en caso contrario el valor que predice es 0.
Los círculos rojos y verdes corresponden a los datos con los cuales se ha entrenado el algoritmo, es decir con los cuales se ha realizado el ajuste de la regresión, y como se observa en la figura, varios de los datos de entrenamiento resultan mal clasificados (círculo verde en zona roja o círculo rojo en

Figura 5.1: Clasificador lineal para dos variables de entrada y una variable de salida dicotómica



zona verde). Si bien esta estrategia es simple, resulta en muchos datos mal clasificados si el problema no es lineal.

Los k vecinos más cercanos (kNN)

Este algoritmo no hace suposiciones sobre la relación entre las variables de entrada y de salida, si no que estima el valor de la variable de salida en función de los k vecinos más cercanos respecto de las variables de entrada.

El estimador tiene la forma:

$$\hat{y}(x) = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i$$

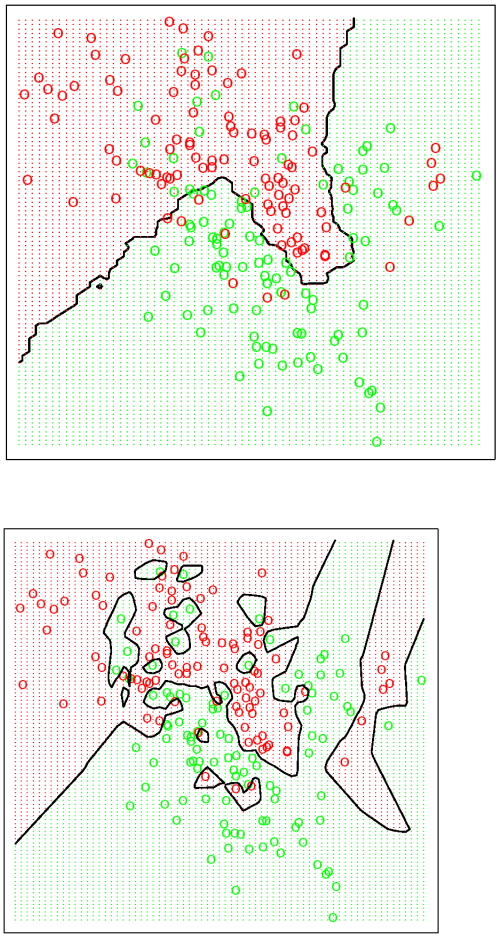
Donde $N_k(x)$ es el conjunto de k puntos más cercanos a x entre los datos de entrenamiento y toma el promedio de sus salidas para predecir y.

El clasificador queda nuevamente como:

$$c(x) = \begin{cases} 0 si & \hat{y}(x) < 0{,}5 \\ 1 si & \hat{y}(x) \geqslant 0{,}5 \end{cases}$$

En la Figura 5.2 se ha considerado el ajuste con kNN para los datos utilizados previamente, con dos valores de k, k=3 y k=6. En este ejemplo es claro que disminuye el número de datos mal clasificados respecto del ajuste de una regresión. Pero al aumentar el número de vecinos se genera un clasificador poco robusto, que es muy sensible a nuevos datos.

Figura 5.2: Clasificador kNN para dos variables de entrada y una variable de salida dicotómica. Resultados para k=3 y k=6 respectivamente



5.2.4 Comparación de los dos enfoques

5.3 Bibliografía

Alpaydin, Ethem (2010) Introduction to Machine Learning. The MIT Press, Segunda Edición.

James, G., Witten, D., Hastie, T. y R. Tibshirani (2013) An Introduction to Statistical Learning with Applications in R. Series: Springer Texts in Statistics.

Hastie, Trevor, Tibshirani, Robert y Jerome Friedman (2009) The Elements of Statistical Learning. Data Mining, Inference, and Prediction, Series: Springer Series in Statistics, Segunda Edición.

Russell Stuart y Peter Norvig (2004) Inteligencia Artificial. Un Enfoque Moderno. Prentice-Hall.

Mínimos Cuadrados	K vecinos más cercanos (kNN)
Número de parámetros: p = #características donde N = #observaciones	Número de parámetros: p= N/k
Poca varianza (robusto)	Mucha varianza(no es robusto)
Muy sesgado (depende de condiciones muy restrictivas)	Escaso sesgo (requiere supuestos mínimos)
Útil cuando datos provienen de distribuciones Normales bivariadas independientes y de distinta media para cada clase.	Útil cuando datos provienen de una mezcla de varias distribuciones Normales de varianza pequeña y medias también distribuidas Normales. Mezclas diferentes para cada clase.

6 — Optimizacion y Heurísticas

Yván Jesús Túpac Valdivia
Universidad Católica de San Pablo, Perú

El objetivo de una optimización global es encontrar los mejores elementos posibles x∗ de un conjunto X que siga un criterio F = { f1 , f2 , . . . , fn }. Estos criterios son expresados como funciones matemáticas f (x) que en el área de optimización se suelen denominar funciones objetivo o funciones de costo, que son definidas a continuación.

Definición. Función objetivo: Una función objetivo f : X → Y donde Y ⊆ R es una función matemática para ser optimizada.

La imagen Y de la función objetivo suele ser un subconjunto del conjunto de los números reales, o sea (Y ⊆ R).

Al dominio X de la función f se le denomina espacio de problema pudiendo ser representado (computacionalmente) por cualquier tipo de elemento, aunque principalmente como listas o vectores n-dimensionales, etc. Cabe destacar que las funciones objetivo no se limitan a ser expresiones matemáticas sino que pueden estar compuestas por ecuaciones o algoritmos complejos que para calcularse necesiten múltiples simulaciones. Así, la optimización global debe comprender todas las técnicas, analíticas y no analíticas que puedan usarse para encontrar los mejores elementos, x ∈ X de acuerdo a la función f ∈ F.

En resumidas cuentas, la optimización consiste en buscar y encontrar la solución ó soluciones optimas a un determinado problema, tomando en cuenta determinadas restricciones. Como ejemplos podemos citar:

- Aumentar la rentabilidad de un negocio.
- Reducir gastos, multas o pérdidas.
- Aumentar la eficacia de un determinado proceso.

Para poder encajar un problema cualquiera de la vida real como un problema de optimización, es necesario identificar las siguientes entidades:

1. El problema: sus características, restricciones y variables,
2. Las variables de decisión x = {x1 , . . . , xn } del problema, cuyos valores in[U+FB02]uyen en la solución,
3. La función f (x) que calcula la calidad de la solución – función objetivo,

4. Un método, algoritmo o heurística de búsqueda de soluciones,
5. El espacio de soluciones válidas X discreto o continuo del problema,
6. Los recursos computacionales necesarios para: implementar el proceso, la función de evaluación (o simulación), tratamiento de características del problema y selección de la mejor solución.

6.1 Definiciones en Optimización

Como fue mencionado antes, el objetivo del problema de optimización es encontrar las soluciones optimas a un determinado problema, es decir la mejor solución posible. Entonces vale la pena definir las características que hacen a una solución optima.

Definición Espacio de Problema: El espacio de problema para un problema de optimización, denotado por X, es el conjunto que contiene todos los elementos x que podrían ser la solución a encontrar.

La mayoría de veces, este espacio X está sujeto a:

1. Restricciones lógicas que indican elementos x que no pueden ser soluciones, como una división entre cero o un n´ mero negativo al sacar raiz cuadrada.

2. Restricciones prácticas que nos limitan el espacio de problema por cuestiones de tecnología. Un ejemplo son los tipos de variable de punto [U+FB02]otante: float o double que tienen precisión limitada.

Definición Candidato a solución: Un candidato a solución x es un elemento valido del conjunto X para un dado problema de optimización.

Definición Espacio de Soluciones: Se define el espacio de soluciones S como la unión de todas las soluciones de un problema de optimización.

$X* \subseteq S \subseteq X$
(6.1)

Como se ve en la ecuación, el espacio de soluciones contiene (pudiendo ser igual) al conjunto optimo global $X*$. Pueden haber soluciones válidas $x \in S$ que no son parte de X*, lo que suele ocurrir en optimización con restricciones

Definición Espacio de Búsqueda: El espacio de búsqueda de un problema de optimización, denotado por G, es el conjunto de todos los elementos $g \in G$ que pueden ser procesados por los operadores de búsqueda. Así G es una codificación del espacio de problema X.

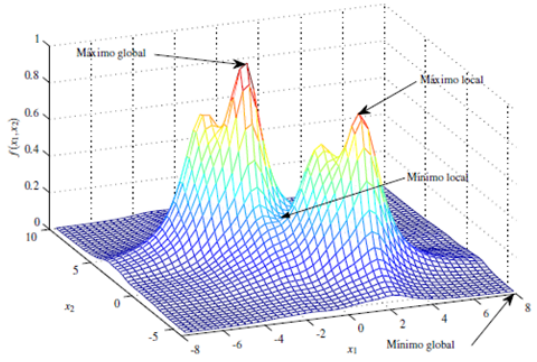
6.2 Funciones de único objetivo

Cuando se optimiza un único criterio f (x), un optimo es un máximo o un mínimo dependiendo de si se está maximizando o minimizando. Por ejemplo, en

problemas de mundo real (de manufactura), se busca por lo general minimizar tiempos, costos o pérdidas para un determinado proceso. Por otro lado, tratándose de negocios, se busca maximizar la rentabilidad o el valor económico.

La figura 6.1 muestra una función f definida en un espacio 2-dimensional X con elementos x = (x1 , x2) ∈ X. Se hace destaque en el gráfico a los óptimos locales y óptimos globales. Un optimo global es el optimo en todo el conjunto mientras que un optimo local es optimo solo en un subespacio de X.

Figura 6.1: Ejemplo de función f(x1,x2) con óptimo global y local



Máximo local: Sea una función de un objetivo f : X 7→ R, se define un máximo local xˆl ∈ X como un valor de entrada que cumple f (xˆl) ≥ f (x) para todo x dentro de una vecindad de xˆl.

Si f : X ⊆ Rn se puede decir que

∀xˆl , ∃ > 0 : f (xˆl) ≥ f (x), ∀x ∈ X, |x − xˆl | <
(6.2)

Mínimo local: Sea una función de un objetivo f : X 7→ R, se define un mínimo local x¯l ∈ X como un valor de entrada que cumple f (x¯l) ≤ f (x) para todo x dentro de una vecindad de x¯l .

Si f : X ⊆ Rn se puede decir que

∀x¯l, ∃ > 0 : f (x¯l) ≤ f (x), ∀x ∈ X, |x − x¯l| <
(6.3)

Óptimo local: Sea una función de un objetivo f : X → R, se define un óptimo local

x*l ∈ X como un valor que cumple con ser un máximo local o un mínimo local.

Máximo global: Sea una función de un objetivo f : X → R, se define Un máximo global

ˆx ∈X como un valor de entrada que cumple f (ˆx)≥ f (x),∀x ∈X

Mínimo global: Sea una función de un objetivo f : X → R, se define un mínimo global

∈X como un valor de entrada que cumple f ()≤ f (x),∀x ∈X

Óptimo global: Sea una función de un objetivo f : X 7→ R, se define un óptimo global

x ∈ X como un valor de entrada que cumple con ser un máximo o un mínimo global.

Es común encontrar más de un máximo o mínimo globales, inclusive dentro del dominio X de una función unidimensional f : X ⊆ Rn → R. Un ejemplo de esta situación es la función coseno cos(x) donde x = {x}, que tiene valores máximos globales en ˆx cuando ˆx = 2iπ y valores mínimos globales ¯x cuando ¯x = (2π+1)π.

La solución correcta sería un conjunto X* de entradas ´optimas y no un mínimo o máximo aislado. Además, el significado de óptimo depende del problema:

- En optimización con un objetivo significa un máximo o mínimo global.
- En optimización multi-objetivo existen varias estrategias para definir el óptimo.

Conjunto óptimo: Se define el conjunto óptimo X* como el conjunto que contiene todos los elementos óptimos.

Por lo general son varias y, muchas veces, infinitas soluciones óptimas; pero, dadas las limitaciones de computación, se pueden encontrar un subconjunto finito de soluciones óptimas. Así, se establece una diferencia entre el conjunto ´optimo X* ideal y el conjunto X de sub óptimos, que puede contener un óptimo global. Este conjunto X es lo que un algoritmo real de optimización puede encontrar. Entonces podemos afirmar que un algoritmo de optimización tiene como tarea:

1. Encontrar soluciones que sean lo mejor posibles para el problema, y,
2. que por su vez, ´estas sean lo más diferentes entre sí.

6.3 Optimización Clásica

En la optimización clásica u optimización numérica se busca encontrar el valor de las variables de una función univariada x que maximize o minimize una determinada función f (x). Este problema de optimización genérico está

definido de la siguiente manera:

Sea la función
f : X → R

x → f (x)
(6.4)

El problema de optimizaci´on consiste en hallar

Y* = máx(mín){ f (x)} (6.5)

sujeto a las siguientes condiciones:

g1(x) ≤ 0, i = 1, . . . , p
hj(x) = 0, j = 1, . . . ,n
(6.6)

donde:

X ⊆ R^n

es el espacio de problema que es un subespacio de R^n que contiene los puntos factibles, o sea puntos que satisfacen las restricciones del problema.

x ∈ X

es un punto en el espacio de problema definido como x = (x1, x2, . . . , xn), componentes x1, x2, . . . , xn son las denominadas variables de decisión del problema.

f (x) es la función objetivo, es decir la que se quiere optimizar.

Restricciones de dominio Definidas como los límites

ai ≤ xi ≤ bi,∀i = 1, . . . ,n

relacionados a los valores mínimos y máximos de las variables xi permitidos en el espacio X.

Restricciones de igualdad definidas como el conjunto de tamaño p

gi(x) = 0,∀i = 1, . . . , p

Restricciones de desigualdad definidas como el conjunto de tamaño q

hj(x) = 0,∀ j = 1, . . . ,q

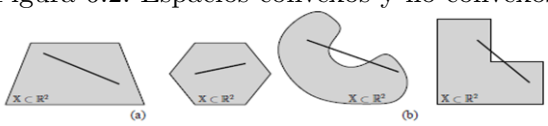
6.4 Convexidad

Espacio convexo: Sea el espacio de problema X. Se dice que X es convexo si para todo elemento x1,x2 ∈ X y para todo $\alpha \in [0,1]$ se cumple que

f (α x1+(1− α)x2) ≤ α f (x1)+(1− α) f (x2)
(6.7)

La interpretación de la ecuación 8.7 es la siguiente: X es convexo si para dos puntos cualesquiera x1,x2 ∈ X, el segmento rectilíneo que une estos puntos también pertenece al conjunto. En otras palabras, X es convexo si se puede ir de cualquier

Figura 6.2: Espacios convexos y no convexos



punto inicial a cualquier punto final en línea recta sin salir del espacio. Los conjuntos de la figura 6.2 (a) son convexos mientras que los de la figura 6.2 no lo son.

El objetivo de cualquier técnica de optimización es hallar el ´optimo global de cualquier problema. Lamentablemente, sólo en casos muy limitados se puede garantizar convergencia hacia el óptimo global. Por ejemplo, para problemas con espacios de búsqueda X convexos, las condiciones de Kuhn-Tucker (Kuhn and Tucker, 1951) son necesarias y suficientes para garantizar optimalidad global de un punto.

En problemas de optimización no lineal, estas condiciones no son suficientes. Por lo que en este caso, cualquier técnica usada puede encontrar óptimos locales sin garantizarse la convergencia hacia el óptimo global. Sólo se garantiza esta convergencia usándose métodos exhaustivos o está existe en tiempo infinito.

Existe una clase especial de problemas de optimización que son también de mucho interés, se trata del caso en que las variables de decisión son discretas, es decir el vector x está compuesto por valores xi ∈ N y x es un producto cartesiano o una permutación de valores xi. Este tipo de problemas, conocidos como optimización

combinatoria o programación entera, son de mucho interés en el área de Ciencias de Computación. cuyo ejemplo más conocido es el problema del vendedor viajero (Travelling Salesman Problem – TSP).

6.5 Técnicas clásicas de optimización

Existen muchas técnicas largamente conocidas y aplicadas para solucionar problemas de optimización, siempre que estos satisfagan ciertas características específicas, como por ejemplo, que f (x) sea una función lineal, o que sea unimodal1 o que las restricciones sean lineales.

La importancia de saber, al menos de la existencia de estas técnicas está en que si el problema a solucionar se ajusta a las exigencias que estas imponen, no se necesario usar heurísticas. Por ejemplo, si la función es lineal, el método Simplex siempre seguirá siendo la opción más viable.

6.5.1 Optimización Lineal – Método Simplex

La idea básica de la Optimización Lineal consiste en buscar una solución válida que mejore la función objetivo tomando como partida una solución válida inicial.

El problema de Programación Lineal consiste en minimizar el producto interno:

mínz = c^T x
(6.8)

sujeto a las siguientes condiciones:

Ax = b
x $\geq$ 0
(6.9)

donde:

Am×n; r(A) = m; b $\in R^m$; c $\in R^n$

Dado el conjunto S = {x : Ax = b,x $\geq$ 0}, cualquier punto xi $\in$S es una combinación lineal convexa de sus puntos extremos (o vértices) más una combinación lineal positiva de sus direcciones extremas (aristas).

6.5.2 Optimización no lineal

Para optimización no lineal, hay métodos directos como la búsqueda aleatoria y métodos indirectos como el método del gradiente conjugado (Rao, 1996).

Una de las principales limitaciones de las técnicas clásicas es justamente

que exigen información que no siempre está disponible. Por ejemplo, los métodos de gradiente necesitan que se calcule la primera derivada de la función objetivo. Otros métodos, como el de Newton exigen la segunda derivada. Por lo tanto, si la función objetivo no es diferenciable, estos métodos no podrían aplicarse. En muchos casos de mundo real, no hay ni siquiera una forma explícita de la función objetivo.

6.5.3 Método Steepest Descent

Un ejemplo de técnicas clásicas de optimización es el método del descenso empinado o steepest descent, propuesto originalmente por Cauchy en 1847. La idea de este método es escoger un punto xi cualquiera del espacio de búsqueda y moverse a lo largo de las direcciones de descenso más inclinado (dirección de gradiente fi) hasta encontrar el valor óptimo. El algoritmo 1 describe este método.

6.5.4 Método de Fletcher-Reeves (Gradiente Conjugado)

Que fue propuesto inicialmente por Hestenes & Stiefel en 1952, como una forma de solucionar sistemas de ecuaciones lineales derivadas de las condiciones estacionarias de una cuadrática.

Algoritmo 1: Algoritmo Steepest descent

```
escoger un punto x1
i = 1
repetir
        calcular $\nabla$fi
        encontrar la direccion si = $-$$\
           nabla$fi = $-$$\nabla$f (xi)
        determinar el incremento optimo $\
           lambda$i en la direccion si
        calcular xi+1 = xi +$\lambda$i si = xi
           $-$$\lambda$i $\nabla$fi
        i = i+1
hasta que xt+1 sea optimo
```

Se puede ver como una variante del método Steepest Descent, que usa el gradiente de una función para encontrar la dirección más prometedora de búsqueda. El algoritmo 2 describe el método de Fletcher-Reeves.

Algoritmo 2: Algoritmo de Gradiente Conjugado

```
escoger un punto arbitrario x1
calcular la direccion de busqueda s1 = $-$$\
   nabla$f (x1) = $-$$\nabla$f1
encontrar x2 = x1 +$\lambda$1s1
```

```
donde $\lambda$1 es el paso optimo en la
   direccion si
i = 2
repetir
        calcular $\nabla$fi = $\nabla$f (xi)
        calcular si = $-$$\nabla$fi +     $\
           frac{|\nabla fi |^2}{|\nabla fi
           -1|^2}$              si$-$1
        calcular xi+1 = xi + $\lambda$i si = xi
            $-$ $\lambda$i fi
        calcular $\lambda$i
        calcular el nuevo punto xi+1 = xi + $\
           lambda$i si
        i = i+1
hasta que xt+1 sea optimo
```

También hay otras técnicas que construyen soluciones parciales a un problema. Como ejemplos tenemos la Programación Dinámica (Bellman, 1957) y el método Branch & Bound.

En conclusión, se puede decir que, si la función a optimizarse se encuentra definida en forma algebraica, es importante intentar solucionarla usando técnicas clásicas antes de atacarla con cualquier heurística.

6.6 Técnicas Heurísticas de Optimización

6.6.1 Heurísticas

En Optimización Clásica, se asume que los problemas a tratarse deben cumplir algunas exigencias que garanticen la convergencia hacia el ´optimo global. Sin embargo, la mayoría de los problemas del mundo real no satisfacen estas exigencias.

Inclusive, muchos de estos problemas no pueden solucionarse usando un algoritmo con tiempo polinomial usando computadores determinísticos (conocidos como problemas NP, NP-Hard, NP-Complete), en los cuales el mejor algoritmo que se conoce requiere tiempo exponencial. De hecho, en muchas aplicaciones prácticas, ni siquiera es posible afirmar que existe una solución eficiente. Cuando enfrentamos espacios de búsqueda tan grandes como el problema del viajero (Travelling Salesman Problem - TSP), y que además los algoritmos más eficientes que existen para resolver el problema requieren tiempo exponencial, resulta obvio que las técnicas clásicas de búsqueda y optimización son insuficientes. Es entonces cuando recurrimos a las heurísticas.

Heurística: La palabra heurística se deriva del griego heuriskein, que significa encontrar o descubrir.

El significado del término ha variado históricamente. Algunos han usado el término como un antónimo de algorítmico.

Se denomina heurística a un proceso que puede resolver un cierto problema, pero que no ofrece ninguna garantía de lograrlo.

Las heurísticas fueron un área predominante en los orígenes de la Inteligencia Artificial. Actualmente, el término suele usarse como un adjetivo, refiriéndose a cualquier técnica que mejore el desempeño en promedio de la solución de un problema, aunque no mejore necesariamente el desempeño en el peor caso (Russell and Norvig, 2002).

Una definición más precisa y adecuada es la proporcionada por (Reeves, 1993).

"Una heurística es una técnica que busca soluciones buenas (es decir, casi óptimas) a un costo computacional razonable, aunque sin garantizar factibilidad u optimalidad de las mismas. En algunos casos, ni siquiera puede determinar qué tan cerca del ´optimo se encuentra una solución factible en particular."

Metaheurística: Una Metaheurística es un método que se aplica para resolver problemas genéricos. Combina funciones objetivo o heurísticas de una forma eficiente y abstracta que usualmente no dependen de la estructura del problema.

Algunos ejemplos de técnicas heur´ısticas y metaheurísticas aplicadas al problema de optimización son las siguientes:

6.6.2 Búsqueda Tabú

La Búsqueda Tabú (Tabu Search) (Glover and Laguna, 1998) en realidad es una meta-heurística, porque es un procedimiento que debe acoplarse a otra técnica, ya que no funciona por sí sola. La búsqueda tabú usa una memoria para guiar la búsqueda, de tal forma que algunas soluciones examinadas recientemente son memorizadas y tomadas como tabú (prohibidas) a la hora de tomar decisiones acerca del siguiente punto de búsqueda. La búsqueda tabú es determinista, aunque se le pueden agregar elementos probabilísticos.

6.6.3 Simulated Annealing

Está basado en el enfriamiento de los cristales (Kirkpatrick et al., 1983). El algoritmo requiere de una temperatura inicial, una final y una función de variación de la temperatura. Dicha función de variación es crucial para el buen desempeño del algoritmo y su definición es, por tanto, sumamente importante. Éste es un algoritmo probabilístico de búsqueda local.

Su principio está basado en el uso del algoritmo de Metropolis que aplicando Simulación Monte Carlo, calcula el cambio de energía que ocurre durante el

enfriamiento de un sistema físico.

Algoritmo de Metrópolis: Se genera una perturbación y se calcula el cambio de energía, si ´esta decrece, el nuevo estado es aceptado, caso contrario, la aceptación estará sujeta a un sorteo con la probabilidad de la Ecuación 8.10:

$$P(\delta E) = e^{\frac{-\delta E}{kt}} \tag{6.10}$$

El algoritmo 3 representa el proceso de Simulated Annealing

Algoritmo 3: Algoritmo Simulated Annealing(f)

```
Entrada: f : la funcion objetivo a minimizar
Dato: kmax : numero maximo de iteraciones
Dato: emax : energia maxima
Dato: xnew el nuevo elemento creado
Dato: x0 el mejor elemento conocido
Salida: x el mejor elemento encontrado
x$\leftarrow$x0 , e$\leftarrow$E(x)
xbest $\leftarrow$x , ebest $\leftarrow$e
k = 0
mientras k $<$ kmax y e $>$ emax hacer
        encontrar xnew = vecino(x)
        enew $\leftarrow$E(xnew)
        si P(e, enew,T(k/kmax) $>$ U(t)
           entonces
                x$\leftarrow$xnew
                e$\leftarrow$enew
        fin si
        si enew $<$ ebest entonces
                xbest $\leftarrow$xnew , ebest $
                   \leftarrow$enew
        fin si
        k = k+1
fin mientras
```

6.6.4 Hill Climbing

El algoritmo de Escalando la Colina o Hill Climbing (Russell and Norvig, 2002) es una técnica simple de búsqueda local y optimización aplicado para una función f de un único objetivo en un espacio de problema X. Haciendo iteraciones a partir de un punto inicial x = x0, que usualmente es la mejor solución conocida para el

problema, se obtiene un nuevo descendiente xnew en la vecindad de x. Si el individuo nuevo xnew es mejor que el predecesor x, lo reemplaza y así

sucesivamente. Este algoritmo no tiene retroceso ni lleva ningún tipo de registro histórico (aunque estos y otros aditamentos son susceptibles de ser incorporados). Es importante resaltar que Hill Climbing busca maximizar o minimizar la función f (x) donde x es discretizado, o sea x ∈ X ⊆ Np.

Se puede decir que Hill Climbing es una forma simple de búsqueda de la dirección de gradiente, por esto fácilmente puede quedar atrapado en óptimos locales. El algoritmo 4 representa al Hill Climbing

Algoritmo 4: Algoritmo Hill Climbing(f)

```
Requiere: f : la funcion objetivo a minimizar
Requiere: n : numero de iteraciones
        Dato: xnew el nuevo elemento creado
        Dato: x0 el mejor elemento conocido
        Salida: x el mejor elemento encontrado
        i = 0
        repetir
                x = x0
                Evaluar f (x)
                Encontrar un xnew = vecino(x)
                Evaluar xnew
                si f (xnew) $>$ f (x) entonces
                        x = xnew
                fin si
        i = i+1
        hasta que i $\geq$ n
```

6.7 Referencias

[Bellman, 1957]Bellman, R. E. (1957). Dynamic Programming. Princeton Univer-sity Press, Princeton NJ.

Glover and Laguna, 1998

Glover, F. and Laguna, M. (1998). Tabu Search. Kluwer Academic Publishers,Norwell Massachusetts.

Kirkpatrick et al., 1983

Kirkpatrick, S., Gelatt, J. C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. Science, 220:671–680.

Kuhn and Tucker, 1951

Kuhn, H. W. and Tucker, A. W. (1951). Nonlinear pro-gramming. In Neyman, J., editor, Proceedings of 2nd Berkeley Symposium, pages 481–492, Berkeley, CA. Berkeley: University of Calif ornia Press.

Rao, 1996

Rao, S. S. (1996). Engineering Optimization. Theory and Practice. John Wiley & Sons, third edition.

Reeves, 1993

Reeves, C. B. (1993). Modern Heuristic Techniques for Combinato-rial Problems. John Wiley & Sons, Great Britain.

Russell and Norvig, 2002

Russell, S. J. and Norvig, P. (2002). Artificial Intelli-gence. A Modern Approach. Prentice Hall„ Upper Saddle River, New Jersey, second edition.

7 — Algoritmos Evolutivos

Yván Jesús Túpac Valdivia
Universidad Católica de San Pablo,Perú, ytupac@ucsp.pe

En este capítulo se dan los conceptos fundamentales sobre computación evolutiva partiendo por la Optimización, los principios básicos de la Computación Evolutiva: principio basado en la naturaleza y en la evolución darwiniana, paradigmas de la computación evolutiva, Algoritmos genéticos canónicos, modelos evolutivos para optimización numerica y modelos evolutivos para optimización combinatoria

7.1 Optimizacion y Heurísticas

7.1.1 Optimización

Uno de los principios fundamentales en la naturaleza es la búsqueda de un estado óptimo. Este principio se observa desde el microcosmos, cuando los átomos intentan formar enlaces de mínima energía de sus electrones [**Pauling1960**]; a nivel molecular se observa durante el proceso de congelamiento cuando moléculas se convierten en cuerpos sólidos al encontrar estructuras cristalinas de energía óptima. En ambos casos, estos procesos son guiados a estos estados de energía mínima sólo por las leyes físicas.
Por otro lado está el principio biológico de *sobrevivencia de los más aptos* [**Spencer1960**] que, junto a la evolución biológica, conllevan a la mejor adaptación de las especies a su ambiente. En este caso, un *óptimo local* es una especie que domina todas las otras a su alrededor. El *Homo Sapiens* ha alcanzado este nivel al dominar hormigas, bacterias, moscas, cucarachas y toda clase de criaturas existentes en nuestro planeta.
A lo largo de nuestra historia los seres humanos, venimos *buscando la perfección* en muchos aspectos. Por un lado deseamos alcanzar el máximo grado de bienestar haciendo el menor esfuerzo, en un aspecto económico, se busca maximizar ventas y rentabilidad haciendo que los costos sean lo mínimo posibles. Así, podemos afirmar que la optimización es un aspecto de la ciencia que se extiende hasta inclusive, nuestra vida diaria [**Neumaier2006**].
A partir de estas ideas se puede inferir que es importante generalizar y comprender que, por detrás de estos fenómenos existe un formalismo matemático que estudia toda esta área: *Optimización Global*, que es una rama de la matemática aplicada y análisis numérico cuyo foco es claro... Optimización.
El objetivo de una optimización global es encontrar los mejores elementos posibles $\mathbf{x}^*$ de un conjunto $\mathbb{X}$ que siga un criterio $F = \{f_1, f_2, \ldots, f_n\}$. Estos criterios son expresados como funciones matemáticas $f(\mathbf{x})$ que en el área de

optimización se suelen denominar *funciones objetivo* o funciones de costo, que son definidas a continuación.

Definition 7.1.1 Función objetivo: Una función objetivo $f : \mathbb{X} \mapsto Y$ donde $Y \subseteq \mathbb{R}$ es una función matemática para ser optimizada.

La imagen Y de la función objetivo suele ser un subconjunto del conjunto de los números reales, o sea ($Y \subseteq \mathbb{R}$).
Al dominio $\mathbb{X}$ de la función f se le denomina espacio de problema pudiendo ser representado (computacionalmente) por cualquier tipo de elemento, aunque principalmente como listas o vectores n-dimensionales, etc. Cabe destacar que las funciones objetivo no se limitan a ser expresiones matemáticas sino que pueden estar compuestas por ecuaciones o algoritmos complejos que para calcularse necesiten múltiples simulaciones. Así, la optimización global debe comprender todas las técnicas, analíticas y no analíticas que puedan usarse para encontrar los mejores elementos, $\mathbf{x} \in \mathbb{X}$ de acuerdo a la función $f \in F$.
En resumidas cuentas, la optimización consiste en buscar y encontrar la solución o soluciones óptimas a un determinado problema, tomando en cuenta determinadas restricciones. Como ejemplos podemos citar:

- Aumentar la rentabilidad de un negocio.
- Reducir gastos, multas o pérdidas.
- Aumentar la eficacia de un determinado proceso

Para poder encajar un problema cualquiera de la vida real como un *problema de optimización*, es necesario identificar las siguientes entidades:

1. El problema: sus características, restricciones y variables,
2. Las variables de decisión $\mathbf{x} = \{x_1, \ldots, x_n\}$ del problema, cuyos valores influyen en la solución,
3. La función $f(\mathbf{x})$ que calcula la calidad de la solución – *función objetivo*,
4. Un método, algoritmo o *heurística* de búsqueda de soluciones,
5. El espacio de soluciones válidas $\mathbb{X}$ discreto o contínuo del problema,
6. Los recursos computacionales necesarios para: implementar el proceso, la función de evaluación (o simulación), tratamiento de características del problema y selección de la mejor solución.

7.1.2 Definiciones en Optimización

Como fue mencionado antes, el objetivo del *problema de optimización* es encontrar las soluciones **óptimas** a un determinado problema, es decir la mejor solución posible. Entonces vale la pena definir las características que hacen a una solución *óptima*.

Definition 7.1.2 Espacio de Problema: El espacio de problema para un problema de optimización, denotado por $\mathbb{X}$, es el conjunto que contiene todos los elementos $\mathbf{x}$ que podrían ser la solución a encontrar.

La mayoría de veces, este espacio $\mathbb{X}$ está sujeto a:

1. *Restricciones logicas* que indican elementos $\mathbf{x}$ que no pueden ser soluciones, como una división entre cero o un número negativo al sacar raiz cuadrada.
2. *Restricciones prácticas* que nos limitan el espacio de problema por cuestiones de tecnología. Un ejemplo son los tipos de variable de punto flotante `float` o `double` que tienen precisión limitada.

Definition 7.1.3 Candidato a solución: Un candidato a solución $\mathbf{x}$ es un elemento valido del conjunto $\mathbb{X}$ para un dado problema de optimización.

Definition 7.1.4 Espacio de Soluciones: Se define el espacio de soluciones $\mathbb{S}$ como la unión de todas las soluciones de un problema de optimización.

$$\mathbf{X}^* \subseteq \mathbb{S} \subseteq \mathbb{X} \tag{7.1}$$

Como se ve en la ecuación 7.1, el espacio de soluciones contiene (pudiendo ser igual) al conjunto óptimo global $\mathbf{X}^*$. Pueden haber soluciones válidas $\mathbf{x} \in \mathbb{S}$ que no son parte de $\mathbf{X}^*$, lo que suele ocurrir en optimización con restricciones.

Definition 7.1.5 Espacio de Búsqueda: El espacio de búsqueda de un problema de optimización, denotado por $\mathbb{G}$, es el conjunto de todos los elementos $\mathbf{g} \in \mathbb{G}$ que pueden ser procesados por los operadores de búsqueda. Así $\mathbb{G}$ es una codificación del espacio de problema $\mathbb{X}$.

7.1.3 Funciones de único objetivo

Cuando se optimiza un único criterio $f(\mathbf{x})$, un óptimo es un máximo o un mínimo dependiendo de si se está maximizando o minimizando. Por ejemplo, en problemas de mundo real (de manufactura), se busca por lo general *minimizar* tiempos, costos o pérdidas para un determinado proceso. Por otro lado, tratántose de negocios, se busca *maximizar* la rentabilidad o el valor económico.
La figura 7.1 muestra una función f definida en un espacio 2-dimensional $\mathbb{X}$ con elementos $\mathbf{x} = (x_1, x_2) \in \mathbb{X}$. Se hace destaque en el gráfico a los óptimos locales y óptimos globales. Un óptimo global es el óptimo en todo el conjunto mientras que un óptimo local es óptimo sólo en un subespacio de $\mathbb{X}$.

Definition 7.1.6 Máximo local: Sea una función de un objetivo $f : \mathbb{X} \mapsto \mathbb{R}$, se define un máximo local $\hat{x}_l \in \mathbb{X}$ como un valor de entrada que cumple $f(\hat{x}_l) \geq f(x)$ para todo x dentro de una vecindad de $\hat{x}_l$.

Si $f : \mathbb{X} \subseteq \mathbb{R}^n$ se puede decir que

$$\forall \hat{x}_l, \exists \epsilon > 0 : f(\hat{x}_l) \geq f(x), \forall x \in \mathbb{X}, |x - \hat{x}_l| < \epsilon \tag{7.2}$$

Definition 7.1.7 Mínimo local: Sea una función de un objetivo $f : \mathbb{X} \mapsto \mathbb{R}$, se define un mínimo local $\bar{x}_l \in \mathbb{X}$ como un valor de entrada que cumple $f(\bar{x}_l) \leq f(x)$ para todo x dentro de una vecindad de $\bar{x}_l$.

Si $f : \mathbb{X} \subseteq \mathbb{R}^n$ se puede decir que

$$\forall \bar{x}_l, \exists \epsilon > 0 : f(\bar{x}_l) \leq f(x), \forall x \in \mathbb{X}, |x - \bar{x}_l| < \epsilon \tag{7.3}$$

Definition 7.1.8 Óptimo local: Sea una función de un objetivo $f : \mathbb{X} \mapsto \mathbb{R}$, se define un óptimo local $x_l^* \in \mathbb{X}$ como un valor que cumple con ser un máximo local o un mínimo local.

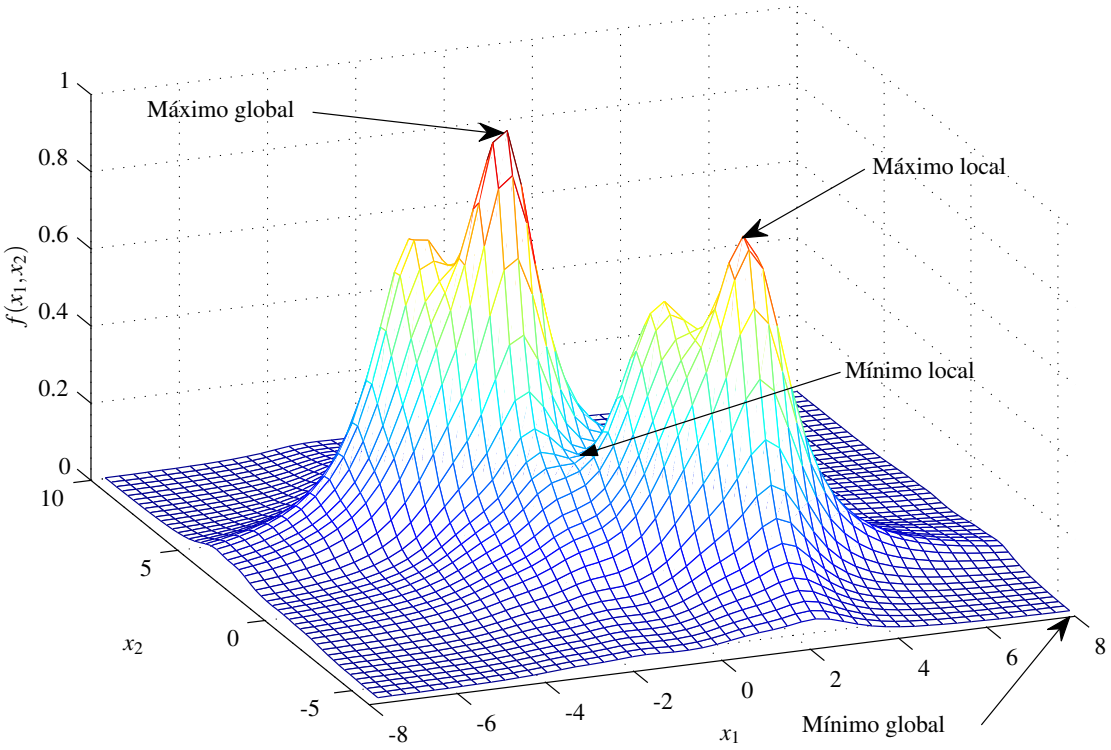


Figura 7.1: Ejemplo de función $f(x_1, x_2)$ con óptimo global y local

Definition 7.1.9 Máximo global: Sea una función de un objetivo $f : \mathbb{X} \mapsto \mathbb{R}$, se define un máximo global $\hat{\mathbf{x}} \in \mathbb{X}$ como un valor de entrada que cumple $f(\hat{\mathbf{x}}) \geq f(x), \forall x \in \mathbb{X}$

Definition 7.1.10 Mínimo global: Sea una función de un objetivo $f : \mathbb{X} \mapsto \mathbb{R}$, se define un mínimo global $\bar{\mathbf{x}} \in \mathbb{X}$ como un valor de entrada que cumple $f(\bar{\mathbf{x}}) \leq f(x), \forall x \in \mathbb{X}$

Definition 7.1.11 Óptimo global: Sea una función de un objetivo $f : \mathbb{X} \mapsto \mathbb{R}$, se define un óptimo global $\mathbf{x}^* \in \mathbb{X}$ como un valor de entrada que cumple con ser un máximo o un mínimo global.

Es común encontrar más de un máximo o mínimo globales, inclusive dentro del dominio $\mathbb{X}$ de una función unidimensional $f : \mathbb{X} \subseteq \mathbb{R}^n \mapsto \mathbb{R}$. Un ejemplo de esta situación es la función coseno $\cos(\mathbf{x})$ donde $\mathbf{x} = \{x\}$, que tiene valores máximos globales en $\hat{\mathbf{x}}$ cuando $\hat{\mathbf{x}} = 2i\pi$ y valores mínimos globales $\bar{\mathbf{x}}$ cuando $\bar{\mathbf{x}} = (2\pi + 1)\pi$. La solución correcta sería un conjunto $\mathbf{X}^*$ de entradas óptimas y no un mínimo o máximo aislado. Además, el significado de *óptimo* depende del problema:

- En optimización con un objetivo significa un máximo o mínimo global.
- En optimización multi-objetivo existen varias estrategias para definir el óptimo.

Definition 7.1.12 Conjunto óptimo: Se define el conjunto óptimo $\mathbf{X}^*$ como el conjunto que contiene todos los elementos óptimos.

Por lo general son varias y, muchas veces, infinitas soluciones óptimas; pero,

dadas las limitaciones de computación, se pueden encontrar un subconjunto finito de soluciones óptimas. Así, se establece una diferencia entre el conjunto óptimo $\mathbf{X}^*$ ideal y el conjunto X de subóptimos, que puede contener un óptimo global. Este conjunto X es lo que un algoritmo real de optimización puede encontrar. Entonces podemos afirmar que un algoritmo de optimización tiene como tarea:

1. Encontrar soluciones que sean lo mejor posibles para el problema, y,
2. que por su vez, éstas sean lo más diferentes entre sí.

7.1.4 Optimización Clásica

En la optimización clásica u *optimización numérica* se busca encontrar el valor de las variables de una función univariada $\mathbf{x}$ que maximize o minimize una determinada función $f(\mathbf{x})$. Este problema de optimización genérico está definido de la siguiente manera:
Sea la función

$$\begin{array}{rccc} f: & \mathbb{X} & \mapsto & \mathbb{R} \\ & \mathbf{x} & \mapsto & f(\mathbf{x}) \end{array} \tag{7.4}$$

El problema de optimización consiste en hallar

$$Y^* = \text{máx(mín)}\{f(\mathbf{x})\} \tag{7.5}$$

sujeto a las siguientes condiciones:

$$\begin{array}{l} g_1(\mathbf{x}) \leq 0, \quad i = 1,\ldots,p \\ h_j(\mathbf{x}) = 0, \quad j = 1,\ldots,n \end{array} \tag{7.6}$$

donde:

$\mathbb{X} \subseteq \mathbb{R}^n$ es el espacio de problema que es un subespacio de $\mathbb{R}^n$ que contiene los puntos factibles, o sea puntos que satisfacen las restricciones del problema.

$\mathbf{x} \in \mathbb{X}$ es un punto en el espacio de problema definido como $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, cuyos componentes $x_1, x_2, \ldots, x_n$ son las denominadas variables de decisión del problema.

$f(\mathbf{x})$ es la función objetivo, es decir la que se quiere optimizar.

Definition 7.1.13 Restricciones de dominio Definidas como los límites

$$a_i \leq x_i \leq b_i, \forall i = 1,\ldots,n$$

relacionados a los valores mínimos y máximos de las variables x_i permitidos en el espacio $\mathbb{X}$.

Definition 7.1.14 Restricciones de igualdad definidas como el conjunto de tamaño p

$$g_i(\mathbf{x}) = 0, \forall i = 1,\ldots,p$$

Definition 7.1.15 Restricciones de desigualdad definidas como el conjunto de tamaño q

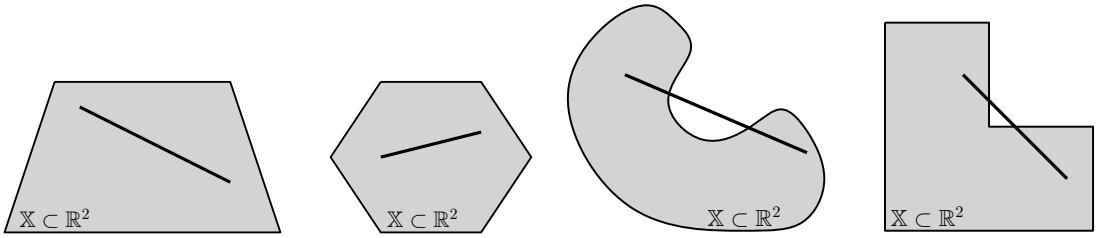
$$h_j(\mathbf{x}) = 0, \forall j = 1,\ldots,q$$

Convexidad

Definition 7.1.16 Espacio convexo: Sea el espacio de problema $\mathbb{X}$. Se dice que $\mathbb{X}$ es convexo si para todo elemento $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{X}$ y para todo $\alpha \in [0,1]$ se cumple que

$$f(\alpha \mathbf{x}_1 + (1-\alpha)\mathbf{x}_2) \leq \alpha f(\mathbf{x}_1) + (1-\alpha) f(\mathbf{x}_2) \tag{7.7}$$

La interpretación de la ecuación 7.7 es la siguiente: $\mathbb{X}$ es convexo si para dos puntos cualesquiera $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{X}$, el segmento rectilíneo que une estos puntos también pertenece al conjunto. En otras palabras, $\mathbb{X}$ es convexo si se puede ir de cualquier punto inicial a cualquier punto final en línea recta sin salir del espacio. Los conjuntos de la figura 7.2-a son convexos mientras que los de la figura 7.2-b no lo son.



(a) (b)

Figura 7.2: Espacios convexos y no convexos

El objetivo de cualquier técnica de optimización es hallar el óptimo global de cualquier problema. Lamentablemente, sólo en casos muy limitados se puede garantizar convergencia hacia el óptimo global. Por ejemplo, para problemas con espacios de búsqueda $\mathbb{X}$ convexos, las condiciones de Kuhn-Tucker [**KuhnTucker1951**] son necesarias y suficientes para garantizar optimalidad global de un punto.
En problemas de optimización no lineal, estas condiciones no son suficientes. Por lo que en este caso, cualquier técnica usada puede encontrar óptimos locales sin garantizarse la convergencia hacia el óptimo global. Sólo se garantiza esta convergencia usándose métodos exhaustivos o ésta existe en tiempo infinito.
Existe una clase especial de problemas de optimización que son también de mucho interés, se trata del caso en que las variables de decisión son discretas, es decir el vector $\mathbf{x}$ está compuesto por valores $x_i \in \mathbb{N}$ y $\mathbf{x}$ es un producto cartesiano o una permutación de valores x_i. Este tipo de problemas, conocidos como optimización combinatoria o programación entera, son de mucho interés en el área de Ciencias de Computación. cuyo ejemplo más conocido es el problema del vendedor viajero (*Travelling Salesman Problem* – TSP).

7.1.5 Técnicas clásicas de optimización

Existen muchas técnicas largamente conocidas y aplicadas para solucionar problemas de optimización, siempre que estos satisfagan ciertas características específicas, como por ejemplo, que $f(\mathbf{x})$ sea una función lineal, o que sea unimodal[1] o que las restricciones sean lineales.
La importancia de saber, al menos de la existencia de estas técnicas está en que si el problema a solucionar se ajusta a las exigencias que éstas imponen,

[1] Que tenga un único máximo o mínimo que consecuentemente es el óptimo global.

no se necesario usar heurísticas. Por ejemplo, si la función es lineal, el método *Simplex* siempre seguira siendo la opción más viable.

Optimización Lineal – Metodo *Simplex*
La idea básica de la Optimización Lineal consiste en buscar una solución válida que mejore la función objetivo tomando como partida una solución válida inicial.
El problema de *Programación Lineal* consiste en minimizar el producto interno:

$$\text{mín}\, z = \mathbf{c}^T \mathbf{x} \tag{7.8}$$

sujeto a las siguientes condiciones:

$$\begin{array}{rcl} \mathbf{A}\mathbf{x} & = & \mathbf{b} \\ \mathbf{x} & \geq & 0 \end{array} \tag{7.9}$$

donde:

$$\mathbf{A}_{m\times n};\quad r(\mathbf{A}) = m;\quad \mathbf{b} \in \mathbb{R}^m;\quad \mathbf{c} \in \mathbb{R}^n$$

Dado el conjunto $\mathcal{S} = \{\mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq 0\}$, cualquier punto $\mathbf{x}_i \in \mathcal{S}$ es una combinación lineal convexa de sus puntos extremos (o vértices) más una combinación lineal positiva de sus direcciones extremas (aristas).

Optimización no lineal
Para optimización no lineal, hay métodos directos como la búsqueda aleatoria y métodos indirectos como el método del gradiente conjugado [**Singiresu1996**]
Una de las principales limitaciones de las técnicas clásicas es justamente que exigen información que no siempre está disponible. Por ejemplo, los métodos de gradiente necesitan que se calcule la primera derivada de la función objetivo. Otros métodos, como el de Newton exigen la segunda derivada. Por lo tanto, si la función objetivo no es diferenciable, estos métodos no podrían aplicarse. En muchos casos de mundo real, no hay ni siquiera una forma explícita de la función objetivo.

Método *Steepest Descent*
Un ejemplo de técnicas clásicas de optimización es el método del descenso empinado o *steepest descent*, propuesto originalmente por Cauchy en 1847. La idea de este método es escoger un punto $\mathbf{x}_i$ cualquiera del espacio de búsqueda y moverse a lo largo de las direcciones de descenso más inclinado (dirección de gradiente ∇f_i) hasta encontrar el valor óptimo. El algoritmo 1 describe este método.

Algorithm 1 Algoritmo *Steepest descent*

escoger un punto $\mathbf{x}_1$ $i = 1$ **calcular** ∇f_i **encontrar** la dirección $\mathbf{s}_i = -\nabla f_i = -\nabla f(\mathbf{x}_i)$ **determinar** el incremento óptimo λ_i^* en la dirección $\mathbf{s}_i$ **calcular** $\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda_i^* \mathbf{s}_i = \mathbf{x}_i - \lambda_i^* \nabla f_i$ $i = i + 1$ $\mathbf{x}_{t+1}$ sea óptimo

Método de Fletcher-Reeves (Gradiente Conjugado)
Que fue propuesto inicialmente por Hestenes & Stiefel en 1952, como una forma de solucionar sistemas de ecuaciones lineales derivadas de las condiciones estacionarias de una cuadrática. Se puede ver como una variante del

método *Steepest Descent*, que usa el gradiente de una función para encontrar la dirección más prometedora de búsqueda. El algoritmo 2 describe el método de Fletcher-Reeves.

Algorithm 2 Algoritmo de Gradiente Conjugado

escoger un punto arbitrario $\mathbf{x}_1$ **calcular** la dirección de búsqueda $\mathbf{s}_1 = -\nabla f(\mathbf{x}_1) = -\nabla f_1$ **encontrar** $\mathbf{x}_2 = \mathbf{x}_1 + \lambda_1^* \mathbf{s}_1$ donde λ_1^* es el paso óptimo en la dirección $\mathbf{s}_i$ $i = 2$ **calcular** $\nabla f_i = \nabla f(\mathbf{x}_i)$ **calcular** $\mathbf{s}_i = -\nabla f_i + \frac{|\nabla f_i|^2}{|\nabla f_{i-1}|^2}\mathbf{s}_{i-1}$ **calcular** $\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda_i^* \mathbf{s}_i = \mathbf{x}_i - \lambda_i^* \nabla f_i$ **calcular** λ_i^* **calcular** el nuevo punto $\mathbf{x}_{i+1} = \mathbf{x}_i + \lambda_i^* \mathbf{s}_i$ $i = i + 1$ $\mathbf{x}_{t+1}$ sea óptimo

También hay otras técnicas que contruyen soluciones parciales a un problema. Como ejemplos tenemos la Programación Dinámica [**Bellman57**] y el método *Branch & Bound*

En conclusión, se puede decir que, si la función a optimizarse se encuentra definida en forma algebraica, es importante intentar solucionarla usando técnicas clásicas antes de atacarla con cualquier heurística.

7.1.6 Técnicas Heurísticas de Optimización

Heurísticas

En Optimización Clásica, se asume que los problemas a tratarse deben cumplir algunas exigencias que garantizen la convergencia hacia el óptimo global. Sin embargo, la mayoría de los problemas del mundo real no satisfacen estas exigencias. Inclusive, muchos de estos problemas no pueden solucionarse usando un algoritmo con tiempo polinomial usando computadores determinísticos (conocidos como problemas NP, NP-*Hard*, NP-*Complete*), en los cuales el mejor algoritmo que se conoce requiere tiempo exponencial. De hecho, en muchas aplicaciones prácticas, ni siquiera es posible afirmar que existe una solución eficiente.

Cuando enfrentamos espacios de búsqueda tan grandes como el problema del viajero (*Travelling Salesman Problem* - TSP), y que además los algoritmos más eficientes que existen para resolver el problema requieren tiempo exponencial, resulta obvio que las técnicas clásicas de búsqueda y optimización son insuficientes. Es entonces cuando recurrimos a las *heurísticas*.

Definition 7.1.17 Heurística: La palabra *heurística* se deriva del griego *heuriskein*, que significa *encontrar* o *descubrir*.

El significado del término ha variado históricamente. Algunos han usado el término como un antónimo de *algorítmico*.

Se denomina heurística a un proceso que puede resolver un cierto problema, pero que no ofrece ninguna garantía de lograrlo.

Las heurísticas fueron un área predominante en los orígenes de la Inteligencia Artificial. Actualmente, el término suele usarse como un adjetivo, refiriéndose a cualquier técnica que mejore el desempeño en promedio de la solución de un problema, aunque no mejore necesariamente el desempeño en el peor caso [**Stuart2002**].

Una definición más precisa y adecuada es la proporcionada por [**Reeves1993**].

> *"Una heurística es una técnica que busca soluciones buenas (es decir, casi óptimas) a un costo computacional razonable, aunque sin garantizar*

factibilidad u optimalidad de las mismas. En algunos casos, ni siquiera puede determinar qué tan cerca del óptimo se encuentra una solución factible en particular."

Definition 7.1.18 Metaheurística: Una Metaheurística es un método que se aplica para resolver problemas genéricos. Combina funciones objetivo o heurísticas de una forma eficiente y abstracta que usualmente no dependen de la estructura del problema.

Algunos ejemplos de técnicas heurísticas y metaheurísticas aplicadas al problema de optimización son las siguientes:

Búsqueda Tabú

La Búsqueda Tabú (*Tabu Search*) [**GloverLaguna1998**] en realidad es una meta-heurística, porque es un procedimiento que debe acoplarse a otra técnica, ya que no funciona por sí sola. La búsqueda tabú usa una *memoria* para guiar la búsqueda, de tal forma que algunas soluciones examinadas recientemente son *memorizadas* y tomadas como tabú (prohibidas) a la hora de tomar decisiones acerca del siguiente punto de búsqueda. La búsqueda tabú es determinista, aunque se le pueden agregar elementos probabilísticos.

Simulated Annealing

Está basado en el *enfriamiento de los cristales* [**Kirkpatrick1983**]. El algoritmo requiere de una temperatura inicial, una final y una función de variación de la temperatura. Dicha función de variación es crucial para el buen desempeño del algoritmo y su definición es, por tanto, sumamente importante. Éste es un algoritmo probabilístico de búsqueda local.
Su principio está basado en el uso del algoritmo de Metropolis que aplicando Simulación Monte Carlo, calcula el cambio de energía que ocurre durante el enfriamiento de un sistema físico.

Definition 7.1.19 Algoritmo de Metrópolis: Se genera una perturbación y se calcula el cambio de energía, si ésta decrece, el nuevo estado es aceptado, caso contrario, la aceptación estará sujeta a un sorteo con la probabilidad de la Ecuación 7.10:

$$P(\partial E) = e^{\frac{-\partial E}{kt}} \tag{7.10}$$

El algoritmo 3 representa el proceso de *Simulated Annealing*

Algorithm 3 Algoritmo *Simulated Annealing*(f)

Entrada: f : la función objetivo a minimizar **Dato:** k_{max} : número máximo de iteraciones **Dato:** e_{max} : energía máxima **Dato:** $\mathbf{x}_{new}$ el nuevo elemento creado **Dato:** $\mathbf{x}_0$ el mejor elemento conocido **Salida:** $\mathbf{x}$ el mejor elemento encontrado $\mathbf{x} \leftarrow \mathbf{x}_0, e \leftarrow E(\mathbf{x})$ $\mathbf{x}_{best} \leftarrow \mathbf{x}, e_{best} \leftarrow e$ $k = 0$ $k < k_{max}$ y $e > e_{max}$ **encontrar** $\mathbf{x}_{new} =$ vecino($\mathbf{x}$) $e_{new} \leftarrow E(\mathbf{x}_{new})$ $P(e, e_{new}, T(k/k_{max}) > \mathbf{U}(t)$ $\mathbf{x} \leftarrow \mathbf{x}_{new}$ $e \leftarrow e_{new}$ $e_{new} < e_{best}$ $\mathbf{x}_{best} \leftarrow \mathbf{x}_{new}, e_{best} \leftarrow e_{new}$ $k = k + 1$

Hill Climbing

El algoritmo de Escalando la Colina o *Hill Climbing* [**Stuart2002**] es una técnica simple de búsqueda local y optimización aplicado para una función f de un único objetivo en un espacio de problema $\mathbb{X}$. Haciendo iteraciones a

partir de un punto inicial $\mathbf{x} = \mathbf{x}_0$, que usualmente es la mejor solución conocida para el problema, se obtiene un nuevo descendiente $\mathbf{x}_{new}$ en la vecindad de $\mathbf{x}$. Si el individuo nuevo $\mathbf{x}_{new}$ es mejor que el predecesor $\mathbf{x}$, lo reemplaza y así sucesivamente. Este algoritmo no tiene retroceso ni lleva ningún tipo de registro histórico (aunque estos y otros aditamentos son susceptibles de ser incorporados). Es importante resaltar que *Hill Climbing* busca maximizar o minimizar la función $f(\mathbf{x})$ donde $\mathbf{x}$ es discretizado, o sea $\mathbf{x} \in \mathbb{X} \subseteq \mathbb{N}^p$.
Se puede decir que *Hill Climbing* es una forma simple de búsqueda de la dirección de gradiente, por esto fácilmente puede quedar atrapado en óptimos locales. El algoritmo 4 representa al *Hill Climbing*

Algorithm 4 Algoritmo *Hill Climbing*(f)

f : la función objetivo a minimizar n : número de iteraciones **Dato:** $\mathbf{x}_{new}$ el nuevo elemento creado **Dato:** $\mathbf{x}_0$ el mejor elemento conocido **Salida:** $\mathbf{x}$ el mejor elemento encontrado $i = 0$ $\mathbf{x} = \mathbf{x}_0$ **Evaluar** $f(\mathbf{x})$ **Encontrar** un $\mathbf{x}_{new}$ = vecino($\mathbf{x}$) **Evaluar** $\mathbf{x}_{new}$ $f(\mathbf{x}_{new}) > f(\mathbf{x})$ $\mathbf{x} = \mathbf{x}_{new}$ $i = i + 1$ $i \geq n$

7.2 Conceptos Básicos de Algoritmo Evolutivo

7.2.1 Algoritmos Evolutivos

Definition 7.2.1 Algoritmos Evolutivos (EA) Son algoritmos metaheurísticos basados en una población de individuos que emplean mecanismos biológicamente inspirados como la mutación, recombinación, selección natural y supervivencia de los más aptos para ir iterativamente ajustando o refinando un conjunto de soluciones.

Una ventaja de los Algoritmos Evolutivos con respecto a otros métodos de optimización es su característica de *black box*, es decir tener poco conocimiento y pocas suposiciones sobre la función objetivo a optimizar. Inclusive, la definición de la función objetivo exige menos conocimiento de la estructura del espacio de problema que el modelamiento de una heurística factible para el problema. Adicionalmente, los Algoritmos Evolutivos tienen una calidad de respuesta consistente para muchas clases de problemas.
Así, cuando se trata de Computación Evolutiva o Algoritmos Evolutivos, debemos entender que existe un conjunto de técnicas y metodologías que la componen, todas ellas con inspiración biológica en la evolución Neo-Darwiniana.

7.2.2 Conceptos usados en Computación Evolutiva

A raíz de la definicion 7.2.1, son varios los conceptos que deben ser bien entendidos ya que son comúnmente empleados en Computación Evolutiva, que se describen a continuación.

Definition 7.2.2 Individuo Es una solución propuestas en la población, sin ninguna alteración. Denotado en este texto como $\mathbf{x}$

Definition 7.2.3 Cromosoma Se refiere a una estructura de datos que contiene una cadena de parámetros de diseño (o genes). Esta estructura se puede almacenar computacionalmente de diversas formas: cadena de bits,

array de números enteros, o números reales. En este texto un cromosoma se denota como $\mathbf{g}$.

Definition 7.2.4 Gene Que es una subsección de un cromosoma que usualmente codifica el valor de uno de los parámetros del problema, el gene que codifica el $i-$ésimo parámetro es denotado por g_i.

Definition 7.2.5 Población Que es el conjunto de cromosomas que serán tratados en el proceso evolutivo. Una población es denotada como $\mathbf{X}$.

Definition 7.2.6 Generación Es una iteración que consiste en calcular la medida de aptitud de todos los individuos de una población $\mathbf{X}$ existente para después obtener una siguiente población a partir de un proceso de selección y operaciones de reproducción genética. Para una generación t su población es $\mathbf{X}_t$.

Definition 7.2.7 Genotipo Es la codificación utilizada en el cromosoma, para los parámetros del problema a solucionarse. Por ejemplo tenemos: números binarios, ternarios, enteros, reales, el conjunto de posibles valores del genotipo conforma el espacio de búsqueda $\mathbb{G}$.

Definition 7.2.8 Fenotipo Es el resultado de decodificar un cromosoma, es decir, los valores obtenidos pasan de la representación genética $\mathbf{g} \in \mathbb{G}$ (fenotipo) al espacio de problema $\mathbf{x} \in \mathbb{X}$.

Definition 7.2.9 Función de evaluación Que es una medida que indica la calidad del individuo en el ambiente. Por ejemplo, siendo $f(\mathbf{x}) = x^2$ la función de evaluación y $\mathbf{g}_k \to \mathbf{x}_k = \{x\} = 9$, entonces la evaluación del individuo es $f(9) = 81$.

Definition 7.2.10 Aptitud Es una transformación g aplicada a la función de evaluación $f(\mathbf{x})$ para medir la oportunidad que un individuo $\mathbf{x}_k \in \mathbf{X}$ tiene para reproducirse cuando se aplica la selección. En muchos casos, la función de aptitud coincide con la función de evaluación, es decir que $g(f(\mathbf{x})) = f(\mathbf{x})$. Cabe destacar que la evaluación de un individuo $f(\mathbf{x}_k)$ no depende de la evaluación de los demás individuos de la población $f(\mathbf{x}_j) \in \mathbf{X}, j \neq k$, pero la aptitud g de un individuo k sí está definida con respecto a los demás miembros de la población.

Definition 7.2.11 Alelo Que es cada valor posible que un gene puede adquirir. Estos dependen del espacio de búsqueda $\mathbb{G}$ definido para el genotipo y dependiendo de la codificación utilizada. Si se usa codificación binaria, entonces el espacio de búsqueda $\mathbb{G} \equiv \mathbb{B}^l$ permitiendo alelos $a_i \in \{0, 1\}$.

Definition 7.2.12 Operador de reproducción Es todo aquel mecanismo que influencia la forma como la información genética se transfiere de padres a hijos. Estos operadores tienen dos categorías conocidas:

- Operadores de cruce (sexuales)
- Operadores de mutación (asexuales)

Adicionalmente, en algunos modelos más genéricos se emplean otros tipos de operadores como:

Operadores panmíticos (un padre que se reproduce con varias parejas)
- Operadores que emplean 3 o más padres.

Definition 7.2.13 Elitismo Es un mecanismo adicional en los algoritmos evolutivos que permite garantizar que el cromosoma más apto $\mathbf{x}_{best} \in \mathbf{X}_t$ se transfiera a la siguiente generación $\mathbf{X}_{t+1}$, sin depender de la selección ni la reproducción.

$$\mathbf{x}_{best} \in \mathbf{X}_t \rightarrow \mathbf{X}_{t+1} \tag{7.11}$$

El elitismo garantiza que la mejor aptitud de una población (individuo $\mathbf{x}_{best} \in \mathbf{X}_t$) nunca será peor de una generación t a la siguiente $t+1$. No es una condición suficiente para encontrar el óptimo global pero sí está demostrado que es una condición necesaria para garantizar la convergencia al óptimo de un algoritmo genético [**Rudolph1994**].

Principio basado en la Naturaleza

Un algoritmo Evolutivo es una abstracción de los procesos y principios establecidos por el Darwinismo y Neo-darwinismo junto al principio *goal-driven* (conducido por los objetivos) [**Hauw1996**].
También se puede afirmar que el espacio de búsqueda $\mathbb{G}$ es una abstracción del conjunto de posibles cadenas de ADN de la naturaleza, jugando cada elemento $\mathbf{g} \in \mathbb{G}$ el papel de los genotipos naturales.
Así, se puede ver al espacio $\mathbb{G}$ como un *genoma* y a los elementos $\mathbf{g} \in \mathbb{G}$ como los genotipos.
De la misma manera que cualquier ser vivo que siendo una *instancia* de su genotipo formado durante la *embriogénesis*, una solución candidata (o fenotipo $\mathbf{x} \in \mathbb{X}$ en el espacio de problema $\mathbb{X}$ también es una instancia de su genotipo que fue obtenida mediante una función de mapeamiento $\gamma : \mathbf{g} \mapsto \mathbf{x}$ (también conocida como codificación, y su aptitud es calculada de acuerdo a las funciones objetivo a las cuales está sujeta la optimización y dirigen la evolución hacia un objetivo específico (*goal driven*)

Ciclo básico de un algoritmo Evolutivo

Un proceso evolutivo en general se puede simular computacionalmente usando los siguientes mecanismos:

1. Una forma de codificar las soluciones $\mathbf{x}$ en estructuras $\mathbf{g}$ que se reproducirán conformando una población $\mathbf{G}_0$ inicial generada aleatoriamente.
2. Una función de asignación de aptitud $h(\bullet)$ que depende de los individuos $\mathbf{x}$ y su evaluación $f(\mathbf{x})$.
3. Un mecanismo de selección basado en la aptitud.
4. Operaciones que actuen sobre los individuos codificados $\mathbf{g}_i \in \mathbf{G}$ para reproducirlos.

Estos mecanismos siguen un orden de ejecución como muestra la figura 7.3

7.2.3 Paradigmas de la Computación Evolutiva

Aunque no es muy fácil distinguir las diferencias entre los distintos tipos de algoritmos evolutivos en la actualidad, se puede diferenciar tres principales paradigmas de la Computación Evolutiva [**Back1997**]

- Programación Evolutiva

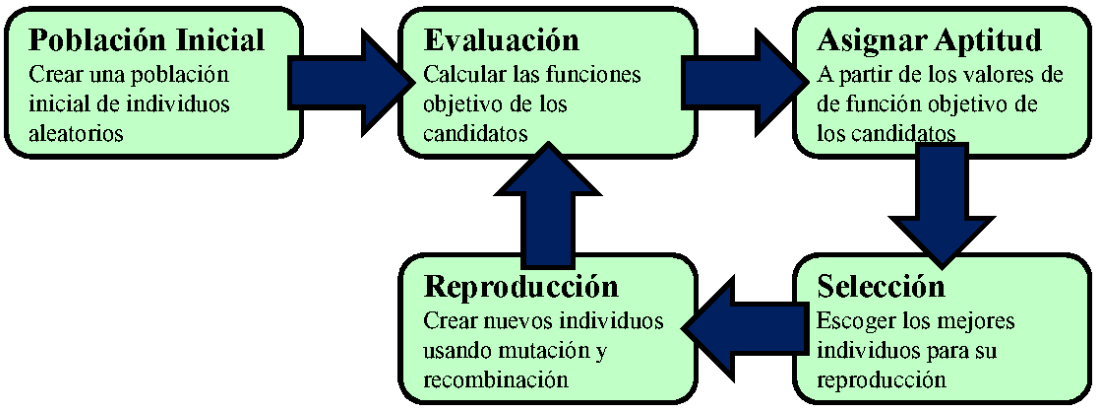


Figura 7.3: El ciclo básico de un Algoritmo Evolutivo

- Estrategias Evolutivas
- Algoritmos Genéticos

Adicionalmente se encuentran dos paradigmas más: *Learning Classifier Systems* y Programación Genética, que son bastante próximos a Algoritmos Genéticos y Programación Evolutiva respectivamente. Cada uno de estos paradigmas se originó de manera independiente y con motivaciones muy distintas. La figura 7.4 ilustra una clasificación de los principales paradigmas que conforman la familia de los Algoritmos Evolutivos.

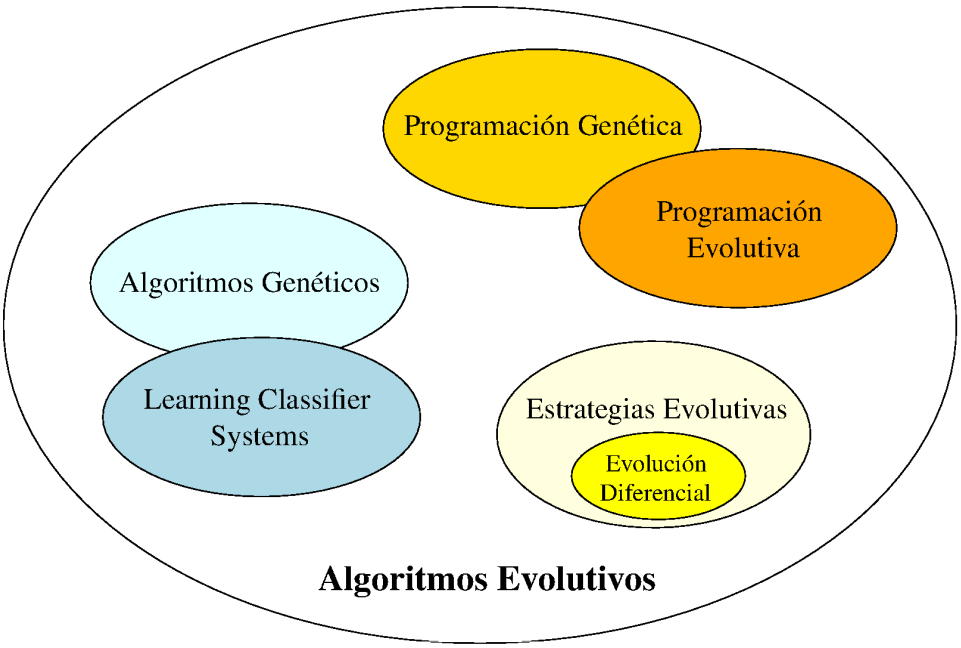


Figura 7.4: Familia de Algoritmos Evolutivos

A continuación se revisarán rápidamente los principales paradigmas y más adelante se dará todo el detalle a Algoritmos Genéticos .

Programación Evolutiva

Es una concepción inicial de la evolución simulada orientada a solucionar problemas, en especial el de predicción [**Fogel65**]. La técnica, denominada *Programación Evolutiva* [**Fogel1966**] consistía básicamente en hacer evolucionar autómatas de estados finitos, los cuales eran expuestos a una serie de símbolos de entrada (el ambiente), esperando que, en algún momento sean capaces de predecir las secuencias futuras de símbolos que recibirían. Fogel utilizó una *función de recompensa* que indicaba qué tan bueno era un cierto autómata para predecir un símbolo, y usó un operador de mutación para efectuar cambios en las transiciones y en los estados de los autómatas que tenderían a hacerlos más aptos para predecir secuencias de símbolos.
Esta técnica no consideraba el uso de un operador de recombinación sexual porque pretendía modelar el proceso evolutivo a nivel de especies y no a nivel de individuos.
La programación evolutiva se aplicó originalmente a problemas de predicción, control automático, identificación de sistemas y teoría de juegos, entre otros.
Probablemente la programación evolutiva fue la primera técnica basada en la evolución en aplicarse a problemas de predicción, además de ser la primera en usar codificaciones de longitud variable (el número de estados de los autómatas podía variar tras efectuarse una mutación), además de constituir uno de los primeros intentos por simular la co-evolución.
En la Programación Evolutiva la inteligencia es vista como un comportamiento adaptativo [**Fogel1966**] donde se le da más importancia a los nexos de comportamiento entre padres e hijos más que los operadores específicos.
El Algoritmo básico de la Programación Evolutiva es el siguiente:

Algorithm 5 Algoritmo básico de la Programación Evolutiva

$t = 0$ **generar** una población inicial $\mathbf{X}_t$ CFin = *falso* **mutar** $\mathbf{X}_t$ **evaluar** $\mathbf{X}_t$ **seleccionar** elementos de $\mathbf{X}_t$ $t = t + 1$

Así, la programación evolutiva es una abstracción de la evolución a nivel de especies, donde cada individuo es una especie y por esto no es necesario usar operadores de recombinación, pues diferentes especies no se pueden cruzar entre sí. La selección que usa es probabilística (por ejemplo torneo estocástico).

Estrategias Evolutivas

Las Estrategias Evolutivas fueron desarrolladas en [**Rechenberg73**], como una heurística de optimización basada en la idea de adaptación y evolución, con la intención inicial de resolver problemas hidrodinámicos con alto grado de complejidad. Estos problemas consistían en la experimentación en un túnel de viento para optimizar la forma de un tubo curvo, minimizar el arrastre entre una placa de unión y optimizar la estructura de una boquilla intermitente de dos fases.
La descripción analítica de estos problemas era imposible y claro, su solución usando métodos tradicionales como el de gradiente lo era también. Esto impulsó a Ingo Rechenberg a desarrollar un método de ajustes discretos aleatorios inspirados en el mecanismo de mutación que existe en la naturaleza. Los resultados de primeros estos experimentos se presentaron en el Instituto de Hidrodinámica de su Universidad [**Fogel98**].

En los dos primeros casos (el tubo y la placa), Rechenberg procedió a efectuar cambios aleatorios en ciertas posiciones de las juntas y en el tercer problema procedió a intercambiar, agregar o quitar segmentos de boquilla. Sabiendo que en la naturaleza las mutaciones pequeñas ocurren con mayor frecuencia que las grandes, Rechenberg decidió efectuar estos cambios en base a una distribución binomial con una varianza prefijada. El mecanismo básico de estos primeros experimentos era crear una mutación, ajustar las juntas o los segmentos de boquilla de acuerdo a ella, llevar a cabo el análisis correspondiente y determinar qué tan buena era la solución. Si ésta era mejor que su predecesora, entonces pasaba a ser utilizada como base para el siguiente experimento. De tal forma, no se requería información sobre la cantidad de mejoras o deterioros que se efectuaban. Esta técnica tan simple dio lugar a resultados inesperadamente buenos para los tres problemas en cuestión.

Características

Las Estrategias Evolutivas tienen las siguientes características:

1. Los candidatos a solución son vectores $\mathbf{x} \in \mathbb{X} \subseteq \mathbb{R}^n$, $\mathbf{x} = (x_1, \ldots, x_n)$ a los que no se les aplican ninguna codificación, es decir que $\mathbb{G} = \mathbb{X}$. Así el espacio de soluciones queda definido como $\mathbb{X} \subseteq \mathbb{R}^n$. Esto quiere decir que tanto el espacio de búsqueda como el espacio de problema se expresan computacionalmente con variables de números reales (punto flotante).
2. La mutación y selección son los operadores principales siendo menos usual el cruce.
3. La mutación consiste en recorrer los elementos x_i del vector $\mathbf{x}$ e ir reemplazándolos por un número obtenido a partir de una distribución normal $N(x_i, \sigma_i^2)$. Es decir que el elemento se muta usando una distribución normal multivariada $N(\mathbf{x}, \Sigma)$.
4. Existe un criterio para actualizar el valor de σ_i^2 a usar denominado autoadaptación.

Algoritmo $(1+1)$**-EE**

La versión original $(1+1)$–EE, usaba apenas un padre y un hijo. El hijo competía con el padre, y el mejor permanecía en la generación, este tipo de estrategia de selección es determinística y con característica *extintiva* ya que los peores individuos tienen probabilidad de selección cero y simplemente desaparecen.

La mutación en el modelo $(1+1)$–EE, para la generación t se obtiene aplicando la siguiente ecuación:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{N}(0, \Sigma) \tag{7.12}$$

donde

$\mathbf{N}(0, \Sigma)$ es una variable aleatoria que siguen una distribución gaussiana multivariada con media $\mu = 0$ y matriz de covarianzas Σ.

Σ es la matriz de covarianzas con elementos $\sigma_i \sigma_j = 0, \forall i \neq j$, es decir que la matriz de covarianzas es diagonal, lo que significa que se usan distribuciones normales independientes para la mutación de cada componente x_i de $\mathbf{x}$.

El algoritmo 6 ilustra la estrategia $(1+1)-$EE

El propio Rechemberg extendió la estrategia inicial al introducir el concepto

Algorithm 6 Algoritmo $(1+1)-$EE

$k, m\ t=0\ p_s=0$ **inicializa** $\mathbf{x}_t=(x_1,\ldots,x_n)$ aleatoriamente **evalua** $f(\mathbf{x}_t)\ t\leq m$ **mutar** $\mathbf{x}_{t-mut}=\mathbf{x}_t+\mathbf{N}(0,\Sigma_t)$ **evaluar** $\mathbf{x}_{t-mut}\ \mathbf{x}_t=\mathbf{mejor}(\mathbf{x}_{t-mut},\mathbf{x}_t)\ \mathbf{x}_t=\mathbf{x}_{t-mut}$

$$p_s=p_s+1\ t=t+1\ t\ \mathbf{mod}\ k=0\ \Sigma_t=\begin{cases}\Sigma_t/c & \text{si}\quad p_s/k<1/5\\ c\Sigma_t & \text{si}\quad p_s/k>1/5\quad p_s=0\ \Sigma_t=\Sigma_{t-1}\\ \Sigma_t & \text{si}\quad p_s/k=1/5\end{cases}$$

de población en la estrategia denominada $(\mu+1)$–EE, en la que existen μ antecesores y se genera un unico descendiente que puede reemplazar al peor de los antecesores de la población (selección extintiva).

Estrategias $(\mu+\lambda)$–EE / (μ,λ)–EE
Más adelante, Schwefel [**Schwefel77**] propuso tener múltiples descendientes estableciéndose las estrategias $(\mu+\lambda)$–EE y (μ,λ)–EE cuya diferencia es la metodología de selección:

- En $(\mu+\lambda)$–EE se juntan los conjuntos de μ antecesores y λ descencientes en un conjunto de tamaño $\mu+\lambda$ y los μ mejores individuos sobreviven
- En (μ,λ)–EE, sólo los μ mejores descendientes sobreviven, esto obliga que se cumpla que $\lambda\geq\mu$.

Convergencia de las Estrategias Evolutivas
Rechemberg formuló una regla para ajustar el valor de desviación estándar σ durante el proceso evolutivo, de tal forma que el procedimiento mantenga la convergencia hacia el óptimo. Esta regla es conocida como la *regla del éxito* 1/5 que se describe como:
La razón entre mutaciones exitosas (que hagan mejorar la solución) y el total de mutaciones debe ser 1/5. Si es mayor, entonces se incrementa la desviación estándar, si es menor, entonces debe decrementarse.
De una forma más elegante:

$$\sigma_t=\begin{cases}\sigma_{t-n}/c & \text{si } p_s<1/5\\ \sigma_{t-n}c & \text{si } p_s>1/5\\ \sigma_{t-n} & \text{si } p_s=1/5\end{cases} \tag{7.13}$$

donde
n es la dimensión del problema,
t es la generación en curso,
p_s es la frecuencia relativa de mutaciones exitosas contada para un determinado intervalo de generaciones y barridas de mutación ($10n$ por ejemplo) y,
c es una constante de actualización, cuyo valor más típico es $c=0{,}817$.

Mecanismo de Autoadaptación
En las estrategias evolutivas además de evolucionar las variables del problema, también evolucionan los parámetros de la técnica, en este caso las las desviaciones estándar σ_i. A los antecesores seleccionados para operar se les aplica las siguientes operaciones:

$$\sigma_i' = \sigma_i e^{(\tau' N(0,1)+\tau N_i(0,1))} \tag{7.14}$$

$$x_i' = x_i + N(O,\sigma_i') \tag{7.15}$$

donde τ y τ' son constantes que están en función de n.
Cabe destacar que las estrategias evolutivas simulan el proceso evolutivo a ni-vel de un individuo permitiendo operadores de recombinación, cuya cantidad antecesores a recombinarse está parametrizada por ρ, además de eso, la recombinación puede ser:

- **Sexual**, donde el operador actua sobre dos indivíduos seleccionados aleatoriamente.
- **Panmítica**, es decir que se selecciona un individuo $\mathbf{x}_k$ fijo y otros diversos individuos $\mathbf{x}_l, l = 1, \ldots m$. El individuo $\mathbf{x}_k$ se recombina con cada uno de los vectores $\mathbf{x}_l$ para obtener cada componente x'_l del vector descendiente $\mathbf{x}'$.

Por otro lado, el proceso de selección usado es determinístico.

7.3 Algoritmo Genético Clásico

7.3.1 Introducción

Los algoritmos genéticos – denominados originalmente *planes reproductivos genéticos* – fueron desarrollados por John H. Holland a principios de la década de 1960 [**Holland62**; **Holland62a**], quien interesado en estudiar los procesos lógicos que se daban en la adaptación, e inspirado por los estudios realizados en esa época con autómatas celulares [**Schiff2007**] y redes neuronales [**Haykin1998**], se percató que el uso de reglas simples podía conllevar a comportamientos flexibles visualizando así, la posibilidad de estudiar la evolución en sistemas complejos.
Holland se percató que, para estudiar la adaptación era necesario considerar los siguientes principios:

a) la adaptación ocurre en un ambiente,
b) la adaptación es un proceso poblacional,
c) los comportamientos individuales se pueden representar como programas,
d) se pueden generar comportamientos futuros haciendo variaciones aleatorias en los programas
e) las salida de los programas tienen relación entre ellas si sus respectivas estructuras también tienen relación entre sí.

Así, Holland logró ver el proceso de adaptación como un mecanismo en el que los programas de una población interactúan y van mejorando de acuerdo a un ambiente que determina su calidad. La combinación de variaciones aleatorias con un proceso de selección basado en la calidad de los programas para el ambiente, debía conducir a un sistema adaptativo general. Es este sistema lo que Holland denominó *Plan Reproductivo Genético* [**Holland1975**]
Aunque los Algoritmos Genéticos fueron concebidos en el contexto de adaptación, parte de *Machine Learning*, actualmente son masivamente utilizados como herramienta de optimización [**Eiben2003**].

7.3.2 Definición de Algoritmos Genéticos

Los Algoritmos Genéticos, como técnicas de la Inteligencia Artificial, son algoritmos estocásticos que implementan métodos de búsqueda a partir de un modelo de algunos fenómenos de la naturaleza que son: la herencia genética y el principio Darwiniano de la supervivencia de los más aptos.

La metáfora que está por detrás de los algoritmos genéticos es la *evolución natural*. En la evolución a nivel de especies, el problema que cada especie enfrenta consiste en buscar adaptaciones que le sean benéficas para el ambiente que es complicado y cambiante. Este *conocimiento* que cada especie adquiere, está incorporado en la estructura de los cromosomas de sus miembros.
El algoritmo genético trabaja sobre una población de soluciones y enfatiza la importancia de la cruza sexual (operador principal) sobre el de la mutación (operador secundario) y usa selección probabilística basada en la *aptitud* de las soluciones, a diferencia de otros paradigmas evolutivos como la programación evolutiva y las estrategias evolutivas.

7.3.3 Componentes de un Algoritmo Genético

[**Michalewicz1996**] afirma que para poder aplicar el algoritmo genético se requiere de los siguientes 5 componentes básicos:

1. Una representación de las soluciones potenciales del problema, de naturaleza genotípica, denominada **cromosoma**.
2. Una manera de crear una población inicial de posibles soluciones, denominada **inicialización**, que por lo general se basa en un proceso aleatorio.
3. Una función objetivo que hace el papel del ambiente, *calificando* las soluciones de acuerdo a su *aptitud*, denominada **función de evaluación**.
4. **Operadores genéticos** que alteren la composición de los genomas seleccionados produciendo descendientes para las siguientes generaciones.
5. **Parametrizaciones**, es decir valores para los diferentes parámetros que utiliza el algoritmo genético (tamaño de la población, probabilidad de aplicar *crossover*, mutación, número máximo de generaciones, etc.)

7.3.4 Algoritmo Genético Canónico

También denominado *algoritmo genético tradicional*, fue introducido por [**Goldberg89**] y se usan individuos compuestos por cadenas de números binarios $\mathbf{b}_i \in \{0,1\}$ de longitud fija l que codifican todas las variables del problema. En este modelo de algoritmo genético se nota claramente el concepto de *genotipo* que es cada una de las cadenas binarias $\mathbf{b}_i$ que codifican a una respectiva solución o *fenotipo* $\mathbf{x}_i$. Aunque una codificación de fenotipo a genotipo $\gamma^{-1} : \mathbf{x} \mapsto \mathbf{g}$ no está limitada a la representación binaria en $\{0,1\}$, esta codificación es la más parecida con los cromosomas biológicos, que es la que [**Holland1975**] ideó e implementó.
En la Figura 7.5 se muestra un modelo típico de individuo usado en el algoritmo canónico de Holland que consta de una cadena binaria $\mathbf{b}_i$.

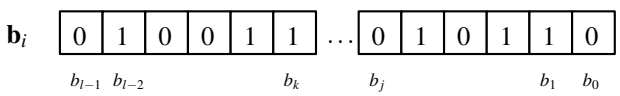


Figura 7.5: Cadena binaria $\mathbf{b}_i$ con longitud l

A la cadena binaria $\mathbf{b}$ se le denomina *cromosoma*. A cada posición de la cadena b_j se le denomina *gene* y al valor dentro de esta posición (que puede ser un valor $\{0,1\}$ se le llama *alelo*.

Procedimiento Elemental

El pseudocódigo para un algoritmo básico es el siguiente:

Algorithm 7 Procedimiento de un Algoritmo Genético

$t = 0$ **inicializa** población de genotipos $\mathbf{G}_t$ **decodificar** y **evaluar** las estructuras de $\mathbf{G}_t$ c_{fin} = **falso** $t = t+1$ **seleccionar** G_t de $\mathbf{G}_{t-1}$ **operar** G_t, formando $\mathbf{G}_t$ **evaluar** $\mathbf{G}_t$

donde

$\mathbf{G}_t$ es la población de cromosomas en la generación t.

t es el iterador de generaciones.

G_t es el conjunto de individuos selecionados para reproducir (que sufrirán cruce o mutación).

c_{fin} es una condicion de finalización del algoritmo

Para realizar la evaluación de un genotipo $\mathbf{g}_i \in \mathbf{G}_t$ es necesario decodificarlo a su fenotipo $\gamma_i : \mathbf{g}_i \mapsto \mathbf{x}_i$ y así poder aplicar la función de evaluación $f(\mathbf{x}_i)$. Al cumplirse la condición de finalización el algoritmo debe dejar de iterar, y el individuo con mejor aptitud $\mathbf{x}_{best}$ encontrado hasta ese momento será considerado la solución obtenida por el algoritmo genético. Este criterio de finalización suele ser un número máximo de generaciones It_{max} o el cumplimiento de una prueba de convergencia aplicada a la población $\mathbf{X}_t$.

La figura 7.6 ilustra este ciclo de población inicial, que incluye la selección, reproduccion (cruce y mutación) y la evaluación.

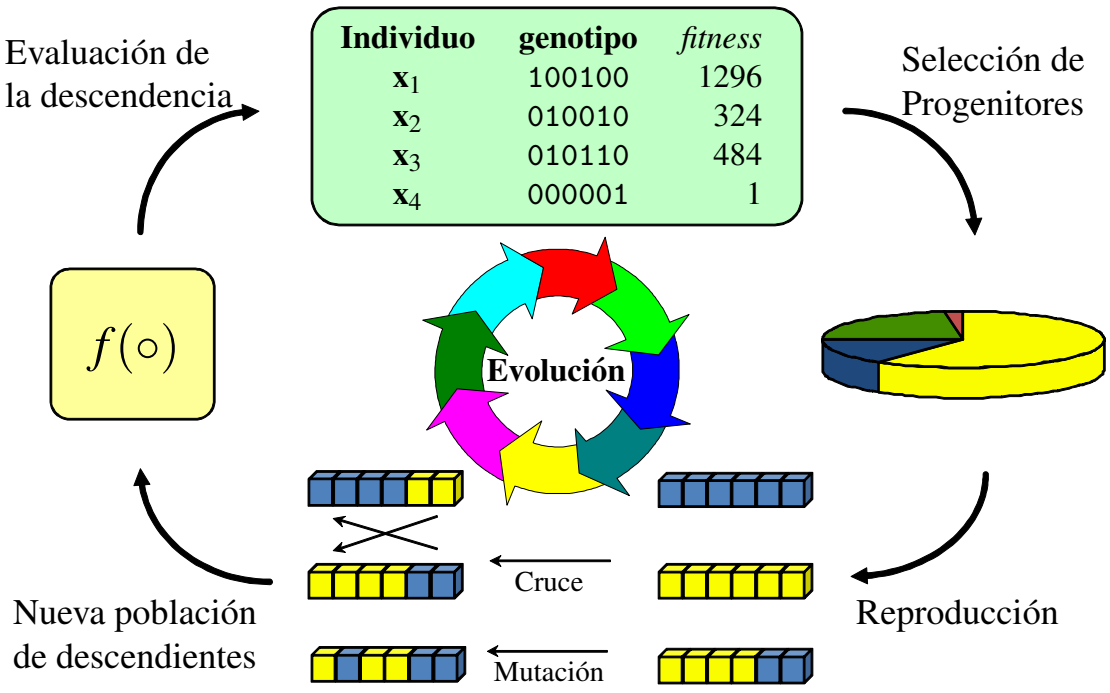


Figura 7.6: Ciclo principal de un Algoritmo Genético

Representación Binaria

Como ya fue mencionado, un algoritmo genético canónico usa cadenas de números binarios de una longitud l como cromosomas que codifican las variables de decisión, ilustradas en la figura 7.5.

Definition 7.3.1 (Cadenas Binarias) Una cadena binaria se define usando el alfabeto binario $\mathbb{B} = \{0,1\}$ y una cadena binaria de longitud l ocupa un espacio $\mathbb{B}^l = \mathbb{B} \times \ldots \times \mathbb{B} = \{0,1\}^l$. Dada una función $f(\mathbf{x})$ a optimizar, donde $\mathbf{x} = \{x_1, \ldots, x_n\}$, cada una de sus variables x_i puede ser codificada en una cadena binaria de una longitud l_i definida por el usuario. La codificación de todas las variables es obtenida concatenando las n cadenas de longitud l_i para formar una sola gran cadena de longitud $l = l_1 + \ldots, l_n$. Es sabido que una cadena binaria de longitud l puede codificar un total de 2^l puntos de búsqueda, eso significa que la longitud l_i para una variable x_i dependerá de la precisión deseada para esta variable. La ecuación 7.16 ilustra una codificación típica de n variables $\mathbf{x} = (x_1, \ldots, x_n)$ en una cadena binaria.

$$\overbrace{\underbrace{100\ldots1}_{l_1}}^{x_1} \quad \ldots \quad \overbrace{\underbrace{010\ldots0}_{l_i}}^{x_i} \quad \ldots \quad \overbrace{\underbrace{110\ldots0}_{l_n}}^{x_n} \tag{7.16}$$

donde la variable $\mathbf{x} = (x_1, \ldots, x_i, \ldots, x_n)$ es codificada mediante una concatenación de cadenas binarias de longitud $l = l_1 + \ldots + l_i + \ldots + l_n$.
Este tipo de codificación permite que un algoritmo genético pueda ser aplicado en una gran variedad de problemas, puesto que el mecanismo genético actuará evolucionando las cadenas binarias según el valor de aptitud derivado de la función de evaluación $f(\mathbf{x})$ sin preocuparse en la naturaleza de las variables $\mathbf{x}$. Cabe destacar que la cantidad verdadera de posibles valores de las variables y sus dominios son tratados por la codificación. Con esta opción se hace posible que un mismo esquema de algoritmo genético se aplique en diversos problemas sin necesidad de muchas alteraciones [**Deb2000**].
La decodificación usada para extraer los valores de las variables a partir de una cadena binaria comenzará realizando lo siguiente:

1. Dada la cadena binaria $\mathbf{b} = \{b_1, \ldots, b_l\} \in \mathbb{B}^l$ se extrae la subcadena $\mathbf{b}_i \in \mathbb{B}^{l_i}$ correspondiende a la variable x_i.
2. Se aplica la decodificación $\gamma : \mathbb{B}^l \mapsto \mathbb{X}$ donde $\mathbb{X}$ está definido de acuerdo a la naturaleza de las variables del problema.

Codificación de variables reales en binario

Dada una función $f(\mathbf{x})$ donde $\mathbf{x} \in \mathbb{X} \subseteq \mathbb{R}^n$, a ser optimizada, se requiere codificarla a cadenas binarias cumpliendo las siguientes propiedades:

- Cada vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{X}$ deve ser codificado mediante una única palabra binaria de longitud l. Esto significa que se debe definir una función de codificación γ^{-1} tal que

$$\gamma^{-1} : \mathbb{X} \mapsto \mathbb{B}^l \tag{7.17}$$

- Cada variable x_i tiene su propio domínio $\langle \text{left}_i, \text{right}_i \rangle$ definido tal que se cumpla que $x_i \in \langle \text{left}_i, \text{right}_i \rangle, \forall i = (1, \ldots, n)$. Se debe definiruna función de codificación γ_i^{-1} para la variable x_i como:

$$\gamma_i^{-1} : \langle \text{left}_i, \text{right}_i \rangle \subset \mathbb{R} \mapsto \mathbb{B}^{l_i} \tag{7.18}$$

- Si se limita la precisión de cada variable x_i, a p_i dígitos decimales el espacio $\mathbb{B}^{l_i}$ deberá ser capaz de codificar al menos $(\text{right}_i - \text{left}_i) \times 10^{p_i}$ valores para la variable x_i, como muestra la figura 7.7.

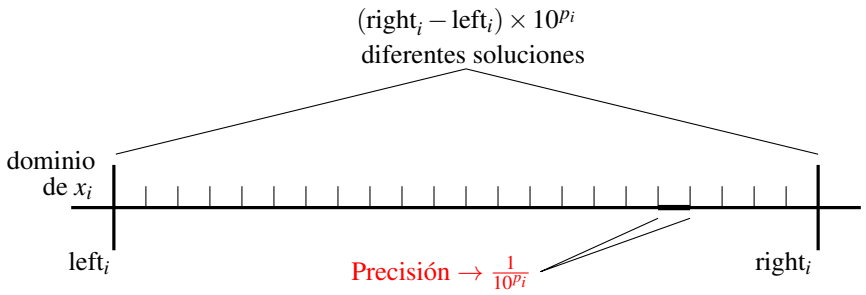


Figura 7.7: Número de posibles valores para x_i

- Dadas estas exigencias de precisión, ya es posible encontrar el valor de longitud l_i de *bits* que las satisfaga para la variable x_i. Este valor l_i queda definido por la ecuación 7.19:

$$\begin{aligned} 2^{l_i} &\geq (\text{right}_i - \text{left}_i) \times 10^{p_i} \\ l_i &\geq \log_2(\text{right}_i - \text{left}_i) + p_i \log_2(10) \end{aligned} \tag{7.19}$$

  donde $l_i \in \mathbb{N}$ ya que representa una cantidad de *bits*, que quedará definido por el valor natural inmediatamente mayor que el valor calculado como:

$$l_i = \text{ceil}[\log_2(\text{right}_i - \text{left}_i) + p_i \log_2(10)] \tag{7.20}$$

  Esta regla es aplicada porque la condición de precisión indica que, al menos, se satisfaga la precisión en dígitos decimales especificada.
- Para encontrar el tamaño l en caracteres binarios del cromosoma que codifique todas las variables de $\mathbf{x} = \{x_1, \ldots, x_n\}$, sólo queda sumar todos los l_i encontrados: $\sum_i l_i$
- Se cumple que, para un dado valor $\text{left}_i \leq x_i \leq \text{right}_i$ codificado en binario, la cadena binaria $\mathbf{b}_{inf} = (00\ldots0)$ representa el valor left_i y la cadena binaria $\mathbf{b}_{sup} = (11\ldots1)$ representa el valor right_i
- La decodificación de binario a real γ_i para una variable x_i, es definida como :

$$\gamma_i : \mathbb{B}^{l_i} \mapsto \langle \text{left}_i, \text{right}_i \rangle \subset \mathbb{R} \tag{7.21}$$

  y consiste en convertir la cadena binaria $\mathbf{b}_i$ a decimal y escalarla al intervalo de dominio $\langle \text{left}_i, \text{right}_i \rangle$ de acuerdo a la ecuación 7.22:

$$x_{i(real)} = \text{left}_i + \left(\sum_{j=0}^{l_i - 1} 2^j a_j \right) \frac{\text{right}_i - \text{left}_i}{2^{l_i} - 1} \tag{7.22}$$

Obsérvese que para esta codificación, se asume que la granularidad del espacio de búsqueda para la variable x_i es uniforme e igual a $\frac{1}{2^{l_i}}$. Si esta suposición no se cumple, es decir que la granularidad no es uniforme, como ocurre en los números de punto flotante $\pm 1.ddd\ldots \times b^E$, la función de codificación-decodificación puede ajustarse adecuadamente usando una transformación. Sin embargo, para problemas reales de búsqueda y optimización pueden no existir patrones definidos de granularidad haciendo más complicadas tanto la función de codificación como la de decodificación.

Decimal	Binario	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Cuadro 7.1: Cadenas binarias y cadenas con código de Gray

Hay casos en que algunas variables toman valores negativos y positivos. Si el dominio de estas variables es simétrico, o sea $x_i \in \langle -u_i, u_i \rangle$ podría emplearse una codificación basada en la representación de números enteros "complemento a 2" donde el *bit* más significativo está relacionado con el signo.

Ventajas y desventajas

La representación binaria es simple y fácil de manipular, nos da buenos resultados y facilita la acción de los operadores genéticos, pudiéndose decodificar a reales o entero; pero no siempre es la más adecuada ya que presenta dos problemas.

Riesgo por distancia de *Hamming*:

Se define la distancia de Hamming como la cantidad de bits diferentes entre dos palabras binarias. Cuando se emplea codificación binaria basada en LSB y MSB, en muchos casos ocurre que la distancia en *bits* entre dos valores binarios no tiene correspondencia con la distancia entre los valores numéricos codificados en binario, esto se puede observar en la tabla 7.1 en los valores 3 y 4. Una opción que puede mitigar este problema, es el empleo de codificación de Gray que se describe a continuación.

El **Código de Gray** se usa para obtener palabras binarias $\mathbb{B} = \{0, 1\}$ usando una metodología diferente para codificar y decodificar que se describe en las siguientes fases:

1. Dada una cadena convencional $\mathbf{b} = \{b_1, \ldots, b_l\} \in \mathbb{B}^l$, se convierte a código gray $\mathbf{a} = \{a_1, \ldots, a_l\} \in \mathbb{B}^l$ usando una codificación $\gamma^{-1} : \mathbb{B}^l \mapsto \mathbb{B}^l_{gray}$ expresada en la ecuación 7.23

$$a_i = \begin{cases} b_i & \text{si } i = 1 \\ b_{i-1} \oplus b_i & \text{en otro caso} \end{cases} \tag{7.23}$$

   donde $\oplus$ es el operador suma módulo 2. La principal ventaja de esta codificación es precisamente que para enteros adyacentes, la distancia de Hamming de las cadenas binarias resultantes es 1.
2. La decodificación de una cadena binaria en Código Gray $\mathbf{a} = \{a_1, \ldots, a_l\} \in \mathbb{B}^l$ para convertirla en cadena binaria convencional, consiste en aplicar la expresión

$$b_i = \bigoplus_{j=1}^{i} a_j \, \forall i = \{1, \ldots, l\} \tag{7.24}$$

entonces, ya es posible aplicar la decodificación $\gamma_i : \mathbb{B}^{l_i} \mapsto \mathbb{X}$ que sea necesaria.

La codificación completa tiene la siguiente secuencia:

$$\gamma^{-1} \quad : \quad \mathbb{X} \mapsto \mathbb{B}^l \mapsto \mathbb{B}^l_{gray} \tag{7.25}$$

$$\gamma \quad : \quad \mathbb{B}^l_{gray} \mapsto \mathbb{B}^l \mapsto \mathbb{X} \tag{7.26}$$

Incremento de la dimensionalidad del problema:

Este efecto ocurre porque la longitud l de la cadena binaria resultante es mucho mayor que la dimension n de los valores reales originales para una precisión en k *bits* especificada. El ejemplo siguiente nos ilustra este efecto para una codificación $\gamma^{-1} : \mathbb{R}^n \mapsto \mathbb{B}^l$:

■ **Example 7.1** Sea la función $f(x_1, x_2)$ con espacio de problema $\mathbb{X} \subset \mathbb{R}^2$ definido por los dominios $\langle \text{left}_1, \text{right}_1 \rangle = \langle \text{left}_2, \text{right}_2 \rangle = \langle 0, 5 \rangle$. Es claro que la dimensión original de este problema es $n = 2$

- Se desea encontrar la codificación $\gamma : \mathbb{R}^2 \mapsto \mathbb{B}^l$ para los dominios del problema y considerando una precisión de $k = 5$ dígitos decimales.
- Aplicando la metodología de la sección 7.3.4 se obtienen cadenas de longitud $l_1 = l_2 = 19$, y la cadena binaria total será de longitud $l_1 + l_2 = 38$ bits binarios.
- Por lo tanto, la codificación resultante es $\gamma^{-1} : \mathbb{R}^2 \mapsto \mathbb{B}^{38}$.

■

Inicialización de la Población

Sea $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^m$ una población compuesta por m individuos de dimensión n, esto es, con n variables $\mathbf{x} = (x_1, \ldots, x_n)$. La inicialización de la población inicial $\mathbf{X}_0$ se realiza para cada individuo $\mathbf{x}_i \in \mathbf{X}_0, i = 1, \ldots, m$ conforme la ecuación 7.27, colocando valores aleatorios en cada una de las variables $x_1, x_2, \ldots, x_n$. Queda claro que esta inicialización debe respetar los dominios de cada variable $x_j, \forall j = (1, \ldots, n)$.

$$x_j \in \left\langle \text{left}_j, \text{right}_j \right\rangle \longrightarrow x_j = \text{left}_j + (\text{right}_j - \text{left}_j)\mathbf{U}(t) \tag{7.27}$$

donde $\mathbf{U}(t)$ es un valor aleatório con distribución uniforme (0,1). Obsérvese que si la realización de $\mathbf{U}(t) = 0$, el valor inicializado coincide con left_j, y si la realización $\mathbf{U}(t) = 1$, el valor inicializado coincide con right_j.

Inicialización para codificación $\gamma : \mathbb{R}^n \mapsto \mathbb{B}^l$

Si se está empleando un mapeamiento $\gamma^{-1} : \mathbb{R}^n \mapsto \mathbb{B}^l$, cada variable x_i estará codificada por cadenas $\mathbf{b} \in \mathbb{B}^{l_i}$ con valores $b_i \in \{0, 1\}$. La inicialización puede hacerse directamente generando *bits* aleatorios 0 ó 1 y colocándolos en cada en cada uno de los alelos b_i de la cadena $\mathbf{b}$ siendo inicializada, sin preocuparse por los dominios puesto que, la precisión ya fue determinada al aplicar la ecuación 7.20 y los dominios están definidos y garantizados tanto en la codificación $\gamma^{-1} : \mathbb{R}^n \mapsto \mathbb{B}^l$ como en la decodificación $\gamma : \mathbb{B}^l \mapsto \mathbb{R}^n$ conforme ya fue visto en la ecuación 7.22.

■ **Example 7.2** Sea la **Función** F_6, aplicada a dos variables, $F_6(x_1, x_2)$ que es una función típica llena de máximos y mínimos locales, lo que la convierte en difícil para encontrar el óptimo global $\mathbf{x}^*$ que es bien conocido y localizado

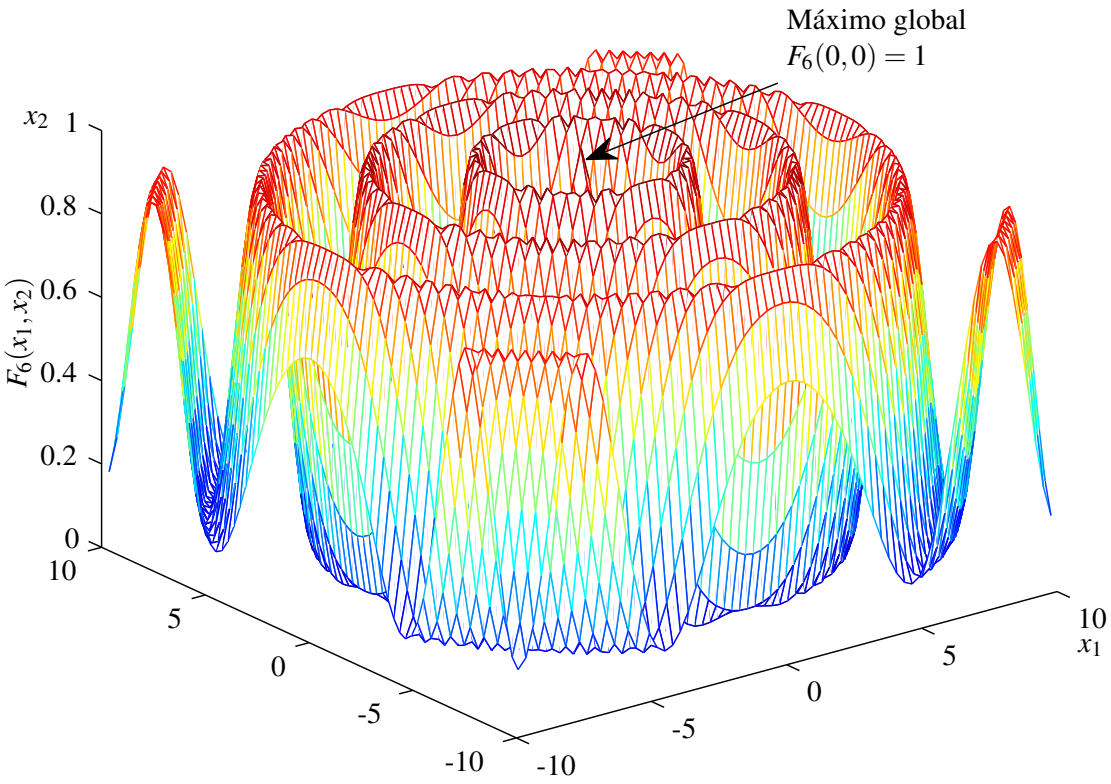


Figura 7.8: Visualización de la Función F_6

en $f(\mathbf{x}^*) = f(0,0) = 1{,}00$. La ecuación 7.28 representa la función F_6.

$$F_6(x_1,x_2) = 0{,}5 - \frac{\left(\operatorname{sen}\sqrt{x_1^2+x_2^2}\right)^2 - 0{,}5}{\left(1{,}0+0{,}001\left(x_1^2+x_2^2\right)\right)^2} \tag{7.28}$$

Esta ecuación tiene 2 variables $x_1, x_2 \in \mathbb{X} \subset \mathbb{R}^2$ y no tiene restricciones a priori para los valores de x_1, ni x_2. Para este ejemplo, se enmarca x_1, x_2 en los siguientes dominios:

$$x_1 \in \langle -100, 100\rangle$$
$$x_2 \in \langle -100, 100\rangle$$

En la figura 7.8 se visualiza parte de esta función, para los intervalos $x_1 \in \langle -10, 10\rangle, x_2 \in \langle -10, 10\rangle$, donde se observan los múltiples máximos y mínimos que son característicos de la función f_6. Nótese el punto $\mathbf{x}^* = (0,0)$ y el valor $f(0,0) = 1{,}0$ que es el máximo global, como se señala en la figura.

■

Definiendo el cromosoma

Supóngase que se desea codificar el individuo usando cadenas binarias, esto nos obliga a pensar en una codificación del tipo $\gamma^{-1} : \mathbb{X} \mapsto \mathbb{B}^l$, donde $\mathbb{B}^l$ es el espacio de cadenas binarias de longitud l que representan individuos enteros. Si se define una precisión de 4 a 5 cifras decimales para cada variable x_i, al aplicar la ecuación 7.20, se calcula el número de $l_i \in \mathbb{N}$ *bits* binarios para cada

x_i de la siguiente manera:

$$\begin{aligned} \log_2((100-(-100)\times 10^4) \leq \quad & l_i \quad \leq \log_2((100-(-100)\times 10^5) \\ \log_2(2\times 10^6) \leq \quad & l_i \quad \leq \log_2(2\times 10^7) \\ 20{,}93 \leq \quad & l_i \quad \leq 24{,}25 \end{aligned}$$

Tenemos que l_i podría asumir los valores $21, 22, 23, 24$, de los cuales se selecciona $l_i = 22$. Por lo tanto, el cromosoma binario que codifica $\mathbf{x} = (x_1, x_2)$ tendrá una longitud total $l_1 + l_2 = 44$ *bits*.
La inicialización de la población inicial consistirá simplemente en el llenado de valores $\{0, 1\}$ aleatorios para todos y cada uno de los cromosomas codificados $\mathbf{b}_i \in \mathbf{B}_0$

7.3.5 Función de Evaluación

La función de evaluación es el enlace entre el algoritmo genético y el problema en cuestión. Se puede definir como el resultado de aplicar en el problema a ser optimizado, los valores $\mathbf{x} = \{x_1, \ldots, x_n\}$ propuestos en un individuo. Dado el individuo $\mathbf{x}_i$, su función de evaluación se denota como $f(\mathbf{x}_i)$.

Aptitud

A partir de la función de evaluación, se define la aptitud a como el valor numérico que se le asigna a cada individuo (fenotipo) $\mathbf{x}_i$ indicando qué tan bueno es con respecto a los demás individuos de la población $\mathbf{X}$ para la solución del problema, como en la ecuación 7.29.

$$a(f(\mathbf{x}_i), \mathbf{X}) = \text{aptitud de } \mathbf{x}_i \tag{7.29}$$

Cabe destacar que para muchos casos, la aptitud es la propia función de evaluacion, esto quiere decir que $a(f(\mathbf{x}_i), \mathbf{X}) = f(\mathbf{x}_i)$ lo que suele crear confusiones entre ambos conceptos. Por esto debe quedar claro que la función de evaluación depende sólo del individuo $\mathbf{x}_i$ y del problema de optimización mientras que la aptitud del mismo individuo puede depender de él y del resto de individuos de la población $\mathbf{X}$ con respecto al problema en optimización.

■ **Example 7.3** Suponiendo el uso de una codificación/decodificación binario a entero $\gamma : \mathbb{B}^{12} \mapsto \mathbb{Z}^2$ con cadenas binarias de longitud $l_1 + l_2 = 6 + 6$ para un fenotipo (x_1, x_2) y una función de aptitud igual a la evaluación $f(x_1, x_2) = x_1^2 + x_2^2$, tenemos que

$$f(010011_2, 010110_2) = 19^2 + 22^2 = 845 \tag{7.30}$$

■

No existe limitación en la complejidad de la función de evaluación, puede ser tan simple como la ecuación polinómica del ejemplo 7.3, o ser tan compleja que requiera la ejecución de varios módulos de simulación para evaluar los valores de las variables especificadas en el individuo. La figura 7.9 ilustra cómo seria el proceso de evaluación para el problema f_6.
El objetivo de esta etapa es obtener, a partir de una población $\mathbf{X}_t$ en la generación t, un vector de aptitudes $\mathcal{A}_t$ que será empleado en la siguiente etapa del proceso evolutivo: la selección.

7.3.6 Estrategias de selección

Una vez creada una población inicial de m individuos $\mathbf{X}_0 = \{\mathbf{x}_i\}_{i=1}^m$ y calculada la aptitud de cada uno de sus individuos $\mathbf{x}_i$ almacenada en el vector

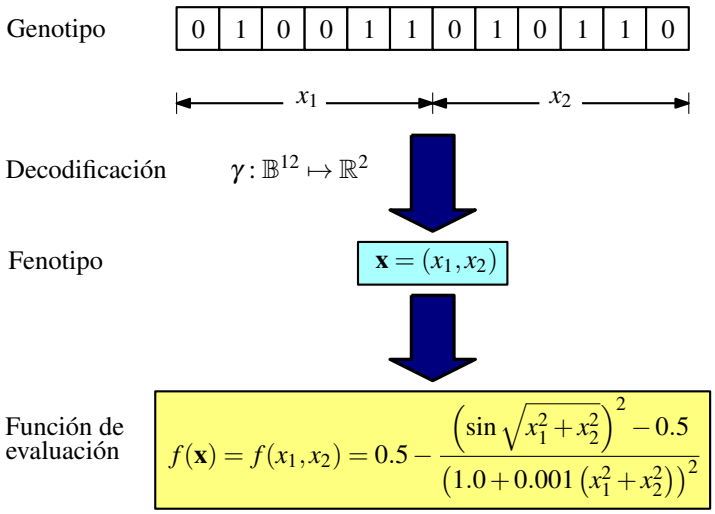


Figura 7.9: Cromosoma y Función de Evaluación

de aptitudes $\mathcal{A}_t$, se seleccionan los individuos que serán reproducidos. Esta selección suele basarse en la aptitud de cada individuo $a(f(\mathbf{x}_i), \mathbf{X}_t)$ que en este momento representa una *probabilidad* de selección de cada individuo $\mathbf{x}_i$ con respecto a los demás individuos de la población. La selección suele involucrar un sorteo usando la propabilidad, lo que la caracteriza como un proceso probabilístico.
Las técnicas de selección usadas en algoritmos genéticos se pueden clasificar en 2 grupos:

- Selección proporcional
- Selección por torneo

Selección Proporcional

Que son algunas estrategias de selección originalmente propuestas por Holland [**Holland1975**] cuya característica es que los individuos se seleccionan de acuerdo a su contribución de aptitud con respecto al total de aptitud de la población. Dentro de este grupo, se tiene las siguientes estrategias

- Selección por Ruleta (*Roulette Selection*),
- Sobrante Estocástico,
- Muestreo Determinístico.
- Universal Estocástico,

Adicionalmente existen otras técnicas que, siendo proporcionales, implementan los siguientes aditamentos

- Escalamiento-σ
- Uso de jerarquías
- Selección de Boltzmann

Selección por Ruleta

Entre todas las estrategias de selección, la más utilizada por su simplicidad y facilidad de entendimiento es la selección por ruleta o *Roulette Selection*. Su principio de trabajo se basa en formar un vector de aptitudes acumuladas donde cada individuo aporta con su peso en aptitud. Luego se realiza un sorteo generando una posición entre 0 y el valor total acumulado y se selecciona el individuo que aportó la parte donde se encuentra el punto sorteado. De

Individuo i	**fenotipo**	$a_i \in \mathcal{A}_t$	$b_i \in \mathcal{B}_t$	$a_i/b_m(\%)$
$\mathbf{x}_1$	11010110	254	254	24.49%
$\mathbf{x}_2$	10100111	47	301	4.53%
$\mathbf{x}_3$	00110110	457	758	44.07%
$\mathbf{x}_4$	01110010	194	952	18.71%
$\mathbf{x}_5$	11110010	85	1037	8.20%
Total		**1037**		**100.00%**

Cuadro 7.2: Ejemplo de aptitudes para aplicación de la ruleta

esta forma, un individuo que aporte más aptitud tendrá más posibilidad de ser seleccionado. El procedimieto de la ruleta se describe en el algoritmo 8:

Algorithm 8 Selección por Ruleta

$\mathbf{X}_t$ $\mathcal{A}_t$ **calcular** $\mathcal{B}_t = \{b_1, \ldots, b_m\}$ $j = 0$ **generar** $u_j = b_m \mathbf{U}(0,1)$ $u_j \notin [\mathbf{x}_j, \mathbf{x}_{j+1}]$ $j = j+1$ **seleccionar** $\mathbf{x}_j \in \mathbf{X}_t$

donde:
$\mathcal{A}_t$ es el vector de aptitudes de $\mathbf{X}_t$
$\mathcal{B}_t$ con componentes $\{b_1, \ldots, b_m\}$, es el vector de aptitudes acumuladas de $\mathbf{X}_t$ que se calcula de la siguiente manera:

$$\begin{aligned} b_i &= a_i & i &= 1 \\ b_i &= b_{i-1} + a_i & i &> 1 \end{aligned} \tag{7.31}$$

b_m es el valor total de aptitud acumulada
$\mathbf{U}(0,1)$ es un generador de números aleatorios entre 0, 1
Dado un individuo $\mathbf{x}_i$ y su aptitud $a(\mathbf{x}_i, \mathbf{X}_t)$, su probabilidad de selección $p_{\mathbf{x}_i}$ para el modelo de selección por ruleta será:

$$p_{\mathbf{x}_i} = \frac{a(\mathbf{x}_i, \mathbf{X}_t)}{\sum_{k=1}^{m} a_k} \tag{7.32}$$

En la tabla 7.2 se muestra un conjunto de valores ejemplo para un algoritmo genético con cromosomas binarios.
cuya ruleta resultante para aplicar el algoritmo de selección es ilustrada por la figura 7.10:

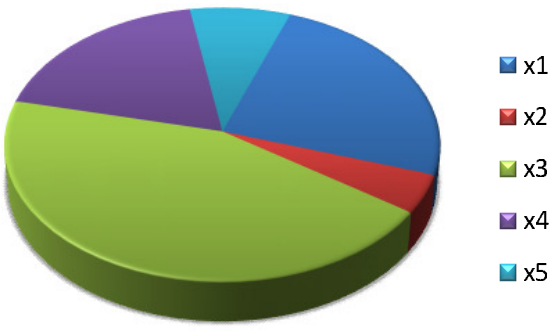


Figura 7.10: Ejemplo de ruleta para la población y aptitudes de la tabla 7.2

Individuo i	**Fenotipo**	$a_i \in \mathcal{A}_t$	e_t	**int**(e_t)	e_t−**int**(e_t)
$\mathbf{x}_1$	110100	220	1.23	1	0.23
$\mathbf{x}_2$	011010	140	0.78	0	0.78
$\mathbf{x}_3$	111001	315	1.76	1	0.76
$\mathbf{x}_4$	001101	42	0.23	0	0.23
Total		**717**	**4.00**	**2**	

Cuadro 7.3: Ejemplo de aplicación del Sobrante Estocástico

Análisis de la Selección por Ruleta

La ruleta tiene algunas deficiencias que se describen a seguir:

- Su complejidad computacional es $O(n^2)$. El algoritmo se hace ineficiente cuando crece el tamaño m de la población.
- Está sujeta al error de muestreo, es decir que el valor esperado $E_s[\mathbf{x}_i]$ de selección, que depende de la aptitud relativa (o probabilidad) de selección de un individuo, no siempre se cumplirá pudiendo haber diferencias grandes. Esto se visualiza en la posibilidad de que el peor individuo puede ser escogido varias veces.
- Es posible utilizar búsqueda binaria en vez de búsqueda secuencial, lo que requiere memoria adicional y una primera barrida con complejidad $O(n)$. Con esto, la complejidad total de la ruleta pasa a ser $O(n \log n)$.

Sobrante Estocástico

Es una alternativa que busca obtener una mejor aproximación de los valores esperados de selección e los individuos $E_s[\mathbf{x}_i]$ definidos por la ecuación 7.33:

$$E_s[\mathbf{x}_i] = m\frac{a(f(\mathbf{x}_i), \mathbf{X}_t)}{b_m} \tag{7.33}$$

donde:
$a(\cdot)$ es la aptitud del individuo $\mathbf{x}_i$
b_m es la aptitud acumulada de la población.
m la cantidad de elementos en la población.
Lo que se hace es:

1. Asignar en forma determinística el conteo de los valores esperados, es decir, la parte entera de $e_t = m(a_i/b_m)$ para después,
2. Usando un esquema probabilístico y proporcional, completar los individuos sobrantes por el redondeo usando la parte decimal restante..

El esquema de completación de individuos faltantes tiene dos variantes:

- **Sin reemplazo** donde cada sobrante se usa para sesgar un sorteo que determina si un individuo se selecciona o no.
- **Con reemplazo** donde se usan los sobrantes para dimensionar una ruleta y se usa esta técnica para la selección.

La tabla 7.3 ilustra un ejemplo del uso del sobrante estocástico
Esta metodología permite disminur los problemas de muestreo de la ruleta, aunque se puede causar convergencia prematura debido a la mayor presión selectiva.

Selección por Torneo

La selección por torneo o *Tournament Selection* que fue propuesta en [**Wetzel1983**], tiene como idea básica seleccionar individuos comparando directamente sus

aptitudes como describe el algoritmo 9:

Algorithm 9 Selección por Torneo

$\mathbf{X}_t$ $\mathcal{A}_t$ desordenar **$\mathbf{X}_t$ escoger** k individuos **comparar** los k individuos por su aptitud $a(\cdot)$ **seleccionar** el individuo con mejor aptitud

donde k suele tener el valor $k = 2$.
La selección por torneo es bastante simple y es *determinista* ya que los individuos menos aptos nunca serán seleccionados (característica extintiva). Para compensar este efecto, existe una versión de torneo *probabilística*, cuya principal diferencia surge al momento de escoger el ganador. En vez de seleccionar al individuo más apto, se hace un sorteo $\mathbf{U}(t) < p$, donde $\mathbf{U}(t)$ es un generador de números aleatorios entre $[0, 1\rangle$ y $0{,}5 < p \leq 1$. Si el resultado se cumple, se selecciona al elemento más apto, caso contrario se selecciona el menos apto. El valor p se mantiene fijo durante todo el proceso evolutivo.
Esta variante *probabilística* reduce la presión selectiva eliminando la característica extintiva, ya que en algunas ocasiones los elementos menos aptos podrían ganar el torneo y ser seleccionados.

Análisis del Torneo

- La versión determinística garantiza que el mejor individuo sea seleccionado
- Cada competencia requiere la selección aleatoria de k individuos, que es un valor constante, por lo que se puede decir que su complejidad es $O(1)$.
- Se requieren m competencias para lograr seleccionar una nueva población completa para una generación. Por lo tanto, la complejidad de este algoritmo es $O(m)$.
- La técnica es eficiente y fácil de implementar.
- No se requiere escalar la función de aptitud, las comparaciones se hacen directamente.
- En la versión determinística, se puede introducir una presión selectiva fuerte, ya que los individuos menos aptos no tienen opción de sobrevivir.
- La presión de selección se puede regular variando el tamaño k de los individuos que compiten.
  - Para $k = 1$, la selección es equivalente a un paseo aleatorio, sin presión selectiva
  - Para $k = m$, la selección es totalmente determinística siempre escogiendo al mejor elemento de la población
  - Para $2 \leq k \leq 5$, se considera una selección *blanda*
  - Para $k \geq 10$ se considera una selección *dura*

7.3.7 Operadores Genéticos

Los operadores son aquellos mecanismos que manipulan el genotipo $\mathbf{g}$ y tienen influencia en la forma como la información genética es transmitida de *padres* a *hijos*. Los principales operadores recaen en las siguientes categorías:

- **Cruces** (cruzamientos o *crossover*): que forman dos nuevos individuos a partir del intercambio de partes o recombinación de información de los dos individuos seleccionados para operar (en un sentido amplio, el número de antecesores podría ser mayor que dos) En este tipo de operador ocurre la transmisión de características hereditarias. Los tipos de cruce

más conocidos son el cruce de un punto, cruce de dos puntos y cruce uniforme.

- **Mutaciones:** que forman un nuevo individuo a partir de alteraciones en los genes de un único individuo seleccionado

Cruce de un punto

En este operador, a partir de una pareja de genotipos seleccionados $\mathbf{g}_1$ y $\mathbf{g}_2$ y seleccionando un punto de corte aleatorio, se generan dos descendientes $\mathbf{g}'_1$ y $\mathbf{g}'_2$ intercambiando parte de su información como ilustra la figura 7.11

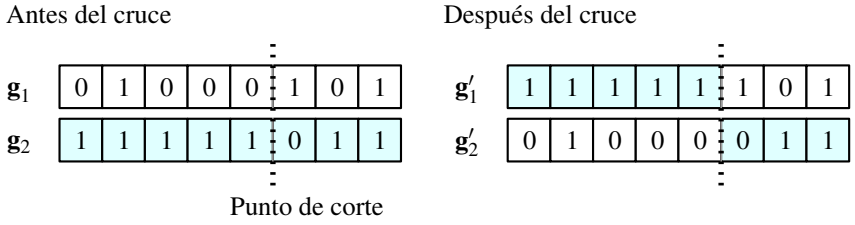


Figura 7.11: Cruce de un punto

Cruce de dos puntos

En este caso, se usan dos puntos de corte, generados aleatoriamente. Se obtiene un segmento de los cromosomas *padre* $\mathbf{g}_1$ y $\mathbf{g}_2$ que se intercambiará conforme ilustra la figura 7.12

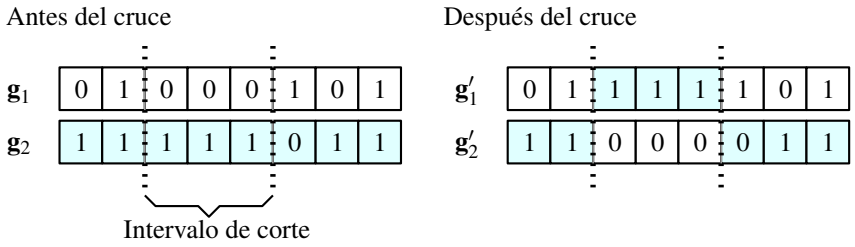


Figura 7.12: Cruce de dos puntos

Cruce uniforme

Dados los cromosomas seleccionados $\mathbf{g}_1$ y $\mathbf{g}_2$, este operador barre sus respectivas estructuras utilizando un sorteo $\mathbf{U}(t)_i$ para determinar cual gene g_{1i} ó g_{2i} de los *padres* aportará para formar los genes de los descendientes $\mathbf{g}'_1$ y $\mathbf{g}'_2$ como muestra la ecuación 7.34 e ilustrado en la figura 7.13. .

$$g'_{1i} = \begin{cases} g_{1i} & \text{si } \mathbf{U}(t)_i \leq 0{,}5 \\ g_{2i} & \text{caso contrario} \end{cases} \quad g'_{2i} = \begin{cases} g_{1i} & \text{si } \mathbf{U}(t)_i > 0{,}5 \\ g_{2i} & \text{caso contrario} \end{cases} \tag{7.34}$$

Mutación

Los operadores de mutación seleccionan algún gen $g_i \in \mathbf{g}$ y lo alteran por otro valor. Tratándose de un individuo con genotipo binario, un gen sería un *bit* y su alteración es de valor 1 a valor 0 o viceversa.

El operador de mutación puede sufrir algunas variaciones para selecionar y alterar el *bit* como se describe:

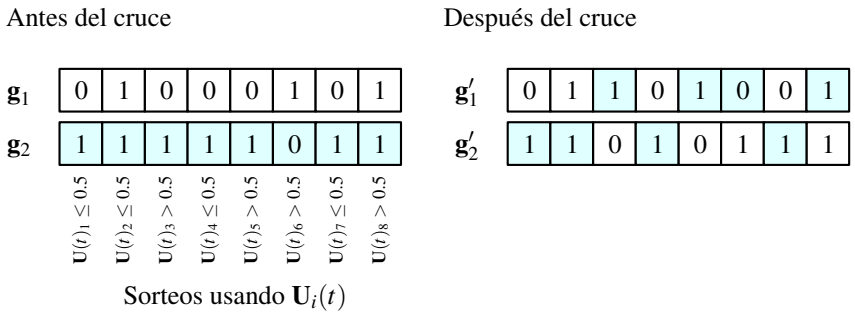


Figura 7.13: *Crossover* uniforme

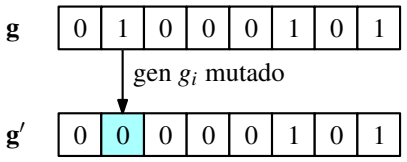


Figura 7.14: Mutación de un *bit*

- **Opción 1:** Aplicar sorteo a todos los genes del cromosoma con probabilidad de mutación p_m y para cada gen seleccionado, colocar un valor aleatorio.
- **Opción 2:** Aplicar sorteo y colocar el *bit* complementario al gene sorteado
- **Opción 3:** Escoger aleatoriamente un solo gen del cromosoma y cambiar su contenido por un valor aleatorio

El Dilema *Exploiting – Exploring*

En la mayoria de sistemas de aprendizaje existe la necesidad de aprovechar los conocimientos existentes (*Exploiting*) y también buscar nuevo conocimiento (*Exploring*), que son características o comportamientos que *a priori* parecen ser contradictorios, caracterizando un dilema o una paradoja. En general, cualquier algoritmo evolutivo, no es ajeno a esta situación.

En un modelo evolutivo, el *Exploiting* se ve como el aprovechamiento de la información que la población actual $\mathbf{X}_t$ posee sobre el espacio del problema $\mathbb{X}$ y determinar los lugares más promisorios o interesantes para visitar, o sea *sacar el jugo* a la información que se tiene en la población actual.

Por otro lado, el *Exploring* se visualiza en la necesidad de dirigirse a regiones en el espacio $\mathbb{X}$ nunca antes visitadas para ver si aparece alguna nueva información prometedora, o sea *dar un salto a lo desconocido*. Esta característica también dará la opción de no quedarse atrapados en óptimos locales.

En este sentido los operadores de *crossover* proveen la característica *Exploiting* y los operadores de mutación proveen la característica *Exploring*.

$$\begin{aligned} \textit{Crossover} &\longrightarrow \textit{Exploiting} \\ \textit{Mutation} &\longrightarrow \textit{Exploring} \end{aligned}$$

Una consecuencia de este efecto es la necesidad de ajuste de las probabilidades de cruce y mutación durante el ajuste de un algoritmo evolutivo buscando un mejor rendimiento en la solución de un determinado problema de búsqueda u optimización.

7.3.8 Ajustes de la Aptitud

Cuando se realiza el cálculo de la aptitud podemos encontrarnos con dos problemas extremos:

- **Poca presión selectiva:** que ocurre cuando las aptitudes, producto de la evaluación de los cromosomas tienen valores numéricos muy similares, haciendo que cualquier algoritmo de selección sea ineficiente. En esta situación, el proceso evolutivo toma un comportamiento muy próximo al de una selección aleatoria.
- **Súperindividuo:** cuando existe uno o pocos individuos de la población con una aptitud muy alta con respecto a los demás individuos de la población. En este caso, el proceso de selección tenderá a escoger sólo estos individuos para reproducirse, en consecuencia, estos invadirán las poblaciones sucesivas ocasionando disminución de la diversidad genética y convergencia prematura.

Para evitar estos dos efectos indeseados, se pueden realizar ajustes de las aptitudes de la población, como:

- Aptitud = Evaluación
- *Windowing*
- Normalización lineal

que se describen a continuación

Aptitud = Evaluación

Es el caso más simple, en el que no se ve necesidad de realizar ajustes a la función de evaluación $f(\mathbf{x}_i)$ para obtener la aptitud, es decir que no se aprecia un problema de súperindividuo ni competición próxima. Consiste en utilizar la definición de aptitud de la ecuación 7.29 igualándola a la propia evaluación del individuo:

$$a(\mathbf{x}_i, \mathbf{X}_t) = f(\mathbf{x}_i) \tag{7.35}$$

Windowing

Cuando ocurre el problema de competición próxima, debido a que los individuos tienen evaluaciones muy similares, es necesario ajustar la aptitud usando la siguiente expresión:

$$a(f(\mathbf{x}_i), \mathbf{X}_t) = f(\mathbf{x}_i) - f_{min}(\mathbf{X}_t) + a_{min} \tag{7.36}$$

donde

$f_{min}(\mathbf{X}_t)$ es la evaluación mínima hallada en la población $\mathbf{X}_t$.

a_{min} (opcional) es una aptitud mínima de sobrevivencia para garantizar la reproducción de los cromosomas menos aptos.

Normalización Lineal

Es el tipo de ajuste más utilizado en Algoritmos Genéticos, que busca mitigar el efecto del súperindividuo y de los individuos con competición próxima, manteniendo un equilibrio de presión selectiva, como se describe a continuación.

Sea una población $\mathbf{X}_t$ con m individuos ordenada por las evaluaciones $\mathcal{F}_t$. Mediante la normalización lineal, se ajustan las aptitudes desde un valor mínimo v_{min} hasta un valor máximo v_{max} con pasos fijos como se indica en el algoritmo 10.

donde

Algorithm 10 Normalización Lineal

$\mathbf{X}_t$ $\mathcal{F}_t$ v_{min}, v_{max} **ordenar** $\mathbf{X}_t$ decrecientemente según $\mathcal{F}_t$ $i = 0$ $i < m$ **crear** $\mathcal{A}_t = \{a_i\}_{i=1}^{m}$ usando: $a_i = v_{min} + \dfrac{(v_{max} - v_{min})}{m-1}(i-1)$ $i = i+1$

$\mathbf{x}_i$	$f(\mathbf{x}_i)$	a_i $\langle v_{min}, v_{max}\rangle = \langle 10, 60\rangle$	a_i $\langle v_{min}, v_{max}\rangle = \langle 1, 101\rangle$
$\mathbf{x}_6$	200	60	101
$\mathbf{x}_5$	9	50	81
$\mathbf{x}_4$	8	40	61
$\mathbf{x}_3$	7	30	41
$\mathbf{x}_2$	4	20	21
$\mathbf{x}_1$	1	10	1

Cuadro 7.4: Ejemplo de Normalización lineal

$\mathcal{F}_t$ es el vector de evaluaciones de la población $\mathbf{X}_t$
v_{min} es el valor mínimo de aptitud normalizada.
v_{max} es el valor máximo de aptitud normalizada.
a_i es la aptitud normalizada linealmente para el individuo $\mathbf{x}_i$ de la población $\mathbf{X}_t$ ordenada.

Cabe resaltar que la presión selectiva está relacionada directamente con el módulo de la diferencia $|v_{max} - v_{min}|$.

■ **Example 7.4** Sea una población $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^{6}$, cuyas evaluaciones $f(\mathbf{x}_i)$ son las especificadas en la segunda columna de la tabla 7.4.
Observando la tabla 7.4, y la figura 7.15, se nota que $\mathbf{x}_6$ es un súper-individuo con una evaluación $f(\mathbf{x}_6) = 200$ es muy alta con respecto a los demás individuos. También se debe notar que existe competición próxima entre los individuos $\mathbf{x}_3, \mathbf{x}_4$ y $\mathbf{x}_5$, cuyos valores de evaluación son muy próximos entre sí. Al aplicar normalización lineal, los efectos indeseados de estos dos casos son atenuados, y la presión selectiva se puede establecer por la diferencia entre v_{min} y v_{max}.

■

Aparte de la normalización lineal, pueden ser aplicados otros tipos de normalizacion a la aptitud de los individuos, tales como ajuste geométrico, exponencial, logarítmico, etc.

7.3.9 Ajustes de la Selección

Una cuestión que se debe observar es la llamada *brecha generacional* que se refiere a la transición de elementos de la población $\mathbf{X}_t$ hacia la población $\mathbf{X}_{t+1}$.
El algoritmo genético convencional considera que cuando se realiza la selección, se obtiene una nueva población completa que posteriormente se operará (cruces y mutaciones). Esto caracteriza un comportamiento extintivo, que en algunos casos puede no ser muy beneficioso ya que eventualmente podrían perderse individuos valiosos. Para tratar esta situación se emplean dos estrategias

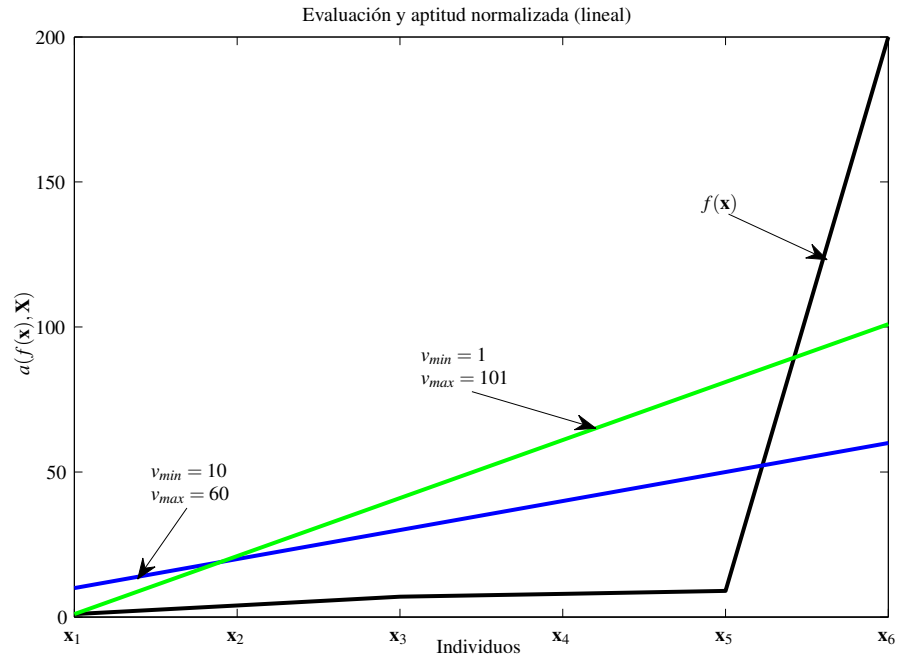


Figura 7.15: Ejemplo de Normalización lineal

Elitismo

Que consiste en almacenar una copia del mejor individuo de la población $\mathbf{x}_{best} \in \mathbf{X}_t$ para colocarlo en la población $\mathbf{X}_{t+1}$ después de haber realizado todo el proceso de evolución (selección, reproducción, evaluación).
Esta estrategia reduce el efecto aleatorio del proceso evolutivo y también garantiza que a través de las generaciones, el mejor individuo en $\mathbf{x}_{best} \in \mathbf{X}_{t+1}$ sempre será igual o mejor que el mejor individuo $\mathbf{x}_{best} \in \mathbf{X}_t$ preservando los mejores individuos para continuar reproduciéndose.

Steady-state

En este caso no hay un reemplazo total de los individuos de la población $\mathbf{X}_t$ al generar la población $\mathbf{X}_{t+1}$. La metodología de realización de *Steady State* es la que nos muestra el algoritmo 11.

Algorithm 11 *Steady-state*

seleccionar $k < m$ individuos de $\mathbf{X}_t$ **operar** estos k individuos (cruces, mutacion) **eliminar** los k peores elementos de $\mathbf{X}_t$ **obtener** $\mathbf{X}_{t+1}$ insertando los n individuos generados

donde k es la fracción de individuos a reemplazarse, $k < m$, $k = GAP \times m$.
La figura 7.16 ilustra el funcionamiento del *Steady State*
Adicionalmente se puede hacer el reemplazo parcial de la población y excluir los individuos repetidos: *Steady-state* sin repetidos. Con esta estrategia se obtienen las siguientes ventajas:

- Evitar las repeticiones que son más frecuentes en poblaciones con *Steady-state* que son más estáticas.
- Más eficiencia en el paralelismo de búsqueda, dado que se garantizan elementos diferentes (diversidad).

En este caso, cada individuo repetido es tratado de las siguientes formas

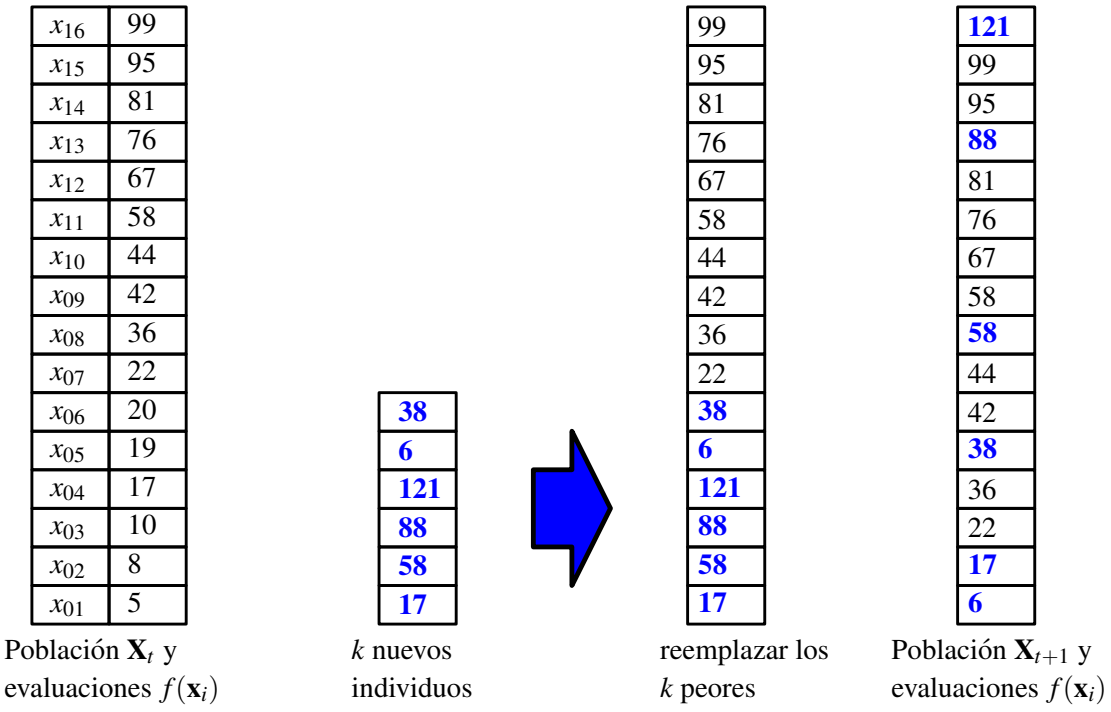


Figura 7.16: Procedimiento *Steady-state*

1. Reemplazado por un nuevo individuo generado aleatoriamente (verificando que no sea repetido) ó
2. Operar el individuo (cruce o mutación) hasta que el resultado sea un individuo diferente

Existe un mayor costo computacional debido a la verificación de los individuos repetidos.

Medidas de Monitoreo

Una forma de visualizar el comportamiento de un algoritmo evolutivo en general, consiste en ver cómo se comporta la población $\mathbf{X}$ durante la ejecución del proceso evolutivo, es decir durante las generaciones hasta que se dé la finalización de la ejecución. Lo más común es obtener curvas de desempeño de las ejecuciones de un algoritmo. Se conocen los siguientes tipos de curva:

- Curva *online*
- Curva *offline*
- Curva *best-so-far*

Curva *online*

Una curva *online* permite visualizar la rápida obtención de buenas soluciones y también la convergencia de los individuos de la población. Dado un experimento que partió con una población inicial $\mathbf{X}_0$, un punto de la correspondiente curva *online* para la generación t está dado por:

$$v_{on}(t) = \frac{1}{t+1} \sum_{i=0}^{t} \bar{f}(\mathbf{X}_t) \tag{7.37}$$

donde $\bar{f}(\mathbf{X}_t)$ es la media de las evaluaciones de todos los individuos $\mathbf{x}_i \in \mathbf{X}_t$, en la generación t. De la ecuación 7.37 se puede inferir que un punto de la curva *online* es la media de todos los individuos que fueron evaluados hasta finalizar la generación t.

Curva *off-line*

Una curva *off-line* permite visualizar la obtención de buenas soluciones sin importarse con el tiempo tomado para encontrarlas. Dado un experimento, para la generación t el valor del punto en la curva *off-line* está dado por:

$$v_{off}(t) = \frac{1}{t+1} \sum_{i=0}^{t} f(\mathbf{x}_{best})(t) \tag{7.38}$$

donde $f(\mathbf{x}_{best})(t)$ es la evaluación del mejor elemento de la generación t.
De la ecuación 7.38 se puede inferir que la curva *off-line* grafica la media de los mejores elementos desde la generación 0 hasta la generación t.

Curva *best-so-far*

Es la curva más elemental de todas en la cual, el valor del punto en la generación t es simplemente el valor $\mathbf{x}_{best}(t)$ sin calcular promedios, como expresado en la ecuación 7.39.

$$v_{best}(t) = f(\mathbf{x}_{best})(t) \tag{7.39}$$

donde $f(\mathbf{x}_{best})(t)$ es la evaluación del mejor elemento de la generación t.
Como un algoritmo genético es eminentemente un proceso estocástico de búsqueda, se hace necesario realizar muchos experimentos (ejecuciones) del mismo problema con la misma parametrización para tener una visualización más real del comportamiento de la población durante las generaciones. Entonces, es común realizar muchas ejecuciones y por cada una de ellas calcular las curvas *off-line*, *online*, y la curva de mejores elementos *best-so-far*, para luego mostrar sendas curvas con las medias de los puntos en las generaciones $0, \ldots, T$.
La figura 7.17 nos ilustra un ejemplo de curvas *best-so-far*, *off-line* y *online* para una ejecución de un algoritmo evolutivo con 200 generaciones.

7.4 Computación Evolutiva en optimización numérica

7.4.1 Uso de codificación binaria o real

La representación binaria, usada en algoritmos genéticos canónicos, tiene las deficiencias de alta dimensionalidad y distancia de Hamming, que se evidencian principalmente en problemas de optimizacion numérica con funciones multidimensionales y alta precisión numérica exigida. Por ejemplo, sea un problema de optimización mín $f(\mathbf{x})$ con las siguientes características:

- Función $f(\mathbf{x})$ a optimizar donde $\mathbf{x} = (x_1, \ldots, x_{100})$,
- Dominios $x_i \in [-500, 500]$
- Precisión mínima de seis dígitos decimales.

al realizar la codificación $\gamma^{-1} : \mathbb{R}^n \mapsto \mathbb{B}^l$, para cada variable x_i se tiene $\gamma_i^{-1} : \mathbb{R} \mapsto \mathbb{B}^{l_i}$, con $l_i = 30$, y $l = l_1 + \ldots + l_{100} = 3000$. Es decir que que la codificación es $\gamma^{-1} : \mathbb{R}^{100} \mapsto \mathbb{B}^{3000}$. Esto generaría un espacio de búsqueda de

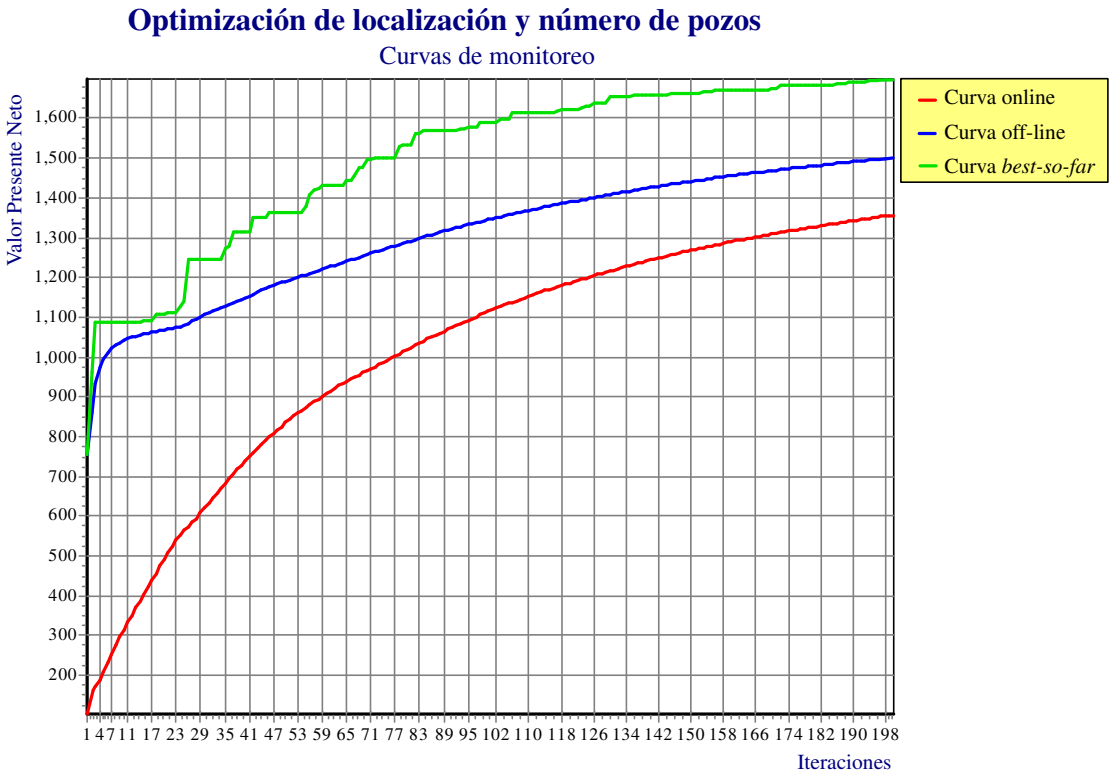


Figura 7.17: Ejemplo de curvas *best-so-far*, *off-line* y *online*.

aproximadamente 10^{1000}, haciendo que para este problema, la codificación binaria ofrecerá un mal desempeño.
Goldberg escribió: "*El uso de genes con código real o punto flotante es una larga y controversial historia en esquemas de búsqueda evolutiva y genética artificial, y su uso parece estar en aumento. Este creciente uso sorprende a los investigadores familiarizados con la teoría clásica de algoritmos genéticos, con esquemas de baja cardinalidad, lo que parece ser contraproducente con las búsquedas empíricas que los algoritmos con codificación real han mostrado en una serie de problemas prácticos.*" [**Goldberg1990**]
En [**Michalewicz1996**] se puede encontrar algunos ejemplos que comparan algoritmos genéticos con codificación binaria y algoritmos genéticos con codificación real, para problemas de optimización numérica, donde el desempeño de los algoritmos genéticos con codificación real siempre es más consistente, obteniéndose resultados más precisos.

7.4.2 Algoritmos evolutivos con codificación real

Son algoritmos evolutivos cuya población $\mathbf{X}_t$ está compuesta por individuos $\mathbf{x} = (x_1, \ldots, x_n)$ donde cada $x_i \in \mathbb{R}, \forall i = 1, \ldots n$, es decir con genes que son números reales. Pensando en una implementación computacional los cromosomas serían listas, *arrays* o vectores con variables de punto flotante (`float`, `double`). De hecho, los operadores genéticos deben tienen que adecuarse a este tipo de dato. La arquitectura genérica del algoritmo genético se mantiene (población, evaluación, selección, reproducción, ajustes en la selección y evaluación).
Esta variante de Algoritmos Genéticos tiene las siguientes ventajas:

- No se necesita codificación, manteniendo la relación entre genes y variables. Por ejemplo, dada la función objetivo $f(x_1, x_2)$, su representación en cromosoma será directamente $\mathbf{x} = (x_1, x_2)$, una consecuencia de esto es que mantiene la dimensionalidad original del problema.
- La precisión de cada gene x_i dependerá de la precisión de las variables de punto flotante del lenguaje de programación utilizado (float, double, extended, etc.)
- Facilidad para representar grandes dominios
- No existe el problemas de Distancia de Hamming
- Esta representación da la opción de tratar restricciones no triviales

Aunque también existen algunas desventajas:

- En *Strictu Sensu*, un Algoritmo Genético con representación real no sigue la definición original de "Algoritmo Genético" de Holland dado que, al no haber codificación, no existe genotipo. Se le considera un Algoritmo Genético Híbrido.
- Requiere operadores genéticos especializados para tratar números reales.
- Su aplicación está limitada a problemas de optimización numérica de espacio continuo y con funciones reales

7.4.3 Problemas con restricciones

En la literatura se encuentran muchos trabajos sobre Algoritmos Genéticos basados en números reales siempre aplicados en problemas definidos en un espacio de búsqueda $\mathbb{X} \subseteq \mathbb{R}^q$, donde $\mathbb{X} = \prod_{k=1}^{q} \langle \text{left}_k, \text{right}_k \rangle$, es decir que cada variable estaba restricta a un intervalo $\langle \text{left}_k, \text{right}_k \rangle$, donde $1 \leq k \leq q$, sin restricciones adicionales a las de dominio. Es importante colocar otras restricciones aparte de éstas ya que es sabido que la mayoría de problemas prácticos son de hecho, problemas con restricciones.
En un problema de optimización numérica con restricciones, la forma geométrica del espacio de búsqueda válido $\mathbb{X} \subseteq \mathbb{R}^q$ es la característica más crítica, de la cual depende el grado de dificultad para encontrar una solución óptima al problema. Como fue mencionado en la sección 7.1.4, si la forma geométrica del espacio de búsqueda $\mathbb{X}$ es *convexa*, existe mucha teoría que garantiza la convergencia a la solución óptima.

Modelo GENOCOP

Esta denominacion viene de las letras iniciales de **GE***netic Algorithm for* **N***umerical* **O***ptimization for* **CO***nstrained* **P***roblems.*
GENOCOP es un modelo de Algoritmo Genético presentado en [**Michalewicz1996**] e implementado en lenguaje C, adecuado para problemas de optimización numérica con las siguientes características:

- Las funciones a optimizar $f(\mathbf{x})$ están definidas en un espacio de problema continuo $\mathbb{X} \subseteq \mathbb{R}^q$,
- Existe un conjunto de restricciones lineales $\mathcal{C}_L$
- Existe un punto inicial válido $\mathbf{x}_0$ (o una población inicial válida $\mathbf{X}_0$).

Los problemas de optimización que GENOCOP trata están definidos en un dominio convexo, que pueden ser formulados de la siguiente manera:

$$\text{optimizar} f(x_1, \ldots, x_q) \in \mathbb{R} \tag{7.40}$$

donde $(x_1, \ldots, x_q) \in \mathbb{X} \subseteq \mathbb{R}^q$ y $\mathbb{X}$ es un conjunto *convexo.*

El dominio $\mathbb{X}$ está definido por rangos de variables $l_k \leq x_k \leq r_k, \forall k = 1,\ldots,q$ y un conjunto de restricciones $\mathcal{C}$. Dado que el conjunto $\mathbb{X}$ es convexo, podemos afirmar que, para cada punto $(x_1,\ldots,x_q) \in \mathbb{X}$ existe un rango válido $\langle \text{left}(k), \text{right}(k)\rangle$ de la variable $x_k (1 \leq k \leq q)$ para el cual otras variables $x_i (i = 1,\ldots,k-1,k+1,\ldots,q)$ permanecen fijas. En otras palabras, para un dado $(x_1,\ldots,x_k,\ldots,x_q) \in \mathbb{X}$ se cumple que:

$$y \in \langle \text{left}(k), \text{right}(k)\rangle \longrightarrow (x_1,\ldots,x_{k-1},y,x_{k+1},\ldots,x_q) \in \mathbb{X} \tag{7.41}$$

donde todos los $x_i (i = 1,\ldots,k-1,k+a,\ldots,q)$ permanecen constantes. También se asume que cada intervalo $\langle \text{left}(k), \text{right}(k)\rangle$ puede ser calculado eficientemente.

Si el conjunto de restricciones $\mathcal{C} = \emptyset$, entonces el espacio de búsqueda $\mathbb{X} = \prod_{k=1}^{q} \langle l_k, r_k\rangle$ será convexo apenas limitado por las restricciones de dominio: $\text{left}(k) = r_k, \text{right}(k) = r_k, \forall k = 1,\ldots,q$.

Esta propiedad es aprovechada en todos los operadores de mutación y garantiza que los resultados sean siempre válidos. Si se muta la variable x_k, el rango de mutación será confinado al intervalo $\langle l_k, r_k\rangle = \langle \text{left}(k), \text{right}(k)\rangle$.

Otra propiedad de los espacios convexos garantiza que, dados dos puntos $x_1, x_2 \in \mathbb{X}$, la combinación lineal $\alpha x_1 + (1-\alpha)x_2$ donde $\alpha \in [0,1]$ es también un punto en $\mathbb{X}$ (de la propia definición de convexidad). Esta propiedad permite definir operadores de cruce.

El problema de optimización de la ecuación 7.40 definido en un espacio convexo que trata GENOCOP es el siguiente:

Optimizar una función $f(\mathbf{x})$, donde $\mathbf{x} = (x_1, x_2, \ldots, x_q)$, sujeta a los siguientes conjuntos de restricciones:

1. Restricciones de dominio $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ donde $\mathbf{l} = (l_1,\ldots,lq)$, $\mathbf{u} = (u_1,\ldots,u_q)$, $\mathbf{x} = (x_1,\ldots,x_q)$
2. Igualdades $\mathbf{Ax} = \mathbf{B}$, donde $\mathbf{x} = (x_1,\ldots,x_q)$, $\mathbf{A} = \{a_{ij}\}$, $\mathbf{B} = (b_1,\ldots,b_p)$, $1 \leq i \leq p$ y $1 \leq j \leq q$ (p es el número de ecuaciones).
3. Inecuaciones $\mathbf{Cx} \leq \mathbf{d}$, donde $\mathbf{x} = (x_1,\ldots,x_q)$, $\mathbf{C} = \{c_{ij}\}$, $\mathbf{d} = (d_1,\ldots,d_m)$, $1 \leq i \leq m$, y $1 \leq j \leq q$ (m es el numero de inecuaciones).

Esta definición de problema es genérica y utilizada por gran parte de los algoritmos de optimización aplicados en investigación operativa con restricciones lineales y cualquier función objetivo.

Tratamiento de Restricciones por los operadores GENOCOP

GENOCOP ofrece una forma generalizada de manipular las restricciones e independiende del problema. Esto se logra combinando ideas que son propias de Algoritmos Genéticos y de la teoría clásica de problemas de Optimización Numérica en un contexto diferente. Así, dado un problema de optimización numérica, en GENOCOP se hace lo siguiente:

1. Eliminar las igualdades presentes en el conjunto de restricciones, esto significa eliminar p de variables del problema, suponiendo que existan p de restricciones de igualdad $g(x_1,\ldots,x_q) = 0$. La dimensión del problema se reducirá a $q-p$ y las nuevas restricciones resultantes serán inecuaciones lineales que mantienen la convexidad del espacio de búsqueda $\mathbb{G}$. El espacio de búsqueda $\mathbb{G}$ resulta más simple que el espacio original del problema $\mathbb{X}$.
2. Obtener un modelo cuidadoso de los operadores genéticos tal que se

garantize la validez de los individuos descendientes en el espacio de búsqueda $\mathbb{G}$ con restricciones.

Este tratamiento funciona eficientemente cuando las restricciones son lineales, lo que es una condición suficiente para que el espacio de problema $\mathbb{X}$ sea convexo.

En las técnicas de Programación Lineal, las restricciones de igualdad son bienvenidas siempre que se asuma que sí existe el óptimo, esté situado en la superficie del conjunto convexo. Las inecuaciones se convierten en ecuaciones usando variables de holgura y el método se ejecuta moviéndose de vértice en vértice a lo largo de la superficie.

En la metodología de GENOCOP, las soluciones se generan aleatoriamente y las restricciones de igualdad significan problemas. Al ser eliminadas al inicio del proceso se reduce la dimensionalidad del problema en p y por ende, se reduce la dimensionalidad del espacio de búsqueda $\mathbb{G} \subseteq \mathbb{R}^{q-p}$. En este espacio de búsqueda reducido se encuentra sólo restricciones de desigualdad que mantienen la convexidad del mismo. Se asegura que cualquier combinación lineal de soluciones generará soluciones válidas sin necesidad de verificar las restricciones. Con las inecuaciones se generan *intervalos factibles* para cualquier variable x_i, estos intervalos factibles son dinámicos y dependen de los valores actuales de las demás variables y pueden calcularse de forma eficiente

■ **Example 7.5** El siguiente ejemplo ilustra el mecanismo incorporado en el modelo GENOCOP.

Sea la función $f(\mathbf{x})$:

$$f(\mathbf{x}) = \sum_{j=1}^{10} \left(c_j + \ln \frac{x_j}{x_1 + \ldots + x_{10}} \right)$$

a ser minimizada y sujeta al siguiente conjunto de restricciones:

$$\begin{aligned} x_1 + 2x_2 + 2x_3 + x_6 + x_{10} &= 2 \\ x_4 + 2x_5 + x_6 + x_7 &= 1 \\ x_3 + x_7 + x_8 + 2x_9 + x_{10} &= 1 \\ 0{,}000001 \leq x_i \leq 1{,}0 \quad , \quad i &= 1, \ldots, 10 \end{aligned}$$

donde $c_1 = -6{,}089$, $c_2 = -6{,}089$, $c_3 = -34{,}054$, $c_4 = -5{,}914$, $c_5 = -24{,}721$, $c_6 = -14{,}986$, $c_7 = -24{,}100$, $c_8 = -10{,}708$, $c_9 = -26{,}6662$, $c_{10} = -222{,}179$,

Al observar las restricciones, se ve que pueden eliminarse tres variables, x_1, x_2, x_4:

$$\begin{aligned} x_1 &= -2x_2 - x_6 + 2x_7 + 2x_8 + 4x_9 + x_{10} \\ x_3 &= 1 - x_7 - x_8 - 2x_9 - x_{10} \\ x_4 &= 1 - 2x_5 - x_6 - x_7 \end{aligned}$$

y dado que $0{,}000001 \leq x_i \leq 1{,}0$, el nuevo problema acaba siendo mininizar $f(x_2, x_5, x_6, x_7, x_8, x_9, x_{10})$

$$
\begin{aligned}
= \quad & (-2x_2 - x_6 + 2x_7 + 2x_8 + 4x_9 + x_{10}) \left(c_1 + \ln \tfrac{-2x_2 - x_6 + 2x_7 + 2x_8 + 4x_9 + x_{10}}{2 - x_2 - x_5 - x_6 + x_7 + 2x_8 + 3x_9 + x_{10}}\right) \\
& + x_2 \left(c_2 + \ln \tfrac{x_2}{2 - x_2 - x_5 - x_6 + x_7 + 2x_8 + 3x_9 + x_{10}}\right) \\
& + (1 - x_7 - x_8 - 2x_9 - x_{10}) \left(c_3 + \ln \tfrac{1 - x_7 - x_8 - 2x_9 - x_{10}}{2 - x_2 - x_5 - x_6 + x_7 + 2x_8 + 3x_9 + x_{10}}\right) \\
& + (1 - 2x_5 - x_6 - x_7) \left(c_4 + \ln \tfrac{1 - 2x_5 - x_6 - x_7}{2 - x_2 - x_5 - x_6 + x_7 + 2x_8 + 3x_9 + x_{10}}\right) \\
& + \textstyle\sum_{j=5}^{10} x_j \left(c_j + \ln \tfrac{x_j}{2 - x_2 - x_5 - x_6 + x_7 + 2x_8 + 3x_9 + x_{10}}\right)
\end{aligned}
$$

sujeto a las siguientes restricciones:

$$
\begin{aligned}
0{,}000001 \quad &\leq \quad -2x_2 - x_6 + 2x_7 + 2x_8 + 4x_9 + x_{10} \leq 1{,}0 \\
0{,}000001 \quad &\leq \quad 1 - x_7 - x_8 - 2x_9 - x_{10} \leq 1{,}0 \\
0{,}000001 \quad &\leq \quad 1 - 2x_5 - x_6 - x_7 \leq 1{,}0 \\
0{,}000001 \quad &\leq \quad x_i \leq 1{,}0, \qquad i = 2, 5, 6, 7, 8, 9, 10
\end{aligned}
$$

Luego de hacer todas estas transformaciones, el sistema ya puede crear su población inicial válida $\mathbf{X}_0$ y realizar el proceso de evolución en sí. ∎

Inicializacion de la población Inicial

Una vez reducida la dimensionalidad al eliminar p restricciones de igualdad, el algoritmo de inicialización consiste en generar aleatoriamente los individuos $\mathbf{x} \in \mathbb{X}$ que cumplan las restricciones de dominio usando

$$x_k = \text{left}(k) + (\text{right}(k) - \text{left}(k))\, \mathbf{U}(t), \forall k = 1, \ldots, n \tag{7.42}$$

y verificando que cumplan las restricciones de inecuaciones.
De forma iterativa se genera valores $\mathbf{x}$ hasta encontrar un elemento válido o hasta agotar el número máximo de iteraciones. Una vez encontrado un individuo válido $\mathbf{x}_0$, es replicado en toda la población $\mathbf{X}_S$. En el caso que no se logre ningún individuo, se solicitará ingresar manualmente un individuo válido.

Operadores Genéticos

En Genocop se definen 6 operadores genéticos adecuados para tratar con representación real. Los tres primeros son mutaciones unarias y los demás son cruces que se describen y discuten a continuación.

Mutación Uniforme

Este operador requiere un único padre $\mathbf{x}$ y genera un único descendiente $\mathbf{x}'$ de la siguiente manera:

1. Dado el vector $\mathbf{x} = (x_1, \ldots, x_k, \ldots, x_n)$, seleccionar un componente aleatorio $x_k, k \in (1, \ldots, n)$.
2. Generar el elemento nuevo $\mathbf{x}' = (x_1, \ldots, x'_k, \ldots, x_n)$ donde x'_k es obtenido de una generación de valor aleatorio con distribución uniforme $\mathbf{U}(t)$ dentro del intervalo $\langle \text{left}(k), \text{right}(k) \rangle$.

El operador de mutación uniforme juega un papel importante en fases iniciales del proceso de evolución, al permitir que los individuos se muevan libremente en el espacio de búsqueda. Este operador se hace más relevante dado que la población inicial está compuesta por réplicas de un punto válido. En fases más finales de la evolución, este operador permite escapar de óptimos locales dando la opción de encontrar mejores puntos.

Mutación de Frontera

Este operador también requiere un único padre $\mathbf{x}$ y genera un unico descendiente $\mathbf{x}$. En realidad se trata de una variación de la mutación uniforme pues el nuevo valor x'_k será cualquiera de los valores dentro del intervalo válido $\langle \text{left}(k)\rangle$, right$(k)$, como se describe a continuación.

1. Dado el vector $\mathbf{x} = (x_1, \ldots, x_k, \ldots, x_n)$, seleccionar un componente aleatorio $x_k, k \in (1, \ldots, q)$.
2. Generar un valor $b = \mathbf{U}(t) \in [0, 1]$.
3. Generar el nuevo elemento $\mathbf{x}' = (x_1, \ldots, x'_k, \ldots, x_n)$ donde :

$$x'_k = \begin{cases} \text{left}(k) & \text{si } b \leq 0{,}5 \\ \text{right}(k) & \text{si } b \geq 0{,}5 \end{cases} \tag{7.43}$$

Su importancia radica en problemas cuya solución óptima se encuentra muy cerca de alguna de las fronteras del espacio de búsqueda, y también en problemas con muchas restricciones. Sin embargo en un problema sin restricciones ($C = \emptyset$) y con dominios amplios, este operador no es recomendable.

Mutación no Uniforme

Es un operador unario que permite realizar ajustes finos del sistema. Se define de la siguiente manera:

1. Dado un padre $\mathbf{x}$, se selecciona un componente x_k
2. Generar um valor $b = \mathbf{U}(t) \in [0, 1]$.
3. Se obtiene el descendiente $\mathbf{x}' = (x_1, \ldots, x_k, \ldots, x_n)$ donde:

$$x'_k = \begin{cases} x_k + \Delta(t, \text{right}(k) - x_k) & \text{si } b \leq 0{,}5 \\ x_k - \Delta(t, x_k - \text{left}(k)) & \text{en caso contrario} \end{cases} \tag{7.44}$$

4. La función $\Delta(t, y)$ devuelve un valor en el intervalo $[0, y]$, tal que la probabilidad de $\Delta(t, y)$ sea próxima a 0 y vaya incrementándose de acuerdo a t que representa la generación en curso. Con esa propiedad, este operador se comporta como una mutación unifore al inicio (con t pequeño) y, a medida que t aumenta, lá búsqueda se hace más local. Para esto se usa la siguiente función

$$\Delta(t, y) = y.r. \left(1 - \frac{t}{T}\right)^b \tag{7.45}$$

donde r es un número aleatorio con distribución $\mathbf{U}(t), T$ es el número máximo de generaciones y b es un parámetro que determina el grado de no uniformidad.

Cruce aritmético

Este operador es definido como la combinacion lineal de dos vectores:

1. Sean dos vectores $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{X}$ seleccionados para cruzarse
2. Los descendientes se obtienen como la combinación lineal de $\mathbf{x}_1$ y $\mathbf{x}_2$ conforme la ecuacion 7.46:

$$\begin{aligned} \mathbf{x}'_1 &= \alpha \mathbf{x}_1 + (1 - \alpha)\mathbf{x}_2 \\ \mathbf{x}'_2 &= (1 - \alpha)\mathbf{x}_1 + \alpha \mathbf{x}_2 \end{aligned} \tag{7.46}$$

donde α es un valor aleatorio con distribución $\mathbf{U}(t) \in [0, 1]$. Con este valor de α y dado que espacio $\mathbb{X}$ es convexo, queda garantizada la clausura $\mathbf{x}'_1, \mathbf{x}'_2 \in \mathbb{X}$

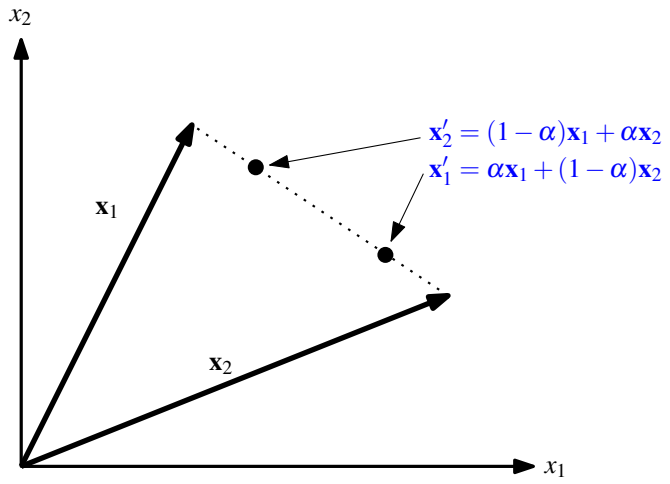


Figura 7.18: Cruce Aritmético

3. Si $\alpha = 0{,}5$ los descendientes $\mathbf{x}_1' = \mathbf{x}_2' = \bar{\mathbf{x}}$ (media aritmética de los padres). La figura 7.18 ilustra este operador.

Cruce Simple

Se define este operador de la siguiente manera:

1. Dados $\mathbf{u} = (u_1, \ldots, u_n), \mathbf{v} = (v_1, \ldots, v_n)$ a cruzarse en la k-ésima posición.
2. A priori, los descendientes serían:

$$\begin{aligned} \mathbf{x}_1' &= (u_1, \ldots, u_k, v_{k+1}, \ldots, v_n) \\ \mathbf{x}_2' &= (v_1, \ldots, v_k, u_{k+1}, \ldots, u_n) \end{aligned} \tag{7.47}$$

   que podrían estar fuera del espacio de problema $\mathbb{X}$. Para evitar este inconveniente, se aprovecha una propiedad de los conjuntos convexos que afirma que $\exists \alpha \in [0,1]$ tal que los elementos

$$\begin{aligned} \mathbf{x}_1' &= (u_1, \ldots, u_k, v_{k+1}\alpha + u_{k+1}(1-\alpha), \ldots, v_n\alpha + u_n(1-\alpha)) \\ \mathbf{x}_2' &= (v_1, \ldots, v_k, u_{k+1}\alpha + v_{k+1}(1-\alpha), \ldots, u_n\alpha + v_n(1-\alpha)) \end{aligned} \tag{7.48}$$

   son válidos.
3. Aún queda por encontrarse el mayor valor posible de α tal que se haga el mayor intercambio de información posible. Se usa un método simple que consiste en comenzar con $\alpha = 1$ y verificar si al menos un $\mathbf{x}_i' \notin \mathbb{X}$, e ir decrementando $\alpha = \frac{\alpha}{\rho}$. Cuando $\alpha \to 0$, los dos descendientes tienden a ser idénticos a su padres y perteneciendo al dominio válido $\mathbb{X}$.

La Figura 7.19, nos ilustra este procedimiento.

Cruce heurístico

Es un operador de cruce con único elemento por los siguientes motivos:

1. Este operador usa resultados de la función objetivo $f(\mathbf{x}_i)$ para determinar la dirección de búsqueda.
2. Genera un único descendiente o puede no generar descendientes

El procedimiento es el siguiente

1. Dados dos padres $\mathbf{x}_1, \mathbf{x}_2$, se obtiene un único descendiente $\mathbf{x}_3$ usando la siguiente regla:

$$\mathbf{x}_3 = r(\mathbf{x}_2 - \mathbf{x}_1) + \mathbf{x}_2 \tag{7.49}$$

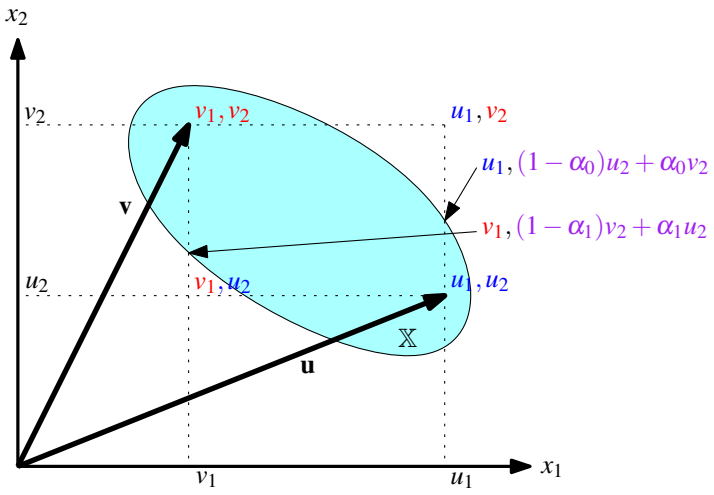


Figura 7.19: Cruce Simple

donde $r = \mathbf{U}(t) \in [0,1]$ y el individuo $\mathbf{x}_2$ es mejor o igual que $\mathbf{x}_1$. O sea $f(\mathbf{x}_2) \geq f(\mathbf{x}_1)$ si se trata de una maximización y $f(\mathbf{x}_2) \leq f(\mathbf{x}_1)$ si el problema es de minimización.

2. Es posible que el resultado $\mathbf{x}_3 \notin \mathbb{X}$, en este caso se genera otro descendiente $\mathbf{x}_3$ usando una nueva realización de r.
3. Si después de w tentativas no se encuentra un $\mathbf{x}_3$ válido, el operador finaliza sin producir descendientes.

En la figura 7.20 se ilustra este operador.

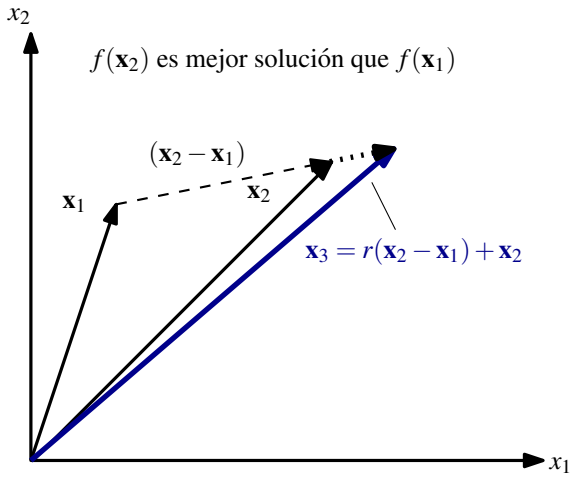


Figura 7.20: Cruce Heurístico

Este operador contribuye en la precisión de la solución encontrada por el ajuste fino y por dirigir la búsqueda en la dirección más promisoria.

7.4.4 Restricciones no lineales – GENOCOP III

Esta nueva versión de GENOCOP incorpora el sistema GENOCOP original ya descrito y lo extiende para tratar con restricciones no lineales, aplicando el principio de coevolución usando dos poblaciones que se influyen mutuamente:

- La primera población $\mathbf{X}_S = \{\mathbf{s}_1, \ldots, \mathbf{s}_m\}$ se denomina *puntos de busqueda*

que sólo satisfacen las restricciones lineales, de forma similar al GENOCOP original. Estos puntos pueden ser tratados con los operadores especializados del GENOCOP original manteniendo su validez (para las restricciones lineales), aunque no puedan ser evaluados.

- La segunda población $\mathbf{X}_R = \{\mathbf{r}_1, \ldots, \mathbf{r}_n\}$ denominada *puntos de referencia* consta de puntos que satisfacen todas las restricciones del problema. Para cualquier punto $\mathbf{r}_i$ siempre es posible evaluar $f(\mathbf{r}_i)$. Habiendo dificultad en encontrar un punto de referencia inicial $\mathbf{r}_0$, se le pedirá al usuario que ingrese un valor.

La figura 7.4.4 ilustra ambas poblaciones Los puntos de referencia $\mathbf{r}_i$ al ser

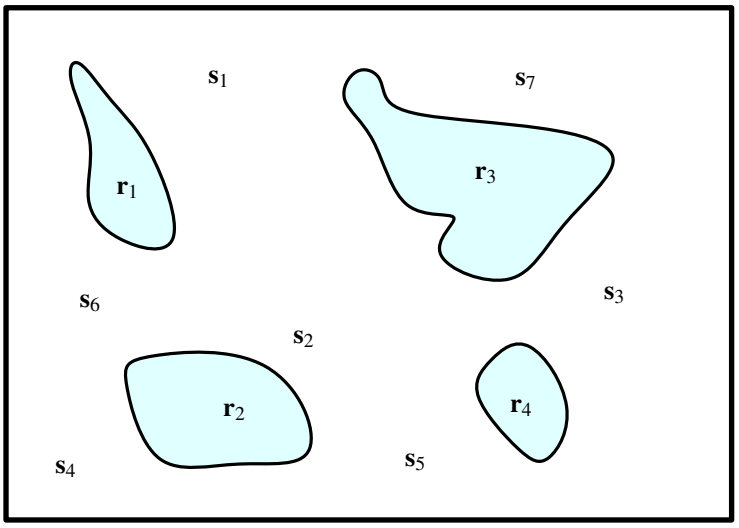


Figura 7.21: Población de búsqueda $\mathbf{X}_S$ y Población de Referencia $\mathbf{X}_R$

válidos, son evaluados directamente por la función objetivo $f(\mathbf{r}_i)$. Los puntos de búsqueda $\mathbf{s}_i$ que no pueden ser evaluados se reparan para convertirlos en puntos válidos como se describe a continuación:

1. Sea un punto de búsqueda $\mathbf{s}_k \in \mathbf{X}_S$ que no es válido
2. Se selecciona un punto de referencia $\mathbf{r}_l \in \mathbf{X}_R$ usando un método de selección de *ranking* no lineal
3. Se generan varios valores nuevos $\alpha_i = \mathbf{U}(i) \in [0,1]$
4. Se obtiene nuevos puntos de búsqueda $\mathbf{z}_i$ combinando linealmente $\mathbf{s}_k$ y $\mathbf{r}_l$ usando los diversos α_i:

$$\mathbf{z}_i = \alpha_i \mathbf{s}_k + (1 - \alpha_i)\mathbf{r}_l \tag{7.50}$$

5. Si se encuentra un $\mathbf{z}_i$ válido, se evalua y el valor obtenido se coloca como una pseudo-evaluación de $\mathbf{s}_k$.
6. Si $f(\mathbf{z}_i) \geq f(\mathbf{r}_l)$, entonces $\mathbf{z}_i$ reemplaza a $\mathbf{r}_l$ como nuevo punto de referencia. Eventualmente $\mathbf{z}_i$ reemplaza a $\mathbf{r}_l$ usando un sorteo com probabilidad de reemplazo p_r.

La Figura 7.22 ilustra la reparación de puntos de búsqueda
El modelo GENOCOP III supera muchas desventajas de otros sistemas introduciendo sólo algunos parámetros adicionales (tamaño de la población de puntos de referencia $\mathbf{X}_R$, probabilidad de reemplazo p_r) y retornando siempre una solución válida. Además, el espacio de búsqueda válido es explorado

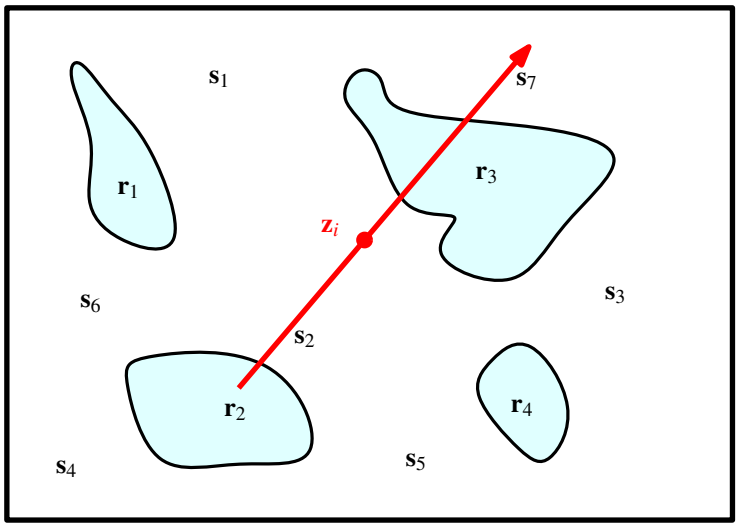


Figura 7.22: Reparación de puntos de búsqueda

creando nuevos puntos referencia a patrir de los puntos de búsqueda. La vecindad de los mejores puntos de referencia es explorada con más frecuencia. También, algunos puntos de referencia son colocados en la población $\mathbf{X}_s$ de puntos de búsqueda, sometiéndose a ser transformados por los operadores especializados de GENOCOP que preservan las restricciones lineales.

7.5 Computación Evolutiva en optimización combinatoria

Como fue visto en las secciones anteriores, el objetivo de un problema de optimización es maximizar o minimizar el valor de una función real en un espacio. Si se trata de funciones del tipo $f : \mathbb{X} \subseteq \mathbb{S} \subset \mathbb{R}^n \mapsto \mathbb{R}$, el espacio de búsqueda es continuo y la optimización es numérica. Este tipo de tratamiento se detalló en la sección 7.4 anterior y su problema de optimización consiste en encontrar :

$$\arg\max f(\mathbf{x}), \mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{X} \subseteq \mathbb{R}^n \tag{7.51}$$

donde $f(\mathbf{x})$ está sujeto a las restricciones $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}, \mathbf{Ax} = \mathbf{b}, \mathbf{Cx} = \mathbf{d}$
En cambio, en problemas de optimización combinatoria se busca optimizar una función

$$f(\mathbf{x}), \mathbf{x} = (x_1, \ldots, x_n) \in \Phi \subset \mathbb{N}^n \tag{7.52}$$

donde Φ es un espacio de soluciones, discreto y finito que puede ser formado de las siguientes maneras:

1. Como el producto cartesiano de un subconjunto de n valores $N \subset \mathbb{N} = \{x_1, \ldots, x_n\}$, $x_i \in N, \forall i = 1 \ldots n$
2. Como el conjunto de permutaciones ϕ_i de un conjunto de n valores $N \subset \mathbb{N} = \{x_1, \ldots, x_n\}$, $x_i \in N, \forall i = 1 \ldots n, x_i \neq x_j, \forall i \neq j$.

Este último tipo de problemas son un tipo especial de optimización combinatoria conocido como Problemas de Orden.

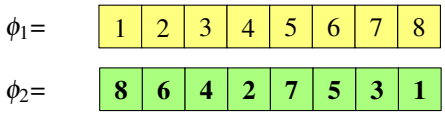


Figura 7.23: Ejemplos de cromosomas que representan permutaciones ϕ

7.5.1 Algoritmos Evolutivos Discretos

Son algoritmos evolutivos cuyos cromosomas $\mathbf{x} \in \Phi$ están compuestos por genes con valores discretos $x_i \in N \subset \mathbb{N}$.

Definition 7.5.1 Algoritmos evolutivos discretos

Sea una generación t con una población $\mathbf{X}_t = \{x_i\}_{i=1}^{m}$.

2. Cada cromosoma $\mathbf{x}_i \in \mathbf{X}_t$ está compuesto por un conjunto de genes $\mathbf{x} = (x_1, \ldots, x_n)$ donde cada elemento x_j puede tomar alguno de los valores del conjunto $N = \{1, \ldots n\} \subset \mathbb{N}$.
3. Es decir que el espacio de problema $\mathbb{X}$ está formado por el producto cartesiano $N \times N \times \ldots \times N \subset \mathbb{N}^n$
4. En un caso más genérico, dado un cromosoma $\mathbf{x}$, cada gene $x_k \in \mathbf{x}$ puede asumir valores del conjunto $N_k = \{i\}_{i=1}^{n_i}$ haciendo que el espacio de búsqueda sea igual al espacio del problema, $\mathbb{G} = \mathbb{X}$ y esté formado por el producto cartesiano $N_1 \times N_2 \times \ldots \times N_n \subset \mathbb{N}^n$.

Algoritmo Genético Binario

Bajo la definición explicada, el algoritmo genético binario es un caso particular de un algoritmo evolutivo discreto, con las siguientes propiedades:

- cuando $N_k = \mathbb{B} = \{i\}_{i=0}^{1}, \forall k = 1, \ldots, n$, conforme ya fue tratado en 7.3.4.

7.5.2 Algoritmos Evolutivos de Orden

Son algoritmos evolutivos cuya poblacion $\mathbf{X}_t$ está compuesta por individuos que son permutaciones ϕ del subconjunto finito $N = \{x_1, \ldots, x_n\} \subset \mathbb{N}$ de tamaño n.

Definition 7.5.2 Algoritmo Evolutivo de orden

Para una generación t, sea Φ_t una población de tamaño m, $\Phi_t = \{\phi_i\}_{i=1}^{m}$.

- Cada cromosoma $\phi_i \in \Phi_t$ está compuesto por un conjunto de genes. $\phi_i = \{\phi_i(1), \ldots, \phi_i(n)\}$, donde un elemento $\phi_i(j)$ es el elemento encontrados en la j-ésima posición de la permutación ϕ_i. Es claro que para dos cromosomas ϕ_p y ϕ_q, con $p \neq q$ no se cumplirá necesariamente que $\phi_p(j) = \phi_q(j)$.

Un algoritmo evolutivo de orden tiene la misma estructura que el Algoritmo Genético Tradicional o el Algoritmo con codificación Real ya que se mantienen los componentes básicos: población, selección, evaluación, reproducción. La diferencia principal radica en los siguientes puntos :

1. La representación de soluciones, cuyos cromosomas son permutaciones ϕ de los n elementos del conjunto $N \in \mathbb{N}$. Como ejemplo suponiendo permutaciones de 5 elementos y un conjunto $N = \{1, 2, 3, 4, 5\}$, tendríamos algunos cromosomas como ilustra la figura 7.23:
2. Las permutaciones por sí solas no representan soluciones. Existe la nece-

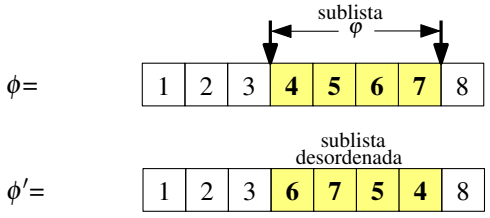


Figura 7.24: Ejemplo de mutación de desorden

sidad de un **constructor de soluciones** que, a partir de las permutaciones y aplicando reglas y restricciones del problema codifica las soluciones reales que siempre serán válidas y posibles de ser evaluadas.

3. Los operadores de cruce y mutación son especializados para tratar los cromosomas ϕ_i, garantizando que sus descendientes sean siempre válidos.

Mutación de Desorden

Dado un único progenitor $\phi_1 = \{\phi_1(1), \ldots \phi_1(n)\}$,

1. Se selecciona al azar una sub-lista φ del cromosoma ϕ_1.
2. Se desordenan los elementos en esta sub-lista.

La figura 7.24 ilustra este tipo de operador.

Mutación basada en Orden

Dado un progenitor $\phi_1 = \{\phi_1(1), \ldots, \phi_1(n)\}$:

1. Se seleccionan dos componentes aleatorios $\phi_1(a), \phi_1(b) \in \phi_1$ donde $a < b$
2. El descendiente ϕ_1' se forma anteponiendo $\phi_1(b)$ a $\phi_1(a)$

Un ejemplo ilustrativo de este operador se muestra en la figura 7.25

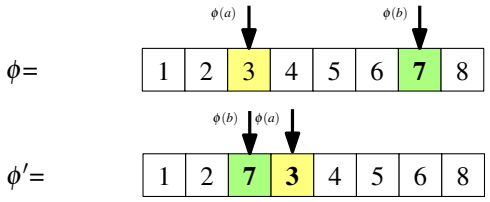


Figura 7.25: Ejemplo de mutación de orden

Mutación basada en Posición

Dado un progenitor $\phi_1 = \{\phi_1(1), \ldots, \phi_1(n)\}$:

1. Se seleccionan dos componentes aleatorios $\phi_1(a), \phi_1(b) \in \phi_1$
2. Se forma el descendiente ϕ_1' intercambiando los valores de $\phi_1(a), \phi_1(b)$.

La figura 7.26 nos da un ejemplo ilustrativo de este operador

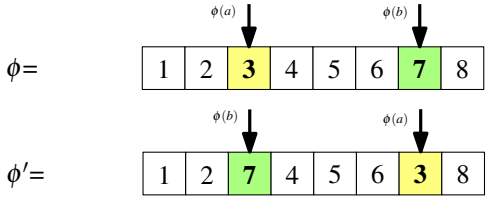


Figura 7.26: Ejemplo de mutación basada en posición

Cruce Uniforme de Orden

Dados los cromosomas ϕ_1, ϕ_2, se obtiene un descendiente ϕ'_1 de la siguiente forma:

1. Obtener una máscara $\mathbf{m}_k$ de n *bits* binarios
2. Rellenar ϕ'_1 copiando los genes de ϕ_1 en las posiciones con *bits* 1, respetando sus posiciones originales.
3. Crear una sublista φ con los elementos de ϕ_1 con *bits* 0
4. Reordenar la sublista φ para que sus valores se vean en el mismo orden que ϕ_2.
5. Completar los espacios faltantes en ϕ'_1 con la lista φ reordenada.
6. El otro descendiente ϕ'_2 se compondra aplicando el mismo procedimiento para una máscara negada $\overline{\mathbf{m}}_k$ y aplicando el orden encontrado en ϕ_1 para la sublista φ.

La figura 7.27 ilustra este tipo de operador

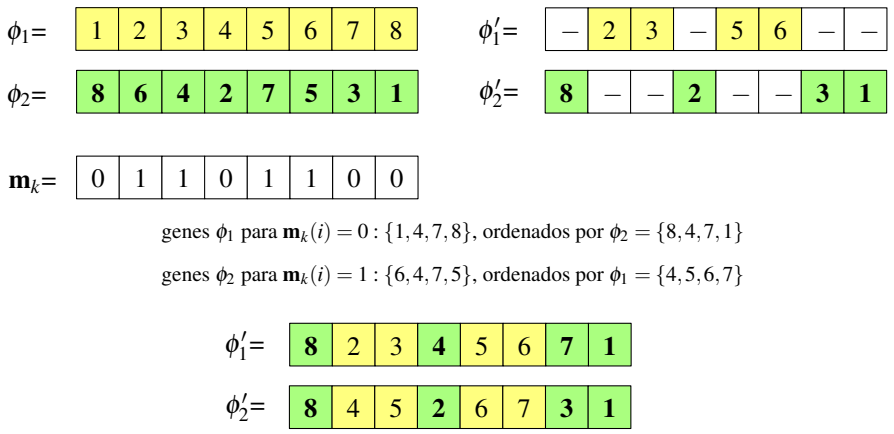


Figura 7.27: Cruce Uniforme de Orden

Cruce de sublista

Bastante similar al cruce uniforme de orden, dados los cromosomas ϕ_1, ϕ_2, se obtiene un descendiente ϕ'_1 de la siguiente forma:

1. Separar una sublista φ del cromosoma ϕ_1.
2. Crear el descendiente ϕ'_2 rellenando los espacios dejados por la sublista φ con sus mismos valores respetando el orden en el que se encuentran en el cromosoma ϕ_2.
3. El otro descendiente ϕ'_2 se compondra aplicando el mismo procedimiento en los individuos de ϕ_2 que no pertenecen a las posiciones de la sublista, recolocándolos en el orden encontrado en el cromosoma ϕ_1.

7.5.3 Problemas de Optimización Combinatoria

El Problema de la Mochila

Conocido como *The knapsack problem* [**Mathews1896**], consiste en un excursionista que tiene n objetos para llevar en su mochila, pero debe decidir cuáles colocará. Cada objeto tiene un valor v_j y un peso w_j. La mochila soporta un peso máximo W. El problema, ilustrado en la figura 7.28 consiste en seleccionar de entre los n objetos los que colocará tal que den el máximo beneficio sin sobrepasar el peso máximo soportado por la mochila.

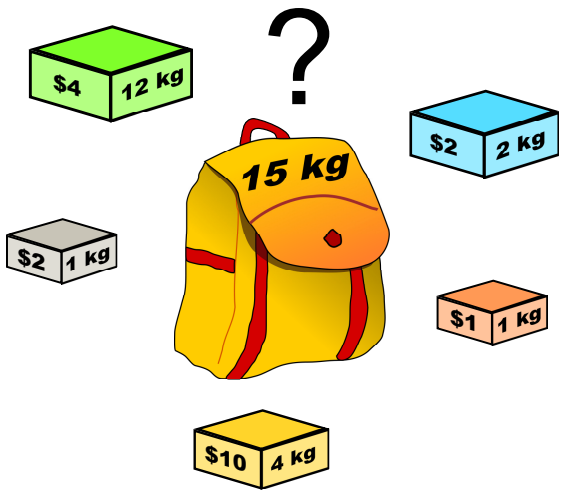


Figura 7.28: *The knapsack problem*

Formulación Clásica

El problema de la mochila se puede formular de la siguiente manera:

1. Sea $\mathbf{x} = \{x_1, \ldots, x_n\}$ un vector, cuyos elementos $x_i \in \{0,1\}$ están relacionados con cada objeto i a colocarse en la mochila, de acuerdo a la siguiente definición:

$$x_i = \begin{cases} 1 & \text{objeto } i \text{ colocado} \\ 0 & \text{objeto } i \text{ no colocado} \end{cases} \tag{7.53}$$

2. Entonces, el problema consiste en maximizar el valor total cargado, es decir:

$$V_{tot} = \text{máx} \sum_{i=1}^{n} v_i x_i \tag{7.54}$$

3. Sujeto a la restricción

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{7.55}$$

Esta forma de modelar que siendo típica de un modelo de Programación Entera 0/1, es fácilmente llevada a una forma evolutiva si se define el individuo $\mathbf{x} = \{x_1, \ldots, x_n\}$ que pertenece a un espacio de búsqueda $\mathbb{B}^n$. Así, se pueden aplicar los métodos de inicialización, selección, operadores de cruce y mutación que ya fueron estudiados en los algoritmos evolutivos con codificación binaria.

Cabe resaltar que en esta modelación, el principal problema es el manejo de la restricción de la ecuación 7.55.

Formulación como problema de orden

El problema de la mochila puede verse de la siguiente manera:

1. Sea $\phi_i = \{\phi_i(1), \ldots, \phi_i(n)\}$ un cromosoma de permutación que representa un orden i de los objetos a colocar en la mochila.
2. Se construye una solución i asociada a la permutación ϕ_i colocando los objetos en el orden dado por $\phi_i(1), \phi_i(2), \ldots, \phi_i(k)$ donde el elemento k,

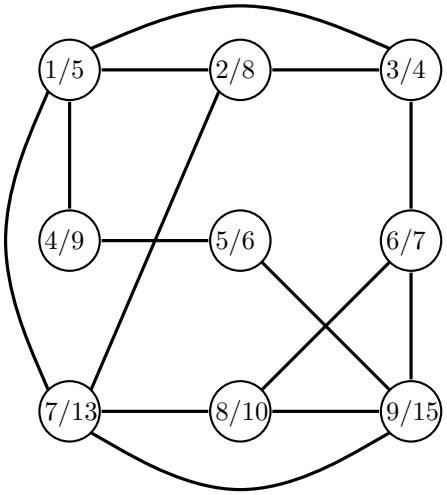


Figura 7.29: Ejemplo de grafo conectado

es el último a colocar sin rebasar el peso máximo permitido W. Los elementos restantes $\phi_i(k+1), \phi_i(k+2), \ldots, \phi_i(n)$ restantes no se consideran en la solución.

3. La evaluación será precisamente la suma de los pesos de los objetos efectivamente colocados en la mochila.

$$V_{tot} = \sum_{i=1}^{k \leq n} v_{\phi(i)} \tag{7.56}$$

donde $k \leq n$ es la cantidad de objetos efectivamente colocados en la mochila.
Es interesante destacar que con esta codificación del problema y el uso de un constructor de solucions, cualquier cromosoma generará una solución válida.

Problema de k-coloreo de Grafos

Un ejemplo de aplicación de un algoritmo genético de orden es el problema de coloreo de grafos que se describe a continuación:
Sea un grafo no dirigido $\mathcal{G}(V, E)$, como el de la figura 7.29, cuyos vértices $V_i(G)$ son ponderados con un peso p_i.
Existe una función $f : V \mapsto \mathbb{N}$ que relaciona vértices y pesos la cual consiste en los coloreos válidos hasta k colores donde no se puede usar un mismo color entre nodos adyacentes. El valor de f puede ser la suma de los pesos de los nodos coloreados.
Una optimización a realizar es colorear nodos (o vértices) de este grafo de forma tal que se maximice f, o sea la suma total de los pesos.
Este tipo de problemas es NP-completo que puede reducirse a tiempo polinomial para grafos perfectos o grafos sin vértices de grado ≥ 4.
Este tipo de problemas sirven como base para los problemas de asignación de recursos que son muy aplicados en el mundo real. Aún más, si los vértices del grafo son ponderados, se trata de recursos cuandificables, donde un recurso es mejor que otro.

Algoritmo *greedy*
El algoritmo *greedy* es simple para el problema de colorear grafos y es el que ilustra el algoritmo 12

Algorithm 12 *Algoritmo greedy*

Lista de colores $C = \{c_j\}, j = 1, \ldots\ n_c$ **ordenar** los n nodos $V(G)$ del grafo de mayor a menor por su peso p $j = 0$ $j < n_c$ $i = 0$ $i < n$ **verificar** si se puede colorear con c_j el nodo $V_i(G)$ **colorear** $V_i(G)$ si la verificación es válida $i = i + 1$ $j = j + 1$

Este algoritmo se caracteriza por su simplicidad y facilidad de implementación, esperándose llegar a la solución óptima global, aunque dada su visión local (ver la mejor decisión inmediata), no siempre conducirá a la solución óptima global.
Aplicando al grafo de la figura 7.29, la lista de nodos en orden decreciente de peso será

$$V = \{v_9, v_7, v_8, v_4, v_2, v_6, v_5, v_1, v_3\}$$

Si la aplicamos para $k = 1$, tenemos la siguiente solución:

$$V^*_{k=1}\{v_9, v_4, v_2\} \longrightarrow \sum p_i = 32$$

Cuyo gráfico se ilustra en la figura 7.30

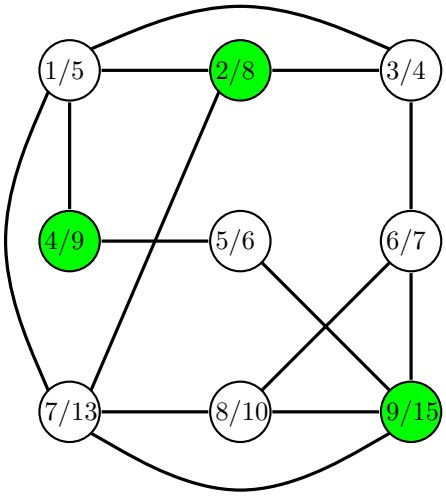


Figura 7.30: Grafo coloreado por el algoritmo goloso para $k = 1$

Si son considerados 2 colores, la solución encontrada será

$$V^*_{k=2}\{v_9, v_4, v_2, v_7, v_6, v_5\} \longrightarrow \sum p_i = 32 + 6 = 38$$

Cuyo gráfico se ilustra en la figura 7.31
En el grafo $\mathcal{G}$ de la figura 7.32, el algoritmo *greedy*ya no será la mejor opción. Se inicia y se detiene en el nodo 7.

Modelo de Algoritmo Evolutivo de Orden
Para tratar el problema de k-colorear grafos se aprovecha parte del conocimiento usado en la estrategia *greedy* sumado a las técnicas de la computación evolutiva como se describe en la tabla 7.5. En cierto modo, se está modelando un Algoritmo Evolutivo Híbrido.

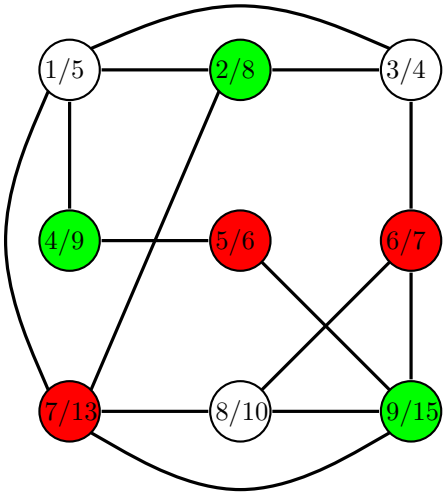


Figura 7.31: Grafo coloreado por el algoritmo goloso para $k = 2$

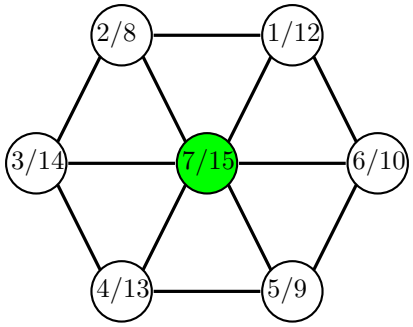


Figura 7.32: Grafo que no es adecuado para el algoritmo goloso

7.5.4 *Traveling Salesman Problem* – (TSP)

Este problema ubha sido uno de los más estudiados en el área de Investigación Operativa, por lo que merece una atención especial. Cuando la teoría de la Complejidad Algorítmica se desarrolló, el TSP fue uno de los primeros problemas en estudiarse, probando Karp en 1972 que pertenece a la clase de los problemas *NP-hard*.

Desde los métodos de Ramificación y Acotación hasta los basados en la Combinatoria Poliédrica, pasando por los procedimientos Metaheurísticos, todos han sido inicialmente probados en el TSP, convirtiéndose en una prueba obligada para validar cualquier técnica de resolución de problemas enteros o combinatorios.

El Problema del Viajante puede enunciarse del siguiente modo: «*Un viajante de comercio ha de visitar n ciudades, comenzando y finalizando en su pro-*

Cuadro 7.5: Algoritmo evolutivo versus estrategia *greedy*

Estrategia *greedy*	**Algoritmo Evolutivo**
Crear una lista de nodos en orden decreciente de peso	Crear una lista de nodos en el orden dado por el cromosoma
Ambos construyen la solución barriendo la lista y colocando colores válidos	

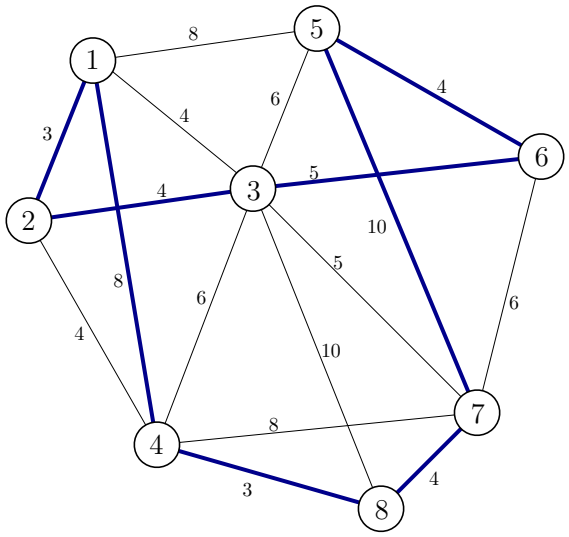


Figura 7.33: Ejemplo de ciclo hamiltoniano

pia ciudad. Conociendo el coste de ir de cada ciudad a otra, determinar el recorrido de coste mínimo.»

El problema se enuncia formalmente de la siguiente manera: Sea un grafo $\mathcal{G} = (V, A, C)$ donde V es el conjunto de vértices, A es el de aristas y $C = \{c_{ij}\}$ es la matriz de costos, donde c_{ij} es el costo o distancia de la arista (i, j).

- Un **camino** (o cadena) es una sucesión de aristas $(e_1, e_2, \ldots, e_k)$ en donde el vértice final de cada arista coincide con el inicial de la siguiente. También puede representarse por la sucesión de vértices utilizados.
- Un camino es **simple** o elemental si no utiliza el mismo vértice más de una vez.
- Un **ciclo** es un camino $(e_1, e_2, \ldots, e_k)$ en el que el vértice final de e_k coincide con el inicial de e_1.
- Un ciclo es **simple** si lo es el camino que lo define.
- Un **subtour** es un ciclo simple que no pasa por todos los vértices del grafo.
- Un **tour** o ciclo hamiltoniano es un ciclo simple que pasa por todos los vértices del grafo.

Así, el problema del viajante consiste en determinar un tour de costo mínimo, es decir encontrar el ciclo hamiltoniano con menor costo, como el mostrado para el caso de 8 vértices en la figura 7.33

Se puede considerar, sin pérdida de generalidad, que el grafo es completo, es decir, que para cada par de vértices existe una arista que los une. Notar que, de no ser así, siempre se puede añadir una arista ficticia entre dos vértices con el costo del camino más corto que los une.

Entre las aplicaciones más importantes del TSP se puede destacar:

- Fabricación de circuitos integrados
- Rutas de vehículos
- Recogida (robotizada) de material en almacenes
- Instalación de componentes en ordenadores
- subproblema en otras aplicaciones

La figura 7.34 ilustra la aplicación del problema de TSP para encontrar la

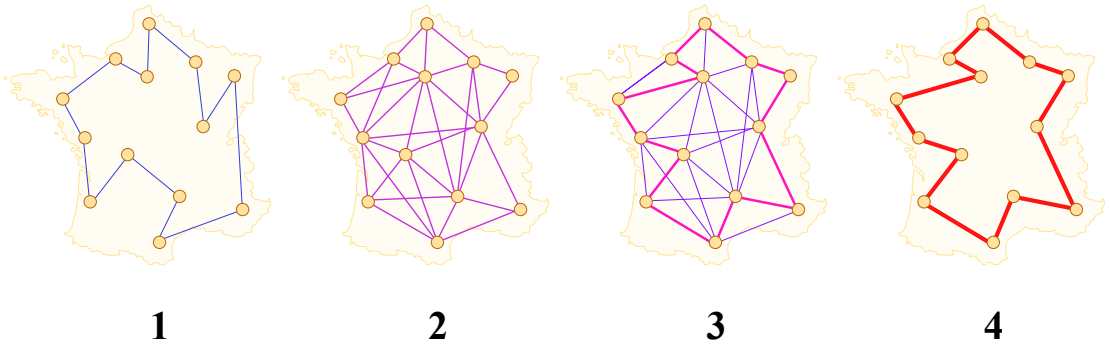


Figura 7.34: Ejemplo de caminos más cortos – *TSP*

mejor ruta entre algunas ciudades de Francia.

Representación e Inicialización
El objetivo de *Traveling Salesman Problem*, cuando no existen restricciones se resume a encontrar el *Ciclo Hamiltoniano* de menor costo en un grafo completo que se caracteriza como un problema de orden.
Sea $N = \{1, \ldots, n\}$ el conjunto de las n ciudades a ser visitadas.
Entonces, se define un cromosoma de tipo permutación $\phi = \{\phi(1), \ldots, \phi(n)\}$ que representa un orden de visita a las ciudades propuesto.

Función de Evaluación
La evaluación consiste en encontrar la distancia total recorrida por el viajero, entonces es necesario tener la información de distancias entre todas las ciudades a visitar dada por una matriz de distancias $\mathbf{D} = \{d_{ij}\}$, donde d_{ij} es la distancia entre las ciudades i y j..
Dado un determinado orden de visitas $\phi = \{\phi(1), \ldots, \phi(n)\}$, el cálculo de la distancia total sería efectuado como en la ecuación 7.57:

$$Z = \sum_{i=1}^{n-1} d_{\phi(i)\phi(i+1)} + d_{\phi(n)\phi(1)} \tag{7.57}$$

donde $d_{\phi(i)\phi(i+1)}$ es la distancia entre dos ciudades subsecuentes y $d_{\phi(n)\phi(1)}$ es la distancia entre la última ciudad visitada y la primera ciudad.
Si se excluye la ruta de retorno de la última a la primera ciudad, la evaluación sería:

$$Z = \sum_{i=1}^{n-1} d_{\phi(i)\phi(i+1)} \tag{7.58}$$

Operadores Genéticos
Para este problema se pueden aplicar los operadores genéticos de orden como
- Cruce Uniforme basado en orden
- Mutación de desorden
- Mutación basada en orden
- Mutación basada en posición

7.6 Otros algoritmos de búsqueda (EDA, *scatter search*)

Los algoritmos genéticos (GAs), que inicialmente fueron llamados "planes reproductivos" (Holland, 1975), son actualmente el representante más popular de la computación evolutiva. Este algoritmo definido por John Koza como "un algoritmo matemático altamente paralelo, que transforma un conjunto de objetos matemáticos individuales, cada uno con un valor de aptitud asociado; en poblaciones nuevas utilizando operaciones modeladas bajo el principio Darwiniano de la reproducción y supervivencia de los más aptos, naturalmente después de la ocurrencia de los operadores genéticos" (Koza, 1992); ha mostrado ser de gran utilidad en la solución de problemas prácticos reales.

La siguiente figura muestra la versión más simple del algoritmo genético, conocida como "Algoritmo Genético Simple" (SGA por sus siglas en inglés). En ella se muestra que para implementar un algoritmo genético en la solución de un problema, es necesario contar con un mecanismo que permita inicializar o generar una población P(t) de individuos que corresponden a posibles soluciones del problema. Esto implica que debemos de modelar las posibles soluciones al problema que estamos abordando de alguna manera que sea factible generarlas mediante algún mecanismo automático. Posteriormente es necesario contar con una herramienta que sea capaz de evaluar la calidad de cada uno de los individuos que constituyen a la población actual. La evaluación de individuos debe ser capaz de discernir entre aquellos que representan buenas soluciones al problema, de aquellos que no lo son; premiando a los que son mejores. A continuación, es necesario encontrar la forma de generar nuevas soluciones al problema a partir de las que ya se tienen y que por su calidad son soluciones prometedoras. Las soluciones (individuos) que se usarán para generar la siguiente generación P(t+1), deben ser seleccionadas cuidadosamente. Debido a que las nuevas soluciones deben producirse a partir de las que ya se tienen, es necesario establecer mecanismos de transformación de las soluciones seleccionadas; estos mecanismos de transformación son los operadores genéticos: cruzamiento y mutación. El proceso descrito debe ser repetido hasta que se alcance algún criterio de paro previamente establecido.

Algoritmo Genético Simple

1. Sea t=0 el contador de generaciones
2. Inicializa P(t)
3. Evalúa P(t)
4. Mientras no se cumpla un criterio de paro, hacer:
   a) Para i=1,...,N/2 hacer
      i. Selecciona 2 padres de P(t)
      ii. Aplica cruzamiento a los dos padres con probabilidad p_c
      iii. Muta la descendencia con probabilidad p_m
      iv. Introduce los dos nuevos individuos en P(t+1)
   v. Fin$_{para}$
   b) P(t)=P(t+1)
   c) Evalúa P(t)

d) Fin$_{mientras}$

A pesar de la simplicidad del algoritmo genético, se ha demostrado mediante

cadenas de Markov, que cuando los algoritmos evolutivos emplean elitismo, convergerán al óptimo global de ciertas funciones cuyo dominio puede ser un espacio arbitrario. Günter Rudolph en 1996, generalizó los desarrollos previos en teoría de la convergencia para espacios de búsqueda binarios y euclidianos a espacios de búsqueda generales (Rudolph, 1996).

Como consecuencia del enorme éxito que mostraron tener los algoritmos genéticos desde que Goldberg los popularizó con su publicación de 1989 (Goldberg, 1989); y teniendo en cuenta que la calibración de todos sus parámetros (probabilidad de cruce, probabilidad de mutación, tamaño de la población, etc.), apareció hace algunos años una nueva rama de la computación evolutiva conocida como "Algoritmos de Estimación de la distribución" (EDAs por sus siglas en inglés). Estos algoritmos representan una alternativa a los métodos heurísticos estocásticos existentes que no necesitan el ajuste de un número alto de parámetros y que además nos permiten explorar las interrelaciones entre las variables usadas en la representación de soluciones.

Los EDAS tienen un funcionamiento muy parecido al de su predecesor (algoritmo genético), en el sentido de que a partir de una población de soluciones a un problema y un grupo de alteraciones sobre éstas, se construye una nueva población evolucionada; sin embargo; en lugar de hacer uso de los operadores genéticos clásicos de cruzamiento y mutación, se utiliza la estimación de la distribución adyacente a un subconjunto de la población, para posteriormente utilizar esta estimación en la generación de una nueva población en base a su muestreo. Otra diferencia importante entre los EDAs y los AGs es que en éstos últimos las interrelaciones entre las variables se manejan de manera implícita, mientras que en los EDAs se expresan explícitamente mediante la distribución de probabilidad asociada a los individuos seleccionados de cada población.

A continuación se presenta el pseudocódigo general de esta estrategia.

Pseudocódigo de un EDA.

El algoritmo clásico de un EDA puede ser descrito mediante los siguientes 5 pasos:

1. Generación aleatoria de M individuos (Población inicial)
2. Repetir los pasos 3-5 para la generación l=1, 2, ... hasta que se alcance un criterio de parada
3. Seleccionar N<=M individuos desde Dl-1 de acuerdo a un método de selección
4. Estimar la distribución de probabilidad pl(x) del conjunto de individuos seleccionados
5. Muestrear M individuos (población nueva) desde pl(x)

Donde:

- M es el tamaño de la población
- Dl hace referencia a la l-ésima población

- pl(x) representa a la distribución de probabilidad de la l-ésima población

A fin de trabajar de acuerdo al pseudocódigo anterior, el primer paso es la generación de una población de posibles soluciones al problema que pretendemos resolver; el mecanismo clásico de generación de la población inicial en un EDA es aleatorio; sin embargo; también es factible que se implemente una generación conducida de las soluciones con las que iniciará la búsqueda el algoritmo. Este primer paso del EDA, refleja que para que el proceso evolutivo tenga lugar, se debe partir de un grupo inicial de posibles soluciones al problema que se plantea. Al igual que en el algoritmo genético, en el EDA las soluciones deben haber sido codificadas de manera que sea factible generarlas de manera automática.

Los siguientes pasos del algoritmo se deben repetir tantas veces como el número de generaciones que se vaya a manipular, o hasta que se haya alcanzado un criterio de paro específico. Un ejemplo de criterio de paro común para estos algoritmos, es cuando la mejor solución obtenida mediante el proceso evolutivo presenta un error menor a cierta tolerancia (para los casos en los que se conoce la solución deseada); o cuando las diferencias entre la mejor solución de una población y la siguiente se encuentran por debajo de un valor previamente establecido.

Los pasos 3 al 5 son los que comprenden el proceso de evolución de las poblaciones mediante la estimación de la distribución de cada generación. En primer término, se debe establecer un método de selección para obtener un subconjunto de la población de tamaño N donde N<=M (en cada paso del proceso evolutivo). Los métodos de selección pueden ser de dos tipos de acuerdo al mecanismo que la conduce: estocásticos y determinísticos.

La selección estocástica es aquella en la que la probabilidad de que un individuo sea escogido depende de una heurística; algunos ejemplos de ésta son: la selección por la regla de la ruleta y la selección por torneo. Por otro lado, la selección determinística es aquella que deja de lado al azar y cuya principal motivación es homogeneizar las características de la población para que no prevalezcan solamente las de los "super-individuos". En esta categoría se han explorado esquemas en los que por ejemplo, los individuos más aptos se cruzan con los menos aptos de la población (Pavez, 2009).

Una vez que se ha seleccionado el conjunto de individuos de la l-ésima población, se realiza la estimación de su distribución de probabilidad. De acuerdo al nivel de interacción entre las variables que el modelo probabilístico toma en consideración, los EDAs se clasifican en tres tipos: aquellos en los que se asume independencia entre las variables que forman el modelo, "univariados"; aquellos en los que se asume que las interacciones se dan entre pares de variables, "bivariados" y aquellos en los que la distribución de probabilidad modela las interacciones entre más de dos variables, "multivariados"(Talbi,

2009). Otra clasificación de los EDAs, los agrupa en función al tipo de variables que involucran en continuos, discretos y mixtos.

Una vez que se ha realizado la estimación de la distribución del grupo de individuos de la l-ésima generación, se procede a la construcción de una nueva población (l+1), obtenida mediante el muestreo de la función de probabilidad obtenida pl(x).

El primer EDA desarrollado y el más utilizado hasta el momento es el que supone una distribución marginal univariada, conocido por sus siglas en inglés como UMDA (Univariate Marginal Distribution Algorithm) (Mhlenbein and Paa[U+F062], 1996). El pseudocódigo de este algoritmo suponiendo una codificación binaria de las soluciones, se muestra a continuación.

7.6.1 Pseudocódigo del algoritmo UMDA:

1. Generación aleatoria de M individuos (Población inicial)
2. Repetir los pasos 3 al 5 para la generación l=1, 2, ... hasta que se alcance un criterio de parada
3. Seleccionar N<=M individuos desde Dl-1 de acuerdo a un método de selección
4. La probabilidad de que el i-ésimo elemento de una solución tenga el valor xi=1 en el conjunto de individuos seleccionados es: $p_i(x_i) = \frac{\sum_{j=1}^{N} \delta_j(X_i = x_i / D_{l-1}^{Seleccionados})}{N}$, donde $\delta_j(X_i = x_i / D_{l-1}^{Seleccionados}) = 1$ si el i-ésimo elemento del j-ésimo individuo seleccionado es x_i, y 0 en caso contrario.
5. Generar M individuos (población nueva) tal que para cada elemento i, la probabilidad de que tome valor x_i es pl(x_i).

7.6.2 Ejemplo del uso del UMDA en el problema del máximo número de unos.

La forma más simple de entender el funcionamiento de estos algoritmos es mediante un ejemplo, de manera que a continuación se describe el uso del UMDA en la solución de un problema de optimización académico conocido como "The OneMax problem" o problema del máximo número de unos.

Supongamos que nos interesa maximizar el número de unos "1" de una función que se representa como cadenas binarias de longitud 4. Es decir, nos interesa maximizar $f(x) = \sum_{i=1}^{4} x_i$ donde x_i={0,1}.

El primer paso del algoritmo consiste en generar una población de M posibles soluciones a este problema. Debido a que cada posible solución se representa como una cadena de 4 caracteres, en la que cada elemento puede tomar los valores de 0 o 1 exclusivamente; una forma de generar automáticamente estas soluciones puede ser mediante la distribución de probabilidad siguiente:

$p_0(x) = \prod_{i=1}^{4} p_0(x_i) = p_0(x_1, x_2, x_3, x_4)$

Donde la probabilidad de que cada elemento xi sea igual a 1 es $p_0(x_i)$.

A fin de que el ejemplo sea breve nos limitaremos a trabajar con poblaciones de M=8 individuos (o posibles soluciones al problema). Consideremos que inicialmente se supone que para cada variable la probabilidad de que sea $x_i = 1$ es 0.5; se generará un número aleatorio "r" comprendido entre (0,1) y si el número generado es menor que 0.5, la i-esima variable de la solución x_i será 0; si el número generado fuera mayor o igual que 0.5, el valor de la i-ésima variable deberá ser 1.

Mediante el uso de este mecanismo se genera la población D0 de M=8 individuos que se muestra en la tabla 1.

Tabla 2. Población inicial D_0

i	X1	X2	X3	X4	F(X)
1	1	0	0	1	2
2	0	0	1	0	1
3	1	1	0	1	3
4	0	1	1	1	3
5	1	1	0	0	2
6	1	0	1	1	3
7	1	0	0	0	1
8	1	1	1	0	3

Como se muestra en la tabla anterior, una vez que se ha generado la población con la que se trabajará, es necesario que cada uno de sus individuos sea evaluado para continuar con el proceso de selección.

Enseguida deberemos seleccionar de la población inicial D0, un grupo de individuos N<=M de acuerdo a un criterio de selección. Para el ejemplo, usaremos N=4 individuos. Los 4 individuos seleccionados para nuestro ejemplo serán aquellos con una evaluación más alta (individuos 3, 4, 6 y 8). Este grupo de elementos D0 sel será el insumo para estimar los parámetros de la distribución de probabilidad que conducirá la generación de la nueva población de individuos. La tabla 2 muestra el grupo de individuos seleccionados y la probabilidad de que el xi sea 1.

En nuestro ejemplo se esta asumiendo que las variables son independientes, por lo que necesitamos un parámetro para cada una de ellas. Los parámetros son calculados en base a su frecuencia.

Tabla 3. Individuos seleccionados de D0

Mediante el muestreo de la distribución de probabilidad obtenida en el paso anterior p1(x) (ver ecuación 2), se procede a la generación de la siguiente población D1.

$p_1(x) = \prod_{i=1}^{4} p_1(x_i|{D_0}^{Sel}) = p_1(x_1, x_2, x_3, x_4)$

i	X1	X2	X3	X4	F(X)
3	1	1	0	1	3
4	0	1	1	1	3
6	1	0	1	1	3
8	1	1	1	0	3
P(x)	0.75	0.75	0.75	0.75	

La tabla 3 muestra la población D1, obtenida a partir de p1(x).

Tabla 4. Población D1

i	X1	X2	X3	X4	F(X)
1	1	0	1	1	3
2	1	1	1	0	3
3	1	1	1	1	4
4	0	1	1	1	3
5	1	1	1	1	4
6	1	0	1	1	3
7	1	1	1	1	4
8	1	1	1	0	3

A partir de esta nueva población, se deberá de seleccionar nuevamente a los 4 mejores individuos D1sel (ver tabla 4) para recalcular el vector de probabilidad de cada variable p2(x).

Tabla 5. Individuos seleccionados de D1

i	X1	X2	X3	X4	F(X)
1	1	0	1	1	3
3	1	1	1	1	4
5	1	1	1	1	4
7	1	1	1	1	4
P(x)	1.0	0.75	1.0	1.0	

Con ese nuevo vector p(x) deberemos repetir el muestreo de la siguiente población y seguir repitiendo el proceso hasta que se cumpla algún criterio de paro.

Como se aprecia entre las poblaciones D_0 y D_1, las soluciones de cada población comienzan a ser cada vez mejores. En nuestro ejemplo ya aparece la solución óptima f(x)=4 en la población D_1 y se espera que al seguir iterando, nuestro vector de probabilidad vaya incrementando sus valores hasta llegar al 100 % de probabilidad de que cada variable tenga valor de uno.

7.6.3 Otros algoritmos de estimación de la distribución.

Como se mencionó anteriormente, los diferentes algoritmos propuestos en el campo de los EDAS, pueden ser agrupados de acuerdo a la complejidad del modelo empleado. El UMDA corresponde al modelo más simple debido a que supone que no existe interacción entre las variables. La desventaja de este modelo es que en la práctica es frecuente que las variables que describen un fenómeno posean dependencias unas con otras.

Otro algoritmo univariado (que supone la ausencia de dependencia entre las variables) muy popular es el conocido como PBIL por sus siglas en inglés "Population-Based Incremental Learning" (Baluja, 1994). El mecanismo de funcionamiento de este algoritmo consiste en el refinamiento del modelo en cada generación del proceso evolutivo; en lugar de la estimación de un nuevo modelo por cada generación como se hace en el UMDA. Es decir, que en el PBIL, para cada paso de refinamiento del modelo, se consideran todos los pasos anteriores.

En la medida que el modelo empleado para la estimación de la distribución se va complicando, las dependencias entre las variables se van capturando de mejor forma, lo que ha llevado a los investigadores a proponer algoritmos como el "MIMIC" (por sus siglas en inglés Mutual Information Maximizing Input Clustering Algorithm) (De Bonet et. al, 1996). Otro algoritmo que supone dependencias entre pares de variables y que a diferencia del MIMIC utiliza árboles en lugar de cadenas, es el COMIT (siglas en inglés de Combining Optimizers with Mutual Information Trees) (Baluja and Dabies, 1997). El COMIT es un algoritmo que utiliza el algoritmo "Maximum Weight Spanning Tree (MWST)" propuesto por Chow y Liu (Chow y Liu, 1968). Este algoritmo realiza la estimación de la distribución de probabilidad en cada generación mediante una red Bayesiana con estructura de árbol.

Otro representante de los algoritmos que consideran que las dependencias entre variables se presentan por pares es el algoritmo BMDA, en el que Pelikan y Mühlenbein proponen una factorización de la distribución de probabilidad conjunta que sólo necesita estadísticos de segundo orden. Este algoritmo está basado en la construcción de un grafo de dependencias dirigido acíclico que no necesariamente es conexo (Pelikan and Mühlenbein, 1999).

Finalmente, se encuentran los algoritmos que suponen un modelo de dependencias múltiples entre variables. Estos algoritmos proponen la construcción de un modelo gráfico probabilístico en el que no hay restricción en el número de padres que puede tener cada variable; de manera que la estructura adquiere una complejidad mucho mayor que con los modelos anteriores. Como ejemplo de algoritmos de esta categoría se encuentran el EcGA (Extended compact Genetic Algorithm) presentado por Harik (Harik et al., 1998); el algoritmo FDA o algoritmo de distribución factorizada (Factorized Distribution Algorithm) propuesto por Mühlenbein y sus colaboradores (Mühlenbein et al., 1999) y entre otros, el EBNA (Estimation of Bayesian Networks Algorithm o algoritmo de estimación de redes Bayesianas), presentado por Etxeberria y Larrañaga (Etxeberria y Larrañaga, 1999).

Muchos de los algoritmos mencionados tienen una versión para entornos continuos además de otro grupo de algoritmos creados especialmente para trabajar con problemas de naturaleza continua, que como en el caso de entornos discretos, siguen la misma clasificación.

En la actualidad la búsqueda de mejoras de este tipo de algoritmos así como su aplicación a problemas reales es materia de investigación.

7.7 ACTIVIDADES DE APRENDIZAJE

Responda a los siguientes cuestionamientos:

1. ¿Cuál es el significado de las siglas EDA?
2. Explique el significado de las siglas UMDA
3. Explique las principales diferencias entre el algoritmo genético y el algoritmo de estimación de la distribución.
4. Mencione el nombre de dos algoritmos de estimación de la distribución que trabajan bajo el supuesto de que las variables del modelo son independientes.
5. Mencione el nombre de un algoritmo que

7.8 LECTURAS ADICIONALES.

Larrañaga, P. and Lozano, J. "Estimation of Distribution Algorithms, A New Tool for Evolutionary Computation. Genetic Algorithms and Evolutionary Computation". Kluwer Academic Publishers. 2002. (Este es un libro de referencia sobre los algoritmos de estimación de la distribución, pues comprende una colección de artículos detallados y explicados ampliamente sobre el tema).
Santana, R., Larrañaga, P., y Lozano, J. A. "Research topics in discrete estimation of distribution algorithms based on factorizations". Memetic Computing, 1(1):35-54. 2009. (Este trabajo muestra una recopilación del trabajo realizado con la intención de mejorar el campo de los EDAs).

7.9 REFERENCIAS

Baluja. S. "Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning". Technical Report TR CMU-CS 94-163, Carnegie Mellon University, 1994.

Baluja. S. and Davies, S. "Combining Multiple Optimization Runs with Optimal Dependency Trees". Technical Report TR CMU-CS-97-157, Carnegie Mellon University, 1997.

Chow. C and Liu. C. "Aproximating Discrete Probability Distributions with Dependence Trees", IEEE Transactions on Information Theory, Vol. IT 14. No. 3. Pp. 462-467. 1968.

De Bonet, J. S., Isbell, C. L. and Viola, P. "MIMIC: Finding Optima by

Estimating Probability Densities. In Proceeding of Neural Information Processing Systems". Pp. 424-430, 1996.

Etxeberria, R. y Larrañaga, P. Global optimization with Bayesian networks. En Special Session on Distributions and Evolutionary Optimization, pp 332{339, La Habana, Cuba. II Symposium on Artificial Intelligence, CIMAF99.

Goldberg, D. E., "Genetic Algorithms in Search Optimization & Machine Learning". Addison-Wesley, 1989.

Harik, G., Lobo, F. G., y Golberg, D. E. (1998). The compact genetic algorithm. En Proceedings of the IEEE Conference on Evolutionary Computation, pp 523- 528, Piscataway, NJ.

Holland, J. "Adaptation in Natural and Artificial Systems". University of Michigan Press. Ann Arbor, MI, 1975; MIT Press, Cambridge, MA 1992.

Koza, J. R. "Genetic Programming. On the Programming of Computers by Means of Natural Selection". MIT Press. Cambridge, Massachussetts, 1992.

Lozano J.A., Larrañaga P., Inza I., Bengoetxea E., "Towards a New Evolutionary Computation. Advances in the Estimation of Distribution Algorithms". Springer-Verlag Berlin Heidelberg New York., 2006.

Larrañaga. P., Lozano. J. A. y Mühlenbein. H. "Algoritmos de Estimación de Distribuciones en Problemas de Optimización Combinatoria" Revista Iberoamericana de Inteligencia Artificial. Asociación Española para la Inteligencia Artificial. Año/Vol. 7, Num. 019. 2003.

Mhlenbein, H., and Paa[U+F062], G. "From Recombination of Genes to the Estimation of Distributions I. Binary Parameters", in H.M.Voigt, et al., eds., Lecture Notes in Computer Science 1411: Parallel Problem Solving from Nature - PPSN IV, pp. 178-187. 1996.

Mühlenbein, H. y Mahning, T. (1999). FDA - a scalable evolutionary algorithm for the optimization of additively decomposed functions. Evolutionary Computation, 7(4):353{376.

Pavez, B., Soto, J., Urrutia, C., y Curilem, M., "Selección Determinística y Cruce Anular en Algoritmos Genéticos: Aplicación a la Planificación de Unidades Térmicas de Generación", Ingeniare. Revista chilena de ingeniería [en línea] 2009, vol. 17 no. 2. Disponible en Internet: URL: http://www.redalyc.org/articulo.oa?id=77211359006 . Visitado en Enero de 2014.

Pelikan, M. and Mühlenbein, H. "The Bivariate Marginal Distribution Algorithm". Advances in Soft Computing-Engineering Design and Manufacturing. Pp. 521-535, 1999.

Rudolph, G. “Convergence of Evolutionary Algorithms in General Search Spaces”, In Proceedings of the Third IEEE Conference on Evolutionary Computation. 1996.

Talbi, E. G., “Metaheuristics: From Design to Implementation”. Editorial Wiley & Sons, Publication. 2009.

Back, T., Fogel, D. B., and Michalewicz, Z., editors (1997).

Handbook of Evolutionary Computation. IOP Publishing Ltd., Bristol, UK, UK, 1st edition.

Bellman, R. E. (1957). Dynamic Programming. Princeton Univer-sity Press, Princeton NJ.

Deb, K. (2000). Encoding and decoding functions. Evolutionary Com-putation 2, Advanced Algorithms and Operators, pages 4–11.

Eiben, A. E. and Smith, J. E. (2003). Introduction to evo-lutionary computing. Springer-Verlag Berlin Heidelberg, Berlin.

Fogel, D. B., editor (1998). Evolutionary Computation. The Fossil Record. Selected Readings on the History of Evolutionary Algorithms. The Institute of Electrical and Electronic Engineers, New York.

Fogel, L. J. (1996). Artificial Intelligence through Simulated Evolu-tion. John Wiley, New York.

Fogel, L. J., Owens, A. J., and Walsh, M. J. (1965). Artificial Intelligence through a Simulation of Evolution. In Maxfield , M., Callahan, A., and Fogel, L. J., editors, Biophysics and Cybernetic Systems: Proceedings of the Second Cybernetic Sciences Symposium, pages 131–155. Spartan Books, Washington, D.C.

Glover, F. and Laguna, M. (1998). Tabu Search. Kluwer Academic Publishers,Norwell Massachusetts.

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimiza-tion and Machine Learning. Addison-Wesley Publishing Co., Reading, Mas-sachusetts.

Goldberg, D. E. (1990). Real-Coded Genetic Algorithms, Virtual Alphabets and Blocking. University of Illinois at Urbana-Champaing, Techni-cal Report No. 90001.

Haykin, S. (1998). Neural Networks: A Comprehensive Foundation. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition.

Holland, J. H. (1962a). Concerning efficie nt adaptive systems. In Yovits, M. C., Jacobi, G. T., and Goldstein, G. D., editors, Self-Organizing Systems—

1962 , pages 215–230. Spartan Books, Washington, D.C.

Holland, J. H. (1962b). Outline for a logical theory of adaptive systems. Journal of the Association for Computing Machinery, 9:297–314.

Holland, J. H. (1975). Adaptation in Natural and Artificial Systems . University of Michigan Press, Ann Arbor, Michigan, first edi tion.

]Kirkpatrick, S., Gelatt, J. C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. Science, 220:671–680.

Kuhn, H. W. and Tucker, A. W. (1951). Nonlinear pro-gramming. In Neyman, J., editor, Proceedings of 2nd Berkeley Symposium, pages 481–492, Berkeley, CA. Berkeley: University of Calif ornia Press.

Mathews, G. B. (1897). On the partition of numbers. In Proceed-ings of the London Mathematical Society, volume XXVIII, pages 486–490.

Michalewicz, Z. (1996). Genetic Algorithms + Data Structures = Evolution Programs. Springer-Verlag, New York, third edition.

Neumaier, A. (2006). Global optimization and constraint satisfac-tion. Proceedings of CICOLAC workshop (of the search project Global Opti-mization, Integrating Convexity, Optimization, Logic Programming and Com-putatiobnal Algebraic Geometry.

Pauling, L. (1960). The Nature of the Chemical Bond. Cornell Univ. Press, Mineola, NY.

Rao, S. S. (1996). Engineering Optimization. Theory and Practice. John Wiley & Sons, third edition.

Rechenberg, I. (1973). Evolutionsstrategie: Optimierung tech-nischer Systeme nach Prinzipien der biologischen Evolution. Frommann– Holzboog, Stuttgart, Alemania.

Reeves, C. B. (1993). Modern Heuristic Techniques for Combinato-rial Problems. John Wiley & Sons, Great Britain.

Rudolph, G. (1994). Convergence analysis of canonical genetic algorithms. IEEE Transactions on Neural Networks, 5:96–101.

Russell, S. J. and Norvig, P. (2002). Artificial Intelli-gence. A Modern Approach. Prentice Hall„ Upper Saddle River, New Jersey, second edition.

Schiff, J. L. (2007). Cellular Automata: A Discrete View of the World. Wiley-Interscience.

Schwefel, H.-P. (1977). Numerische Optimierung von Computer-Modellen

mittels der Evolutionsstrategie. Birkhauser,¨ Basel, Alemania.

Spencer, H. (1960). The Principles of Biology, volume 1. London and Edinburgh: Williams and Norgate, first edition.

van der Hauw, J. (1996). Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems. Master's the-sis, Computer Science Department of Leiden University, Leiden, Netherlands.

8 — Algoritmos Bioinspirados

Francisco Javier Ornelas Zapata, Julio Cesar Ponce Gallegos
Universidad Autonoma de Aguascalientes, Mexico

8.1 INTRODUCCIÓN.

La invención de la Computadora dotó a los seres humanos de una herramienta poderosa para auxiliarle en la resolución de problemas.

Sin embargo, los algoritmos tradicionales no siempre han servido o han sido óptimos para la resolución de cierto tipo de problemas donde el espacio de búsqueda o el número de soluciones no es único.

Por tal razón se comenzaron a abstraer fenómenos naturales y a modelarse primero de forma matemática y después computacionalmente en un intento de dar respuesta a problemas que de otra forma es muy difícil resolver.

A este tipo Técnicas para la solución de problemas se les dio el nombre de "Técnicas Bioinspiradas" debido a que intentaban copiar modelos y/o fenómenos naturales.

El primer trabajo data de la década de 1960 cuando el científico Frank Rosemblatt desarrolla, en la Universidad de Cornell, un modelo de la mente humana a través de una red neuronal y produce un primer resultado al cual llama perceptrón. Este trabajo constituye la base de las redes neuronales de hoy en día.

Las Redes Neuronales son un paradigma de aprendizaje y procesamiento automático inspirado en la forma en que funciona el cerebro para realizar las tareas de pensar y tomar decisiones (sistema nervioso). El cerebro se trata de un sistema de interconexión de neuronas en una red que colabora para producir un estímulo de salida.

Por otro lado tenemos a los Algoritmos Genéticos, los cuales se basan en la evolución biológica y su base genético-molecular. Esta técnica tuvo sus inicios en 1970 con el científico John Holland. Estos algoritmos hacen evolucionar una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución biológica (mutaciones y recombinaciones genéticas), así como también a una Selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que

sobreviven, y cuáles los menos, que son descartados, ya que se basan en la teoría de la evolución de la especies propuesta por Charles Darwin.

Sistema Inmune Artificial. Los sistemas inmunes artificiales son sistemas computacionales adaptativos cuya inspiración está basada en los diferentes mecanismos del sistema inmune biológico, especialmente de los mamíferos, con la finalidad de solucionar problemas de ingeniería complejos, en los que los investigadores del área han mostrado resultados exitosos (Cruz Cortés, 2004).

Existen también los algoritmos clasificados como Swarm Intelligence, que son técnicas metaheurísticas bioinspiradas, que en especial modelan el comportamiento de agentes que trabajan colaborativamente para resolver situaciones que se presentan en la naturaleza (inteligencia colectiva), basados en los modelos naturales de comportamiento de animales que viven en colonias, parvadas, enjambres o bancos como colonias de hormigas, termitas, abejas, patos, peces, entre otros.

En este capítulo trataremos un poco más a fondo, algunos de estos algoritmos como son las Colonias de Hormigas y los algoritmos de Optimización de Cúmulo de Partículas (PSO, Particle Swarm Optimization), que se basan en el comportamiento de las parvadas de pájaros.

8.2 Swarm Intelligence

Swarm Intelligence o Inteligencia Colectiva puede ser visualizado como una metáfora de la naturaleza que permite resolver problemas distribuidamente, basándose en la vida social de cierto tipo de animales:

- Termitas y avispas en la construcción del nido.
- Abejas en la construcción del panal.
- Hormigas en la búsqueda y obtención de alimento.
- Algunos pájaros en sus movimientos en bandada.
- Cardumen de peces en caza.

Swarm Intelligence son técnicas metaheurísticas bioinspiradas, que en especial modelan el comportamiento de la inteligencia colectiva. Algunas de las técnicas que se clasifican dentro de este tipo de algoritmos son:

1. Algoritmos de Cúmulos de Partículas (Particle Swarm Optimization, PSO).
2. Algoritmos de Colonias de Hormigas (Ant Colony Optimization, ACO).
3. Algoritmos Basados en la Polinización de Abejas Artificiales (Artificial Bee Colony, ABC).
4. Algoritmo de Cúmulos Basado en Espacios Geométricos (Geometric PSO, GPSO).

Swarm Intelligence se desarrolló para intentar resolver problemas donde el espacio de búsqueda es muy grande y además, pueden existir varias soluciones al problema planteado. Generalmente, se aplica a problemas conocidos como NP-Completos.

En este tipo de problemas generalmente se busca obtener una de las "mejores soluciones" de todas las posibles soluciones que puedan existir.

Se busca una optimización a través de la maximización o minimización de un objetivo (conocido como "función objetivo") el cual puede ser modelado desde una simple ecuación, hasta un complejo sistema de reglas dependiendo del dominio de aplicación.

En nuestros días las técnicas de Inteligencia Colectiva (Swarm Intelligence) tienen una variedad de aplicaciones en distintas áreas y problemas específicos, como por ejemplo:

Minería de Datos (Abraham, Grosan, & Ramos, 2006), (Parpinelli, Lopes, & Freitas, 2002); Enrutamiento de Vehículos con Ventanas de Tiempo (Gambardella, Taillard, & Agazzi, 1999); Problema del Agente Viajero (Dorigo & Gambardella, 1997); Enrutamiento en redes de comunicación (Kassabalidis, El-Sharkawi, Marks, Arabshahi, & Gray, 2001); Entrenamiento de Redes Neuronales para identificar Impedancia (Peng, Venayagamoorthy, & Corzine, 2007); Control de robots para la eliminación de aceite contaminante en el mar (Fritsch, Wegener, & Schraft, 2007); entre otros.

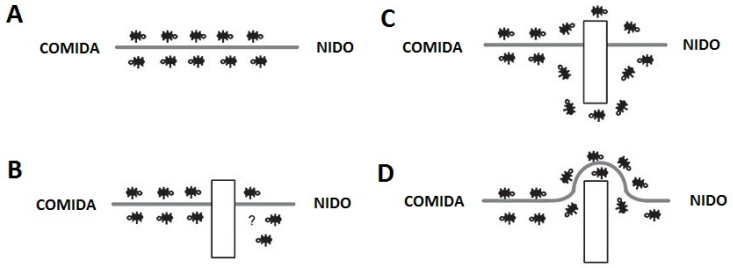
8.2.1 Optimización de Colonias de Hormigas (Ant Colony Optimization, ACO)

Los algoritmos de colonias de hormigas son una metaheurística bioinspirada en el comportamiento estructurado de las colonias de hormigas naturales, donde individuos muy simples de una colonia (como aprecia en la figura 1A) se comunican entre sí por medio de una sustancia química denominada feromona, estableciendo el camino más adecuado entre el hormiguero y una fuente de alimentación lo que es un comportamiento es muy interesante (Dorigo, Maniezzo, & Colorni, 1996).

Los algoritmos de optimización con colonias de hormigas se han clasificado también como algoritmos constructivos, por la forma en que trabaja por dentro el algoritmo, ya que cada hormiga construye una solución al problema, recorriendo un grafo de construcción al ir de un estado a otro dentro del espacio de estados. El espacio de estados puede ser representado por un grafo en el que un vértice representa un estado y cada arista del grafo representa los posibles pasos o acciones que la hormiga puede realizar. Normalmente cada arista tiene asociada información acerca del problema que guía el movimiento de la hormiga. La información asociada es el rastro de feromonas y la visibilidad la distancia. En estos algoritmos, las hormigas escogen a donde ir de una manera probabilística (Dorigo, Maniezzo, & Colorni, 1996). En las figuras 1C y 1D se muestra como es diferente la concentración de feromonas

entre los dos caminos.

Figura 8.1: Comportamiento de las hormigas para generar caminos entre su nido y la fuente de alimento



El algoritmo de colonia de hormigas ha sido aplicado, con el paso del tiempo para resolver diferentes problemas, obteniendo cada vez mejores resultados debido a los cambios en el poder computacional. Algunos de los problemas solucionados con ACO, sus desarrolladores, así como sus principales características son:

El AS que fue el primer algoritmo de colonia de hormigas desarrollado por Marco Dorigo.

El algoritmo Ant- (Gambardella & Dorigo, 1995), es un sistema híbrido del algoritmo AS con aprendizaje-Q (Q-learning), un modelo de aprendizaje por refuerzo muy conocido.

Rank-based (Bullnheimer , Hartl, & Strauss, 1997), es una variación del AS en el cual, la actualización de las feromonas se realiza depositando una cantidad de feromona entre ciudades por cada hormiga y se suma la cantidad de esta, de los caminos por los que pasaron las n mejores hormigas, además se realiza un incremento adicional si la mejor hormiga viajó por esa ruta.

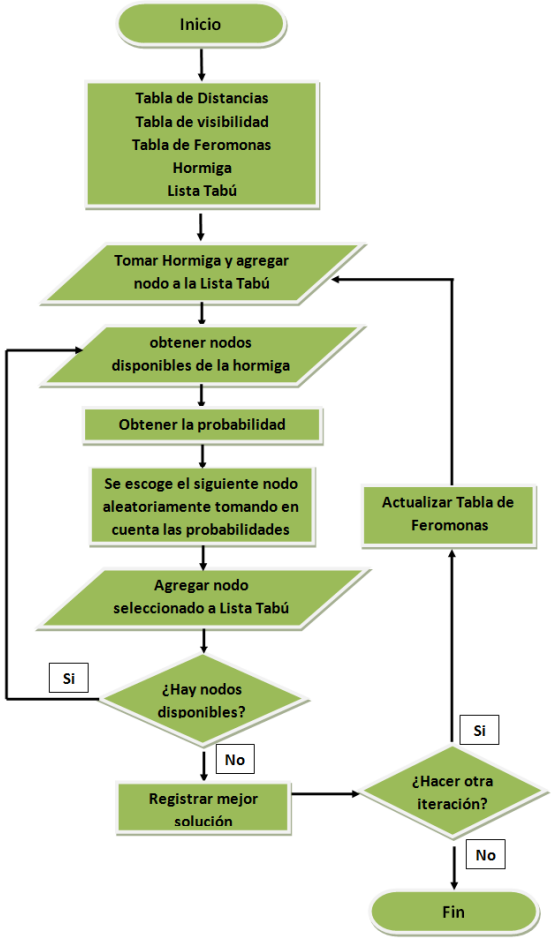
Ant Colony System, ACS (Dorigo & Gambardella, 1997), es un algoritmo que es una extensión del Ant-Q.

MAX-MIN Ant System (Stützle & Hoos, 1998), este algoritmo se caracteriza por solo actualizar los rastros de feromona de la mejor hormiga de cada ciclo y se establecen valores máximos y mínimos como límites en la acumulación de feromona.

ASGA (White, Pagurek, & Oppacher, 1998), es un algoritmo híbrido entre AS y un GA en el cual se utiliza el AG para adaptar los parámetros del AS, de tal forma que la búsqueda se va adaptando en base a los resultados.

ACSGA-TSP (Pilat & White, 2002)). Es un algoritmo el cual incorpora una hibridación entre un algoritmo genético y el algoritmo ACS-TSP.

Figura 8.2: Algoritmo para la implementación del método de búsqueda de Colonia de Hormigas



Algoritmo General

```
Procedimiento OCH ()
        Inicializacion_de_parametros
        establecer_feromona_inicial();
        mientras (criterio de terminacion no
            este satisfecho)
                Generar_las_hormigas();
                para (cada hormiga)
                        Generar_una_solucion();
                Fin para
                si (
                    actualizacion_feromona_no_es_en_linea
                    )
                        para cada arco visitado
                            actualizar_feromona
                        ;
```

```
                destruir_hormigas_o_recursos();
        fin mientras
fin Procedimiento
```

Procedimiento de inicialización de las feromonas

```
Procedimiento establecer_feromona_inicial ()
        Para cada arco del grafo existente
           colocar un concentracon inicial (C)
           de feromona
fin Procedimiento
```

Procedimiento donde se colocan las hormigas en su posición inicial

```
Procedimiento Generacion_de_Hormigas()
        repetir desde k=1 hasta m (
           numero_hormigas)
                crear hormiga y colocar en su
                   nodo inicial (actualizar
                   lista tabu);
fin repetir fin Procedimiento
```

Procedimiento para generar una solución

```
Procedimiento Generar_una_solucion()
        mientras (el criterio para obtener una
           solucion no este satisfecho)
                mover_hormiga();
        fin mientras
fin procedimiento
```

Procedimiento para mover una hormiga de un lugar a otro

```
Procedimiento mover_hormiga()
        para (todo el vecindario factible)
                calcular_probabilidades_de_movimento
                    con la formula
        fin para
        estado_seleccionado:=
           seleccionar_el_movimiento();
        llevar_hormiga_a_ciudad(
           ciudad_seleccionada);
        si (
           actualizacion_feromona_en_linea_paso_a_paso
           )
```

```
                depositar_feromona_en_el_arco_vistado()
                    ;
        fin Procedimiento
```

La probabilidad de que una hormiga seleccione uno de los posibles caminos está dada por la siguiente formula:

$$p(v_i) = \frac{|\tau_{ij}|^{\alpha}|\eta_{ij}|^{\beta}}{\sum v\ j\ candidatos\ |\tau_{ij}|^{\alpha}|\eta_{ij}|^{\beta}}$$

Una vez que cada hormiga ha generado una solución esta se evalúa y puede depositar una cantidad de feromona que es función de la calidad de su solución y está dada por la fórmula:

$$\tau_{ij}(t+n) = \rho\tau_{ij}(t) + \Delta\tau_{ij}$$

Tabla de Visibilidad

La visibilidad está dada por información específica del problema y es representada por la letra n en el caso específico del problema del agente viajero al ser un problema de minimización está dado por el inverso de la distancia n = 1/d. La visibilidad es representada a través de una matriz.

Tabla de Feromonas

La feromona es la representación de la concentración depositada por las hormigas a través del tiempo, algunas pueden ir disminuyendo debido a la evaporación.

Ejemplo simple

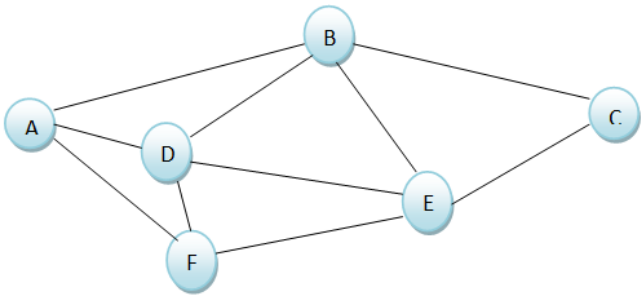
En este ejemplo solo se hará una pequeña demostración de cómo se realiza el procedimiento para dar solución a un problema por medio de el algoritmo de colonia de hormigas, no se realizara el ejemplo completo por lo que el fin de esto es mostrar los pasos necesarios para que el alumno sepa manejarlo.

Se cuenta con el siguiente grafo.

Primero se obtiene la instancia del problema (matriz de distancia).

Se inicializan la feromonas (matriz de feromonas).

Figura 8.3: Muestra el grafo que se genera basado en el problema.



	A	B	C	D	E	F
A	0	9	0	4	0	5
B	9	0	8	6	6	0
C	0	8	0	0	4	0
D	4	6	0	0	6	2
E	0	6	4	6	0	5
F	5	0	0	2	5	0

	A	B	C	D	E	F
A	0	0.08	0	0.08	0	0.08
B	0.08	0	0.08	0.08	0.08	0
C	0	0.08	0	0	0.08	0
D	0.08	0.08	0	0	0.08	0.08
E	0	0.08	0.08	0.08	0	0.08
F	0.08	0	0	0.08	0.08	0

Se obtiene la matriz de visibilidad (1 / distancia).

Se colocan las hormigas aleatoriamente (lista Tabú).

Paso 1: Se toma la hormiga K que en este caso sería la hormiga 1 en el nodo B.

Ahora se obtienen los nodos permitidos a los cuales se puede desplazar.

Paso 2: Se obtienen las probabilidades utilizando la siguiente fórmula:

	A	B	C	D	E	F
A	0	1/9	0	1/4	0	1/5
B	1/9	0	1/8	1/6	1/6	0
C	0	1/8	0	0	1/4	0
D	1/4	1/6	0	0	1/6	1/2
E	0	1/6	1/4	1/6	0	1/5
F	1/5	0	0	1/2	1/5	0

	1	2	3	4	5	6
Hormiga 1	B					
Hormiga 2	D					
Hormiga 3	A					

Figura 8.4: Muestra la inicialización del recorrido posicionando a una hormiga en el nodo al azar.

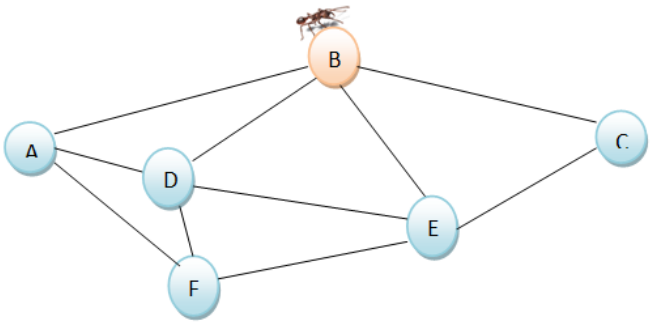
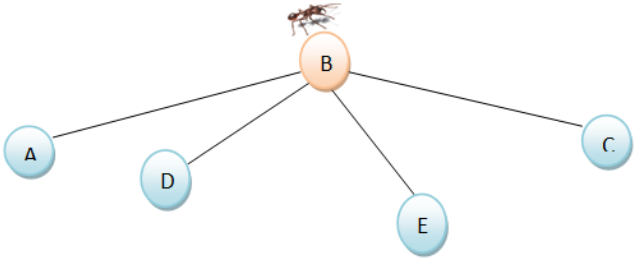


Figura 8.5: Muestra los nodos validos a partir del nodo actual.



$$p(v_i) = \frac{|\tau_{ij}|^{\alpha}|\eta_{ij}|^{\beta}}{\sum v\ j\ candidatos\ |\tau_{ij}|^{\alpha}|\eta_{ij}|^{\beta}}$$

Dónde:

k = Número de hormigas
i = Nodos de origen
j = Nodo a desplazarse
α = 1, importancia de la feromona
β = 5, importancia de la visibilidad

τ = valor tomado de la matriz de feromonas
η = valor tomado de la matriz de visibilidad 0.000010288

$$p^1_{BA}(t) = \frac{[0.08]^1[1/9]^5}{[0.08]^1[1/9]^5 + [0.08]^1[1/8]^5 + [0.08]^1[1/6]^5 + [0.08]^1[1/6]^5} = \frac{0.000001354}{0.000024371} = 0.05$$

$$p^1_{BC}(t) = \frac{[0.08]^1[1/8]^5}{[0.08]^1[1/9]^5 + [0.08]^1[1/8]^5 + [0.08]^1[1/6]^5 + [0.08]^1[1/6]^5} = \frac{0.000002441}{0.000024371} = 0.10$$

$$p^1_{BD}(t) = \frac{[0.08]^1[1/6]^5}{[0.08]^1[1/9]^5 + [0.08]^1[1/8]^5 + [0.08]^1[1/6]^5 + [0.08]^1[1/6]^5} = \frac{0.000010288}{0.000024371} = 0.422$$

$$p^1_{BE}(t) = \frac{[0.08]^1[1/6]^5}{[0.08]^1[1/9]^5 + [0.08]^1[1/8]^5 + [0.08]^1[1/6]^5 + [0.08]^1[1/6]^5} = \frac{0.000010288}{0.000024371} = 0.422$$

Paso 3: Se escoge el siguiente nodo aleatoriamente tomando en cuenta las probabilidades, las cuales se calcularon en el paso anterior, para esto generamos un numero aleatorio entre [0 - 1].

Numero aleatorio = 0.41

Nodo A	Nodo C	Nodo D	Nodo E
0 – 0.05	0.05 – 0.15	0.15 – 0-57	0.57 – 1.0

Nos fijamos en cual rango encaja el número aleatorio y ese será nuestro nodo seleccionado, en este caso el nodo al que nos desplazaremos es al Nodo D.

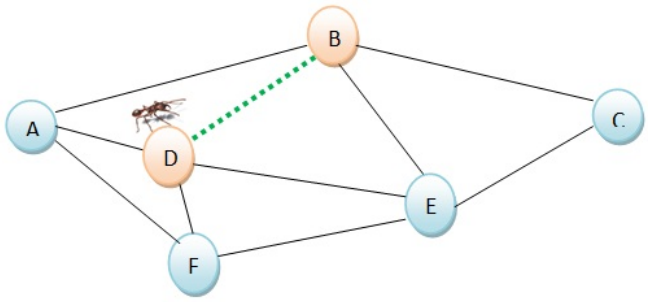
Paso 4: Una vez se ha seleccionado el nodo al cual desplazarse este se agrega a la lista tabú y se traza el recorrido de la hormiga para ver si aún hay nodos candidatos disponibles.

	1	2	3	4	5	6
Hormiga 1	B	D				
Hormiga 2	D					
Hormiga 3	A					

Así es como quedaría el recorrido después de agregar un nodo a la lista tabú.

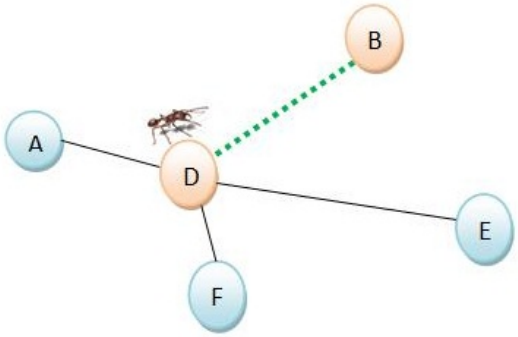
Paso 5: Checar si aún hay nodos candidatos disponibles, si los hay se repetirá el nuevamente todo este proceso desde el Paso 1.

Figura 8.6: Muestra el movimiento de la hormiga basado en el proceso de evaluación.



Al ver el grafo podemos apreciar que aún hay nodos candidatos a los cuales podemos ir, en este caso serían: Nodo A, Nodo E y Nodo F. Recordemos que no podemos regresar a un nodo que que ya ha sido recorrido, por lo que el Nodo B no se toma en cuenta.

Figura 8.7: Muestra los nodos validos en base a la nueva posición de la hormiga.



Como aún hay nodos candidatos disponibles a los cuales podemos ir comenzaremos nuevamente calcularemos las probabilidades para desplazarse a uno de los nodos y así seguir el proceso hasta que ya no haya más nodos candidatos a los cuales poder ir, cuando se llega a ese resultado podemos concluir con el proceso y tomar la lista tabú como nuestro mejor recorrido.

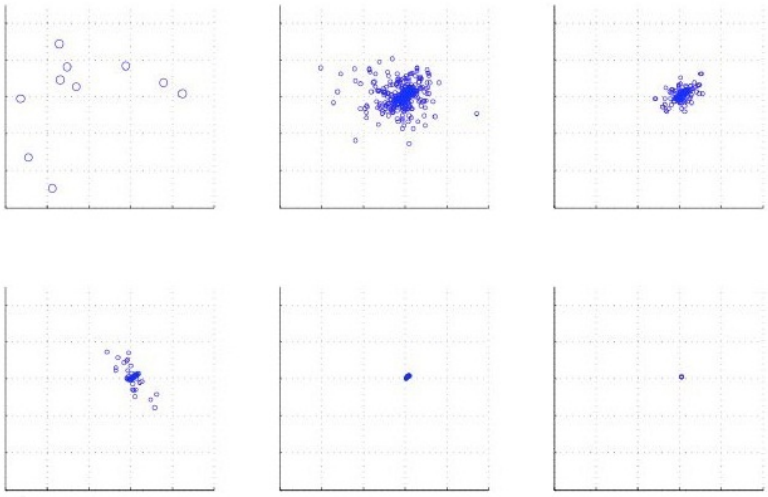
8.2.2 Optimización por Cumulo de Partículas (Particle Swarm Optimization, PSO).

La Optimización por Cumulo de Partículas, es un algoritmo de optimización que fue introducido en 1995 por Kennedy y Eberhart. Este algoritmo está bioinspirado en el comportamiento de las parvadas de aves al buscar alimento.

Las parvadas normalmente no conocen a priori el lugar donde se encuentra el alimento más si la distancia aproximada a este, por lo que como estrategia siguen al ave que se encuentra más cercana a la fuente de alimentación.

PSO simula dicho comportamiento para resolver problemas de optimización, generando posibles soluciones, donde cada una de ellas simula a un ave (dentro del espacio de búsqueda) que está en constante movimiento. La trayectoria y velocidad con la que cada partícula se mueve es ajustada basándose en la mejor posición que ha obtenido ella misma y a la mejor solución global o local (de acuerdo al vecindario definido por el usuario) encontrada hasta el momento por la parvada, lo que conduce a que se encuentre una buena solución para resolver el problema (Chen, 2008) (García, 2006).

Figura 8.8: Muestra el comportamiento de las partículas en diferentes etapas del PSO.



PSO puede utilizar dos tipos de representaciones la binaria y la real (León, 2009). El pseudocódigo de PSO con representación binaria es:

```
Hacer
        Para i=1 hasta numero de particulas
            Si f ( xi ) > f ( pi ) entonces
                \{se evalua la aptitud\}
                    Para d=1 hasta
                        dimensiones
                            pid = xid
                                        \{
                                pid es la
                                mejor
                                posicion
                                obtenida por
                                 la
                                particula id
                                \}
                    Siguiente d
```

```
                    Fin si
                    g = i
                    Para j = 1 hasta indices de los
                        vecinos
                            Si f (pj) >f (pg)
                               entonces
                                    g = j  \{g es
                                       el indice de
                                        la mejor
                                       particula
                                       del
                                       vecindario\}
                    Siguiente j
                    Para d = 1 hasta dimensiones
                            vid (t) = vid (t - 1) +
                                φ1 (pid - xid (t -
                               1)) + φ2 (pgd - xid
                               (t - 1))
                            vid[U+FFFD]-Vmax , + Vmax
                               )
                            Si ρid < s ( vid ( t ))
                                entonces
                                    xid ( t ) = 1
                            sino
                                    xid ( t ) = 0
                            fin si
                            Siguiente d}
                    Siguiente i
        Hasta (criterio de parada)}
```

Dónde:

xid (t) es el estado actual del d-ésimo bit de la partícula i
vid (t -1) es la velocidad de la particula i (se inicializa a 1 o 0)
pid es el estado actual del d-ésimo bit de la partícula en la mejor posición experimentada
pg es la mejor particula del vecindario
ϕ es un número aleatorio positivo con distribución uniforme y límite superior predefinido
ρid es d-ésimo número aleatorio de un vector de números aleatorios con valor entre 0 y 17
Vmax es una constante definida al inicio del algoritmo y que limita los valores de vid
s (vid (t)) representa la evaluación de vid (t) en la función sigmoide, esto es:
$s(v_{id}) = \frac{1}{1+e^{-v_{id}}}$

La función sigmoide reduce su entrada a valores 0 o 1 y sus propiedades la hacen adecuada para ser usada como umbral de probabilidad (Eberhart,

Shi, & Kennedy, 2001).

Figura 8.9: Muestra el comportamiento de las partículas en el PSO

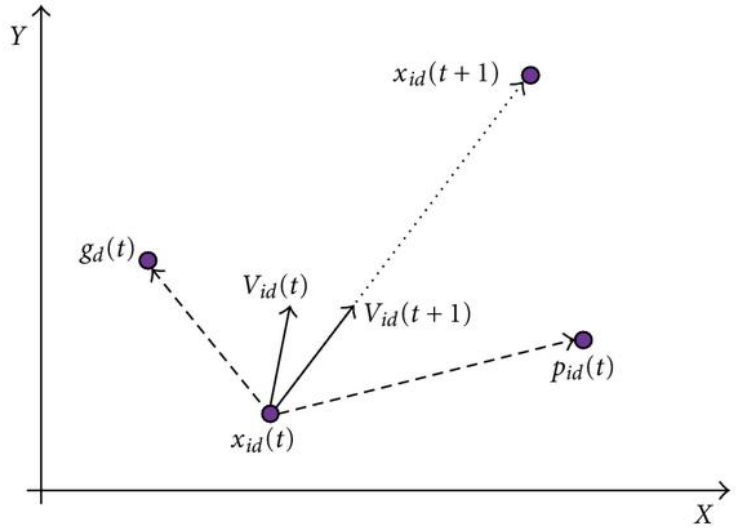
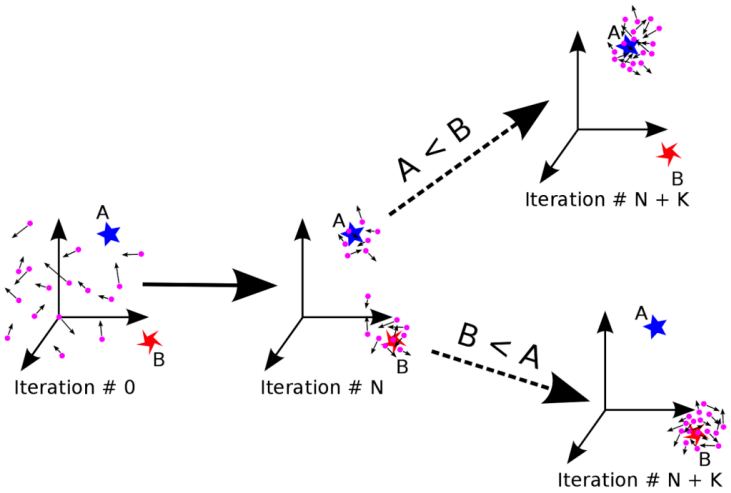


Figura 8.10: Comportamiento del PSO



PSO con números reales trabaja de modo similar a la versión binaria y su pseudocódigo es el siguiente:

Hacer

Para i=1 hasta numero de particulas

```
                    Si f ( xi ) $>$ f ( pi )
                       entonces   \{se evalua la
                       aptitud\}
                            Para d=1 hasta
                               dimensiones
                                    pid = xid
                                                \{
                                       pid es la
                                       mejor
                                       posicion
                                       obtenida por
                                        la
                                       particula id
                                       \}
                    Siguiente d
                    Fin si
                    g = i
                    Para j = 1 hasta indices de los
                        vecinos
                            Si f (pj) $>$f (pg)
                               entonces
                                    g = j  \{g es
                                       el indice de
                                        la mejor
                                       particula
                                       del
                                       vecindario\}
                    Siguiente j
                    Para d = 1 hasta dimensiones
                            vid (t) = vid (t - 1) +
                                $\phi$1 (pid -- xid
                                (t - 1)) + $\phi$2
                               (pgd -- xid (t - 1))
                            vid[U+FFFD] - Vmax , + Vmax
                               )
                            xid (t) = xid (t - 1) +
                                vid  (t)
            {Siguiente d
            Siguiente i
    Hasta (criterio de parada)
```

En el PSO con valores reales, las posiciones de las partículas representan como tal una posible solución al problema planteado, por lo que cada subíndice d representa el d-ésimo valor de un vector de números reales. A diferencia de PSO versión binaria, la actualización de la posición de la partícula se lleva a cabo directamente, sin utilizar la función sigmoide.

En la actualidad existen múltiples variantes de PSO, pero los pseudocódigos mostrados son de las versiones clásicas

8.3 Conclusión.

Como se puede apreciar los algoritmos que se encuentran clasificados dentro de Swarm Intelligence permiten obtener resultados de muy buena calidad en problemas de optimización.

Aun cuando PSO trabaja con poblaciones este algoritmo no utiliza los procesos evolutivos, sino guía a las partículas hacia espacios más prominentes pero realizando un proceso exploratorio del camino entre cada partícula y la mejor de la población.

ACO utiliza un proceso constructivo que se va formando en base a cada movimiento que hace la hormiga utilizando los rastros de feromona dejados por otras hormigas sin perder un cierto grado de aleatoriedad que le permite escapar de óptimos locales.

Existen muchas otras técnicas dentro de esta clasificación que pueden ayudar dependiendo del problema obtener mejores resultados y en tiempos mucho mejores.

8.4 Cuestionario de Colonias de Hormigas

1. ¿Que es un algoritmo de colonia de hormiga?
   - Una Heurística inspirada en las hormigas
   - Una Meta-heurística bio-inspirada comportamiento de las hormigas
   - Una Función de búsqueda
2. ¿Qué es una meta-heurística?
   - Un algoritmo de búsqueda
   - Una Heurística de prueba y error pero orientada a las computadoras
   - Un algoritmo para la resolución de problemas
3. ¿Dónde fue aplicado por primera vez el algoritmo de colonia de hormiga?
   - En el problema de la mochila
   - En el problema de TSP
   - En el Problema de clique máximo
4. ¿Quién fue el creador del Ant System (Sistema de Hormigas)?
   - Mauro Burattari
   - John Holland
   - Marco Dorigo
5. ¿Son tablas que se manejan para la resolución de problemas?
   - Tabla de visibilidad y tabla feromonas
   - Tabla de búsqueda y tabla de objetos
   - Tabla de recorrido y tabla de nodos
6. ¿Cuál es el principal objetivo de los ACG?
   - Reducir el tiempo de cálculos
   - Agilizar el proceso de solución
   - Utilizar el menor costo computacional posible

7. ¿La visibilidad está dada por?
   - Las distancia entre los nodos del problema
   - Información específica del problema
   - El número de hormigas del problema
8. Es la representación de la concentración depositada por las hormigas a través del tiempo.
   - Tabla de distancias
   - Tabla de recorrido
   - Tabla de feromonas
9. ¿Cuál es la principal ventaja de un algoritmo de colonia de hormiga?
   - Que es el algoritmo más rápido y eficiente
   - Que garantiza encontrar el óptimo local el cualquier problema
   - siempre encuentra la solución exacta de los problemas
10. ¿Que es una lista tabú?
    - Es donde se almacena el recorrido de las hormigas
    - Es donde se almacenan las distancias
    - Es donde se guardan los óptimos de cada iteración

8.5 Bibliografía.

Abraham, A., Grosan, C., & Ramos, V. (2006). Swarm Intelligence in Data Mining. Berlin, Heidelberg: Springer Verlag.

Bullnheimer , B., Hartl, R., & Strauss, C. (1997). An improved ant system algorithm for the vehicle routing problem. Annals of Operations Research, 319-328.

Chen, M. (2008). Second Generation Particle Swarm Optimization. Evolutionary Computation, CEC 2008, 90-96.

Cruz Cortés, N. (2004). Thesis PhD in Sciences, Computer Science option. Sistema Inmune Artificial para Solucionar Problemas de Optimización.

Dorigo, M., & Gambardella, L. (1997). Ant colony system: a cooperative learning approach to the traveling salesman problem. Evolutionary Computation, IEEE Transactions on (Volume 1 , Issue 1), 53-66.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). The Ant System: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics-part B.

Eberhart, R., Shi, Y., & Kennedy, J. (2001). Swarm Intelligence. Elsevier.

Fritsch, D., Wegener, K., & Schraft, R. (2007). Control of a robotic swarm for the elimination of marine oil pollutions. Swarm Intelligence Symposium, 2007. SIS 2007. IEEE, 29-36.

Gambardella, L., & Dorigo, M. (1995). Ant-Q: A Reinforcement Learning approach to the traveling salesman problem. Twelfth International Conference on Machine Learning, A. Prieditis and S. Russell (Eds.), 252-260.

Gambardella, L., Taillard, É., & Agazzi, G. (1999). MACS-VRPTW: A Multiple Colony System For Vehicle Routing Problems With Time Windows. New Ideas in Optimization, 63-76.

García, J. (2006). Algoritmos basados en cúmulo de partículas para la resolución de problemas complejos. Tesis de Licenciatura. Malaga, España.

Kassabalidis, I., El-Sharkawi, M., Marks, R., Arabshahi, P., & Gray, A. (2001). Swarm intelligence for routing in communication networks. Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE (Volume:6), 3613 - 3617.

León, A. (2009). Diseño e implementación en hardware de un algoritmo bioinspirado. México, D.F.

Parpinelli, R., Lopes, H., & Freitas, A. (2002). Data mining with an ant colony optimization algorithm. Evolutionary Computation, IEEE Transactions on (Volumen 6, Issue 4) , 321-332.

Peng, X., Venayagamoorthy, G., & Corzine, K. (2007). Combined Training of Recurrent Neural Networks with Particle Swarm Optimization and Backpropagation Algorithms for Impedance Identification. Swarm Intelligence Symposium, 2007. SIS 2007. IEEE, 9-15.

Pilat, M., & White, T. (2002). Using Genetic Algorithms to Optimize ACS-TSP. Ant Algorithms. Lecture Notes in Computer Science Volume 2463, 282-287.

Stützle, T., & Hoos, H. (1998). Improvements on the Ant-System: Introducing the MAX-MIN Ant System. Artificial Neural Nets and Genetic Algorithms, 245-249.

White, T., Pagurek, B., & Oppacher, F. (1998). ASGA: Improving the Ant System by Integration with Genetic Algorithms. Genetic Programming.

9 — Algoritmos en Paralelo mediante Uso de GPUs

José Alberto Hernández A., Crizpín Zavala D. , Nodari Vakhnia, Oswaldo Pedreño
Universidad Autónoma del Estado de Morelos, México

9.1 OBJETIVO

El presente capítulo tiene como objetivo el describir una metodología para paralelizar algoritmos bioinspirados en unidades de procesamiento gráfico (GPU- Graphics Processing Units por sus siglas en Inglés) empleando el lenguaje de programación CUDA (C Unified Device Architecture). Para este propósito, en primer lugar se discute brevemente el funcionamiento de las GPU, posteriormente los diferentes tipos de paralelismo haciendo énfasis en el paralelismo orientado a datos, en el que se busca paralelizar ciclos for anidados, para posteriormente analizar implementaciones secuenciales de algoritmos bioinspirados como el Algoritmo de Colonia de Hormigas (ACO) y la paralelización de algunas de sus funciones en núcleos de procesamiento paralelo o Kernels. Los fundamentos teóricos del paralelismo orientado a datos son puestos en práctica a través de un caso de estudio, así como de ejercicios propuestos.

9.2 RESUMEN DEL CAPÍTULO

Uno de los mayores beneficios que un usuario puede obtener de un ambiente de simulación es el de poder experimentar y analizar un modelo, en lugar del sistema real, con un menor costo, tiempo, y riesgo. El desarrollo de ambientes de modelado y simulación es reconocido como una tarea difícil debido a la falta de técnicas que permitan registrar las experiencias exitosas del diseño de las abstracciones del software que requieren este tipo de aplicaciones. El presente trabajo propone una metodología para paralelizar algoritmos secuenciales que contienen ciclos for aninados que realizan cálculos independientes para ser ejecutadas en unidades de procesamiento gráfico.

Hasta hace unos pocos años los equipos que permitían el procesamiento en paralelo resultaban costosos y eran compartidos por grupos de científicos e investigadores, de manera que se tenía que hacer cola para hacer uso de los

mismos; afortunadamente a mediados de los 90's se abrió la posibilidad de este tipo de procesamiento mediante las Unidades de Procesamiento Gráfico, primeramente para el renderizado de vectores de vídeo juegos y algunos años más tarde para la realización de operaciones de punto flotante, así con el correr del tiempo y evolución de las tecnologías, las tendencias indican que cada centro de investigación o facultad buscan su propio equipo de procesamiento en paralelo mediante el uso de tecnologías heterogéneas que implican microprocesadores acompañados de cientos o miles de núcleos de procesamiento incorporados en GPUs. En el presente trabajo se propone la implementación de algoritmos en paralelo empleando las Unidades de Procesamiento Gráfico GPUs y el lenguaje de programación CUDA. Ambos elementos temas actuales de discusión por la comunidad científica y académica gracias a su gran potencia de procesamiento, bajo consumo de energía y costo, y su versatilidad para ser utilizado en diferentes aplicaciones.

9.3 CONOCIMIENTOS PREVIOS

Para una mejor comprensión de los contenidos de este capítulo se recomienda al estudiante contar con un nivel básico de experiencia en la programación estructurada y programación en lenguaje C, conocimientos de heurísticas y sus algoritmos, así como de principios básicos de programación en paralelo.

9.4 INTRODUCCIÓN A LAS GPUs y CUDA

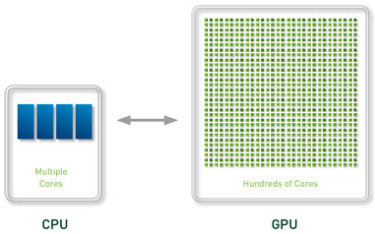
Las GPU (Graphical Process Units) o Unidades de Procesamiento Gráfico son una tecnología muy potente en comparación con la de una CPU o Unidad de Procesamiento Central, ya que supera la frecuencia de reloj que maneja la CPU (más de 500MHz). Este mejor rendimiento se debe a que las GPU están optimizadas para el desarrollo de cálculo de coma flotante. Además la memoria propia con la que cuenta la GPU permite el acceso mucho más rápido con respecto al de una CPU, esto beneficia para que los resultados sean leídos y/o guardados de manera más rápida.

Una GPU posee muchas funciones entre las que sobresale el procesamiento de datos de manera como si fuera un gráfico en el que se toman en cuenta tanto los vértices como los pixeles. La memoria de las GPU es muy diferente a las de la CPU, esto se puede ver por el número de núcleos que cuenta una GPU.

A continuación se expone una figura en donde se comparan los núcleos entre una CPU y una GPU.

Como se puede observar en la figura reffig:rId240. El CPU solo contiene 4 núcleos a diferencia de la GPU que puede tener cientos o miles de núcleos

Figura 9.1: Comparación de núcleos entre CPU y GPU (Nvidia, 2011)



lo que la hace más rápida.

9.4.1 Antecedentes históricos de las GPU's

Durante los años 80's, en los que aconteció la aparición de la primera computadora personal, con una velocidad aproximada de procesamiento de 1MHz, se había logrado dar un gran paso para el campo de la computación. Posteriormente, durante los 90's, la evolución fue aún más significativa debido a la creciente popularidad de Microsoft Windows, debido principalmente a que ésta ofrecía un ambiente más amigable que ya utilizaba aceleradores en segunda Dimensión (2D) para sus pantallas en sus Sistemas Operativos (Sanders & Kandrot, 2010). Posteriormente, en 1992, la Empresa "Silicon Graphics" liberó la librería openGL, la que permitía la programación de aplicaciones en 3D, estos elementos lograron una considerable popularidad y comenzaron a utilizarse primariamente para perfeccionar la aceleración gráfica, como es el caso de videojuegos; fue así como en el año 1993, surgió la Empresa "Nvidia", fundada por Jen Hsun Hung ,Chris Malachowski y Curtis Priem (Nvidia, Company History | Nvidia, s.f.). Esta compañía en 1999 lanza la primera tarjeta gráfica la Nvidia GeForce 256, esta fue diseñada con la función específica de aceleramiento gráfico, jamás se pensó que mediante estas se lograría un incremento en velocidad de procesamiento de datos.

Desde finales de los 90's, las GPU's eran más fáciles para programar, esto gracias a que varias empresas se dieron a la tarea de crear un lenguaje de programación para GPU's ya que los usuarios finales aun conociendo lenguajes de programación como OpenGL les resultaba muy difícil su programación, ya que tenían que manipular el código como si se tratara de triángulos y polígonos. La universidad de Stanford re-imagino la GPU como un procesador de flujos (Nvidia, 2012).

En esencia durante la primera década de este siglo, las GPU's fueron diseñadas para producir un color para cada píxel, al analizar esto los programadores e investigadores vieron que era posible cambiar las unidades de píxeles de colores por datos, facilitando el trabajo de programarlas (Sanders & Kandrot, 2010).

En 2003, se creó el primer modelo de programación para las GPU's llamado "Brook" que adecuó el código C para llevarlo a la paralelización. Se utilizaban

flujos de datos, kernels y los operadores de reducción, todo esto estableció a la GPU como un procesador con fines generales tomando en cuenta un lenguaje de alto nivel. Este modelo de programación fue aceptado gracias a que era siete veces más rápido que los lenguajes de programación existentes en ese momento (Nvidia, 2012).

Con respecto a la programación, la empresa que más avances ha realizado es Nvidia, cuyos fabricantes sabían que con el hardware tan amplio que contaban las GPU's podían elaborar un lenguaje de programación propio. Nvidia contacto a lan Buck para crear un lenguaje de programación combinando C en una perfección para manipular las GPU's, así la empresa Nvidia en 2006 lanzó al mercado CUDA (Compute Unified Device Architecture) que fue tomado como la mejor solución para la programación en GPU's en ese momento (Nvidia, 2012).

9.5 FUNDAMENTOS TEORICOS

El procesamiento paralelo ha evolucionado desde que las computadoras tenían 2 núcleos, y que se empezó a utilizar supercomputadoras para el procesamiento de información en cantidades exorbitantes. CUDA de Nvidia reduce el tiempo de procesamiento aprovechando la gran cantidad de núcleos que tiene una GPU, esto quiere decir que el código ejecutado en GPU's es en paralelo ya que puede realizar "n" cantidad de operaciones al mismo tiempo asignando uno de sus núcleos de procesamiento para cada una de ellas. Esto es muy útil hoy en día, ya que muchas de las aplicaciones existentes conllevan un alto grado de paralelismo. Aquí cabe señalar que el código secuencial seguirá siendo secuencial mientras que aquello que requiera una gran cantidad de cálculos matemáticos puede ser paralelizado.

Es decir que se puede aprovechar la fuerza bruta que tienen las GPU's en cuanto a procesamiento y tratar de llevar algoritmos a la paralelización ya que con esto se reducirán los tiempos y se optimizarán los cálculos de operaciones de coma flotante. Los modelos actuales de GPU's nos ofrecen trabajar con 500-600MHz que si se compara con los 3.8-4GHz de las CPU podemos llegar a la conclusión que es mejor trabajar con una combinación entre GPU's y CPU's. Esto nos favorece en tanto que aunque se cuenta en la actualidad con cluster's de CPU es mejor trabajar con GPU's ya que en costo se hace una diferencia realmente grande y en cuanto al procesamiento se puede mejorar hasta 3 veces más el tiempo en terminar un proceso computacional.

La paralelización es una división de trabajo entre los procesadores de una CPU y una GPU, esto quiere decir que los procesos de cálculo intensivo de datos se llevarán a cabo en la parte de la GPU y la parte de diseño de las interfaces de usuarios se ejecutarán en la parte de los procesadores de la CPU, dejando entonces el trabajo más pesado en tanto a procesamiento de datos se habla a la GPU que puede realizarlo de una manera mucho más rápida gracias a sus grandes cantidades de núcleo de procesamiento (Sanders & Kandrot, 2010).

En Noviembre del 2006 Nvidia sacó al mercado su arquitectura de programación CUDA. Esta arquitectura fue diseñada para el cálculo paralelo en donde se puede aprovechar la gran potencia en procesamiento que ofrecen las GPU's. Esto ayuda a incrementar extraordinariamente el rendimiento de la plataforma.

En seguida se muestra una tabla con todas las familias de tarjetas con su mejor modelo cada uno, su capacidad computacional y la cantidad de núcleos de procesamiento (Nvidia, 2013).

Tabla 3.1 Tarjetas NVIDIA que soportan CUDA

FAMILIA	MODELO CON MAYOR CAPACIDAD	CAPACIDAD COMPUTACIONAL	NÚCLEOS PARA PROCESAMIENTO
Tesla	Tesla K20	3.5	2946
Quadro	Quadro K5000	3.0	1536
NVS	Quadro NVS 450	1.1	16
GeForce	GeForce GTX TITAN	3.5	2688

9.6 CUDA (Compute Unified Device Architecture)

De acuerdo a (Sanders & Kandrot, 2010) CUDA es un lenguaje de programación que es utilizado para la paralelización con propósito general introducido por la empresa Nvidia. Este lenguaje incluye un conjunto de instrucciones que trabajan sobre la arquitectura de las GPU's, el cual es muy utilizado por la facilidad que les da a los programadores ya que el código generado es muy parecido al del lenguaje C.

CUDA tiene una gran cantidad de áreas de aplicación ya que día a día se encuentran más aplicaciones prácticas para esta nueva tecnología. Algunas de las áreas de aplicación donde podemos observar el uso de CUDA son: biología, química computacional, procesamiento de videos entre otras muchas áreas más, ya que CUDA combinado con GPU's puede ser utilizada para propósitos generales. Con CUDA C los programadores se preocupan más por la tarea de paralelización que por el manejo de la aplicación.

Además CUDA C apoya al cómputo heterogéneo. Es decir que la parte del código secuencial de la aplicación se ejecutará en el CPU, y las partes del código paralelo se llevan a cabo en la GPU. Para este propósito CUDA trata como dispositivos independientes tanto a la CPU como a la GPU aunque trabajan como uno solo. Esto hace que la CPU y la GPU no entren en competencia por recursos de memoria si no que cada uno use sus propios recursos de manera independiente.

Cuando una GPU trabaja con CUDA cuenta con cientos de núcleos, esto quiere decir que podemos ejecutar miles de threads (hilos) de programación simultáneamente. Cada núcleo cuenta con recursos compartidos, esto nos ayuda a que se realice de manera más eficiente la paralelización.

En la actualidad la programación se está enfocando más hacia el coprocesamiento entre una CPU y una GPU (llamada también programación paralela) dejando atrás la forma tradicional de realizar todos los procesos de forma centralizada (llamada programación secuencial), ya que la programación paralela se realiza de manera más rápida gracias a los múltiples núcleos que nos ofrecen las GPU's. Todo esto ha llevado a que CUDA sea aceptado por la comunidad científica ya que facilita el descubrimiento de nuevas aplicaciones que requieren ser aceleradas, utilizando tanto el coprocesamiento como la computación paralela que nos ofrece CUDA. Los nuevos sistemas operativos contemplan la computación con GPU's ya que cada vez se está utilizando con mayor frecuencia esta tecnología (Nvidia, 2012).

Para entender CUDA se debe de conocer los conceptos principales que hacen la diferencia de la arquitectura de programación en lenguaje C.

Los conceptos principales que analizaremos de CUDA son (Nvidia, 2011a):

- Kernels
- Jerarquía de Hilos
- Jerarquía de memoria
- Programación heterogénea

Kernels. El lenguaje de programación CUDA C tiene flexibilidad para utilizar funciones que se utilizan en el lenguaje de programación C. Estas funciones pueden llamarse desde cualquier parte del código y ejecutarse n veces en paralelo ya que estas las pone en un thread (hilo) de programación.

Para crear un kernel es necesario utilizar una declaración específica además de poner el número de identificador de esa función el cual se puede ver en el siguiente ejemplo:

___global___<<<4,1>>>

En donde ___global___ es la declaración de la función y <<<4,1>>> es el identificador o mejor llamado threadId. Así es como podemos hacer el uso de varias llamadas al kernel, en el cual se encontrarán todas las funciones para después poder ser utilizado en cualquier parte, y las veces que sea necesario en un código paralelo.

9.6.1 Jerarquía de Hilos

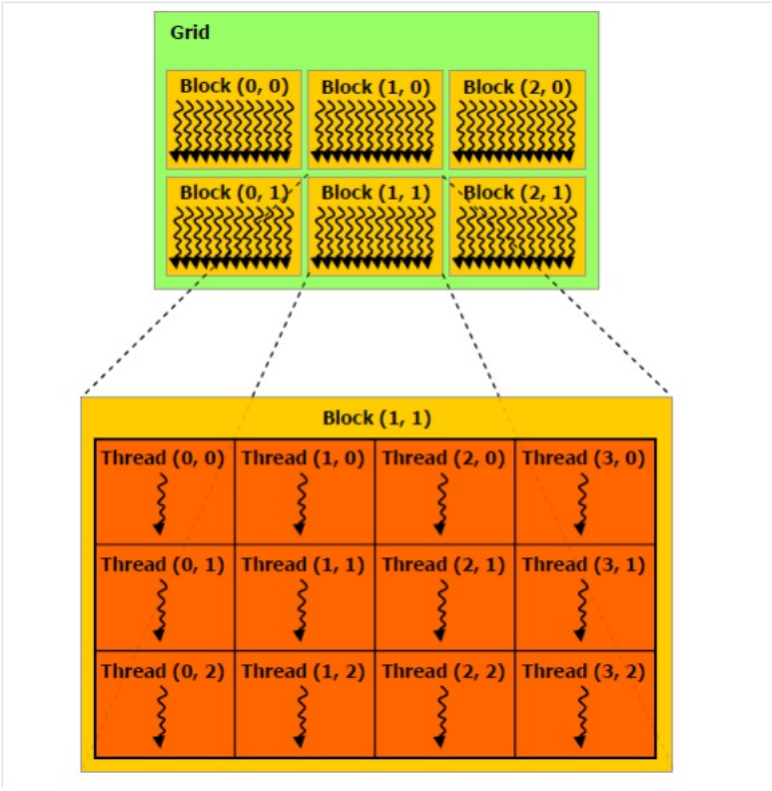
Los hilos se identifican por el threadIdx el cual hace un bloque de estos mismos para, de esta manera, sea más fácil invocarlos en un código paralelo.

Cada bloque contiene n cantidad de thread (hilos) de programación, dentro de estos bloques se puede identificar a los diferentes thread gracias a su ID, el cual es conformado por coordenadas dx y dy en un bloque bidimensional.

Hay un límite de hilos que puede contener un bloque, esto por la cantidad de recursos en la memoria, actualmente las GPU's dan la facilidad de que cada bloque contenga hasta 1024 threads.

Al conjunto de bloques se le conoce como una grid, en esta los bloques también son identificados por un ID llamado blockIdx con dos coordenadas dx y dy.

Figura 9.2: Arquitectura CUDA (Nvidia, 2011 a)



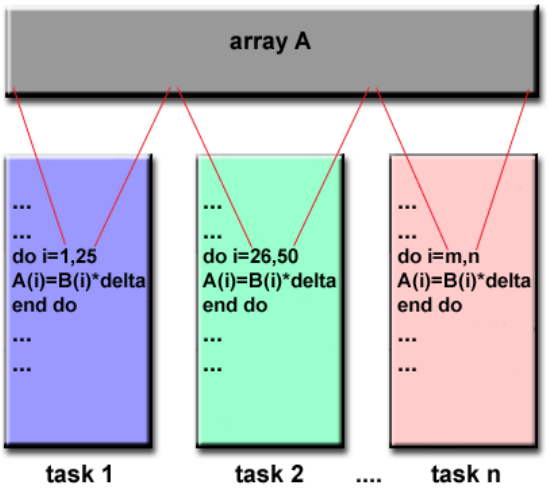
En la figura reffig:rId241 se muestra cómo se hace referencia a los componentes que incluye la programación en CUDA.

9.6.2 Paralelismo basado en datos

Existen distintos tipos de programación en paralelo, entre los que podemos encontrar: Paralelismo basado en datos, Paralelismo Híbrido, Simple Pro-

gram Multiple Data (Programa simple múltiples datos) y Multiple Program Multiple Data (Múltiple programa múltiples datos). CUDA usa el paralelismo basado en datos, también conocido como "Modelo de Espacio de direcciones globales particionadas» o por su nombre en inglés «Partitioned Global Address Space (PGAS)"; Según (Pedemonte, Alba, & Luna, 2011) y (Blaise, 2013), la mayoría de los programas en paralelo buscan realizar operaciones desde alguna estructura de datos, llámese: vector, matriz, cubo, etc. Esto se lleva a cabo mediante estructuras de control, sin embargo no necesariamente se trabajan de manera ordenada, ya que como es sabido, las estructuras de memoria compartida (como son las GPU's) permiten el acceso a estas estructuras mediante punteros y memoria global. Véase la figura reffig:rId241.

Figura 9.3: Representación del modelo de Paralelismo Basado en Datos fuente: (Blaise, 2013)



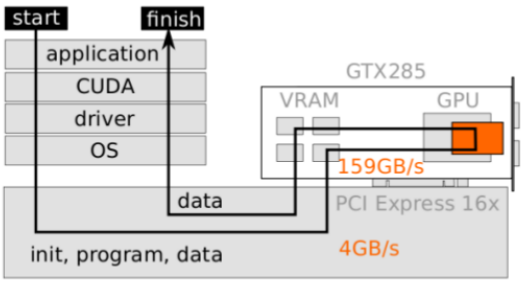
En la figura anterior se logra apreciar cómo funciona este tipo de programación en paralelo, cada uno de los cuadros representa una tarea en específico y cada uno de ellas trabaja en distintos tipos de segmento de un arreglo, no necesariamente en el mismo orden pero si con alguna relación entre ellos.

El presente trabajo propone paralelizar algoritmos secuenciales utilizando el esquema de paralelismo basado en datos (ver figura reffig:rId241) utilizando CUDA.

Lo anterior significa que lo que sea secuencial del algoritmo seguirá siendo secuencial, mientras que aquello que requiera cálculos intensivos independientes contralados mediante ciclos for aninados, serán ejecutados en kernels dentro las GPUs, cuando estos cálculos estén terminados, los resultados serán enviados de regreso a la CPU para continuar con la ejecución del algoritmo.

En la figura anterior se describe como interactúa el CPU y la GPU du-

Figura 9.4: (secuencial) + GPU (paralelo) (Pospichal, Jaros, & Schwarz, 2010)



rante la ejecución de un programa en CUDA, esto se realiza mediante el compilador de CUDA llamado nvcc, el cual se encarga de separar el código del host (CPU) del código del GPU (device). Cada uno de estos códigos es tratado por un compilador, para el caso del Host se utiliza un compilador de propósito general, este puede variar dependiendo de la plataforma en la que se esté trabajando, para esta investigación que se realizó sobre Windows, el compilador recomendado por CUDA es el de Microsoft Visual Studio "cl", por otra parte el código de CUDA es ejecutado por el nvcc en el device, el host reserva y traslada el código al device, una vez que el device termina el tratamiento a los datos procede a regresar los datos al host.

9.7 Metodología

Basados en methodología propuesta en Hernandez et al. 2011.,

Propuesta Metodológica para explotar las capacidades de procesamiento en paralelo de Sistemas Heterogéneos

Nuestra propuesta considera los siguientes pasos:

0. Inicio

1. Identificar el problema a resolver

¿Cual es problema que pretendemos resolver? Y no solo nos referimos a mejorar la capacidad de cómputo instalada. Mas bien en este punto debemos plantear que procesos deseamos paralelizar.

1.1. ¿Es susceptible de ser paralelizado? Una vez que hemos identificado el problema. Se debe analizar: ¿Se puede paralelizar? Si la respuesta es sí, continuamos con el paso dos, en caso contrario continuamos en el punto 6 (Fin).

2. Configuración del equipo

Se debe identificar el tipo de Tarjeta gráfica GPU con la que cuenta nuestro

equipo (Nvidia, AMD u otra), así como el número de procesadores. Hecho lo anterior se deberán descargar los drivers para la GPU dependiendo de su fabricante y del número de bits que procesen los microprocesadores (32 o 64 bits) y del sistema operativo que se use. Así mismo se deberá descargar OpenCL 1.0 y Python. En Linux se utiliza el comando lspci para listar todos los dispositivos conectados en las ranuras PCI de la computadora o estación de trabajo.

3. Prueba del equipo

Si el sistema fue configurado exitosamente, y ya se tiene correctamente instalado Python y las librerías requeridas por el mismo (p.e. Numpy para operaciones con matrices), se podrá correr sin problema cualquiera de los programas de demostración disponibles en PyOpenCL, si el programa elegido no corre o el sistema indica la falta de alguna librería se deberá regresar al paso 2. Se recomienda ejecutar el programa benchmark-all.py para dicho fin. Para correr el programa desde la línea de comandos o terminal se deberá usar la instrucción siguiente:

usuario:~/ruta/pyopencl-0.92/examples$ python benchmark-all.py

Para ejecutar el programa desde el Idle de Python. Se deberán seguir las siguientes instrucciones:

Entrar al IDLE, ir al menú de archivo, ir a la opción de abrir archivo, buscar la ruta del archivo benchmark-all.py, abrir el archivo y ejecutarlo directamente con F5 o bien con ir al menú ejecutar (Run) y posteriormente seleccionar la opción ejecutar módulo (Run Module).

4. Implementar el algoritmo

La implementación del algoritmo en particular puede ser una tarea más o menos consumidora de tiempo, pero de lo estamos seguros es que gracias a las bondades de OpenCL esto no tomará mucho tiempo. En este punto se recomienda consultar la documentación de PyOpenCL.

5. Evaluar los resultados

Se verifican los resultados obtenidos y si son correctos pasamos al punto 6, si no regresamos al punto anterior.

6. Fin

Se recomienda consultar la literatura para identificar las funciones susceptibles de ser paralelizadas, y posteriormente buscar alguna solución secuencial para resilver el problema en cuestión para identificar funciones o segmentos de código que contengan ciclos for anidados, para identificar funciones candidatas a ser paralelizadas, hecho esto se procede a implementarlas usando CUDA, se validan los resultados de estas funciones y se realiza una com-

paración del tiempo de procesamiento de la implementación secuencial con respecto al tiempo de procesamiento de la implementación paralelizada.

9.8 CASO PRÁCTICO

Con el fin de aplicar la metodología propuesta en la sección anterior, a continuación presenta el análisis y diseño de un programa en CUDA cuyo propósito es resolver el problema del agente viajero (TSP) utilizando Colonia de Hormigas Ant Colony (ACO). Este programa está basado en la solución propuesta por (Jones, 2010) y permite al usuario analizar distintas instancias del TSPLIB.

9.9 ANÁLISIS DE REQUERIMIENTOS

La parte externa del ambiente de ambiente de simulación de señales está representada por la interfaz gráfica, la cual está conformada por el conjunto de interactores y el evento a los cuales responde cada uno ellos (ver tabla 1).

9.9.1 El Problema del Agente Viajero

El problema del agente viajero (TSP) es un problema combinatorio ampliamente estudiado a lo largo del tiempo y por esta razón podemos hallar más de un algoritmo para su resolución. Para tener una idea sobre este problema, tenemos que entender que en este, un vendedor el cual tiene que recorrer n cantidad de ciudades (nodos), sin volver a pasar una ciudad ya antes visitada y terminar dicho tour en la ciudad inicial (nodo inicial) (véase la figura 1.2). El vendedor naturalmente quiere realizar la ruta más corta y en el menor tiempo posible. Este problema puede considerarse sencillo, ya que al ser un problema combinatorio solo se debe de analizar todas las rutas posibles para cubrir todos los nodos y elegir cual de ella es la menor, esto se puede realizar de una manera sencilla si consideramos pocos nodos como por ejemplo 5, pero al hablar de más de 1000 nodos las combinaciones de las rutas son demasiadas para llevar a cabo un análisis de una por una y elegir de esta manera la mejor (Cook, 2000).

El problema del agente viajero (TSP) es un problema clásico y ampliamente estudiado, puesto que es de fácil entendimiento pero es de difícil implementación, lo podemos observar al momento de pasar las formulas a cualquier lenguaje de programación.

Como se ha visto en la historia la cantidad de nodos ha ido aumentando, como por ejemplo en 1990 la instancia más grande de la cual se tenía una solución óptima era de 380 ciudades, en 1998 se reportó la solución óptima con la instancia más grande que se ha registrado, la instancia de 13,509 ciudades o nodos. Estos datos nos sirven para observar el gran progreso que se

ha obtenido con el paso del tiempo, no es raro que en un futuro se hallen soluciones óptimas para instancias de mayor tamaño (González, 1999).

Una de las soluciones que se ocupa para el TSP son los algoritmos de optimización, en muchos casos se utiliza el Algoritmo de Colonia de Hormigas, esto se realiza por varias razones como por ejemplo (Stützle & Dorigo, 1999):

(a) Los algoritmos son de fácil aplicación para solucionar el TSP.
(b) El TSP es considerado un problema NP-Duro
(c) El TSP te da la facilidad de encontrar diferentes tipos de soluciones y obtener en cada corrida un resultado diferente (debido a la probabilidad).
(d) El TSP es fácil de comprender, ya que no es necesario conocer tecnicismos para comprender el funcionamiento de este problema.

Una manera más formal de describir el TSP según (Stützle & Dorigo, 1999), es que se debe de entender que en un grafo (G) de cualquier tamaño, donde se ubican n cantidad de nodos (n), en donde cada nodo debe de estar conectado con todos los demás nodos. Para calcular la distancia total, se debe de sumar las distancias parciales, que se obtenga del recorrido de todos los nodos, donde el nodo inicial (i) al nodo más cercano a este (j), tiene una distancia parcial (d), en la cual se debe de cumplir que entre los nodos "i" y "j" la distancia tiene que ser igual, cumpliéndose esto al pasar al nodo más cercano este se convertirá en nodo i y se buscará el nodo más cercano a este convirtiéndose en el nodo j y así sucesivamente hasta volver al nodo donde se inició el tour sin que no falte ni un nodo de visitar y no repetir algún nodo ya visitado.

9.9.2 Áreas de Aplicación

El Problema del Agente Viajero (TSP), es aplicado en diferentes áreas, las 2 principales son en el área manufacturera y el área de logística.

En el área manufacturera el ejemplo en donde se puede implementar, puede ser, al tener una sola máquina que realice varias tareas, pero solo puede realizar una tarea a la vez y donde cada tarea tiene características distintas, entonces en este ejemplo, el TSP puede aplicarse si observamos cada tarea como un nodo y el tiempo en que tardan en cambiar las características para la nueva tarea se considera la distancia. Con lo anterior tenemos que idear un plan en donde las características de las tareas sean las más similares para que al momento del cambio no se pierda mucho tiempo en esta transacción y de esta manera ser más productivos.

Otra área de aplicación, es la logística, donde comúnmente es aplicado, en la empresa que vende productos perecedero en una ciudad, a una cantidad n de clientes las dudas que surgen serán:

(a) ¿Con cuántas unidades se debe de contar para repartir dichos productos

a todos los clientes?

(b) ¿En cuánto tiempo se recorre la ruta total visitando a todos los clientes?

Estas dos dudas se relacionan entre sí por qué con una sola unidad se puede repartir los productos a todos nuestros clientes pero el tiempo para cubrir toda la ruta será muy elevado, pero en caso contrario no se puede contar con una unidad para cada cliente ya que el costo de esto sería muy alto. Entonces se deben de considerar los 2 factores principales del TSP: el tiempo y la distancia, y llegar a un promedio aceptable.

Este ejemplo puede ser muy fácil de solucionar si solo se cuanta con 5 clientes ya que se puede realizar un análisis de todas las rutas posibles y obtener el mejor resultado, pero si se tienen 1002 clientes o más, el análisis se podría llevaría años para obtener todas las rutas posibles ya que para este problema el número de rutas crece exponencialmente (González y Ríos, 1999).

9.9.3 Ant Colony

Los Algoritmos de Colonia de Hormigas o Ant System, son un conjunto de algoritmos meta-heurísticos bio-inspirados, los cuales tienen aplicación para resolver problemas de optimización combinatoria. Este trabajo fue desarrollado como parte de tesis de doctorado de Marco Dorigo el cual toma como base la observación de las hormigas reales, las cuales su objetivo es el de encontrar la ruta más corta entre su nido (nodo inicial) y su comida (nodo final) (Baran y Almirón, 1999).

Según (Ponce, 2010) los antecedentes que se tienen en estas investigaciones son muy amplios ya que son fáciles de entender, pero a veces de difícil aplicación, y muchos investigadores y científicos han realizado trabajos con ellos, Ponce muestra una tabla donde se pueden ver diferentes problemas resueltos con colonias de hormigas, así como los autores y el año de publicación, su trabajo llega hasta el año 2010. En seguida se describirá la evolución que han tenido estos tipos de algoritmos incluyendo a sus autores y el año de publicación de los mismos.

- Ant System

El Ants System fue el primer algoritmo de colonia de hormigas. Este algoritmo fue creado por Marco Dorigo y publicado en su tesis de doctorado en el año de1992.

- Algoritmo Ant-Q

El Ant-Q (Gambardella & Dorigo, 1995) es un sistema híbrido ya que combina el Ant System con aprendizaje -Q (también conocido como Q-learning), siendo un modelo muy conocido de aprendizaje.

- Rank-based

Este algoritmo es una modificación del Ant System ya que toma en cuenta la cantidad de feromona depositada en la ruta entre nodos sumándola e incrementando el valor si otra hormiga utiliza esa ruta (Bullnheimer et al., 1997).

- Ant Colony System

Este algoritmo nace como un adicional mejorado del Ant-Q (Dorigo & Gambardella, 1997).

- MAX-MIN Ant System

Este algoritmo se caracteriza porque actualiza el valor para calcular la cantidad de feromona depositada en el tour además de establecer un valor mínimo y máximo como límites de acumulación de la feromona (Stützle y Hoos, 1998).

Estos trabajos son la base de lo que hoy se conoce en los famosos Ant´s System y algoritmos ACO.

Desde el año 1998 hasta la fecha han venido evolucionando y tomando un papel muy importante en la solución de problemas NP-Duros entre otros más, el problema en donde es más utilizado por su naturaleza es el Problema del Agente Viajero (TSP) ya antes analizado en esta investigación.

9.9.4 Análisis del algoritmo de colonia de hormigas

Antes de empezar un análisis del Algoritmo de Colonia de Hormigas (ACO), podemos mencionar la importancia de los sistemas híbridos, ya que este algoritmo está basado en este concepto (Jones, 2008). Un ejemplo, es que para el ser humano es muy fácil hacer cálculos matemáticos, y también tenemos memoria asociativa, lo que significa que a comparación con las hormigas estas no pueden recordar la ruta para llegar a su comida, en cambio el hombre puede asociar un momento, recordar imágenes y conceptos basados en hechos incompletos. Ya contemplado lo anterior podemos adentrarnos a el análisis del ACO.

Optimización de Colonia de Hormigas o ACO, es un algoritmo útil para resolver problemas en un espacio físico determinado o grafo. ACO trabaja con hormigas simuladas o virtuales a diferencia de las hormigas naturales las cuales como ya se ha explicado con anterioridad no pueden llegar a asociar el camino a seguir entre su colonia y su comida.

Las hormigas naturales son ciegas y al no poder ver utilizan la feromona. La feromona es una sustancia la cual van dejando las hormigas como rastro para saber el camino hacia su comida. Esto lo hacen para optimizar el tiempo y la distancia que tengan que recorrer, ya que al inicio de este ciclo se mandan varias hormigas a inspeccionar un lugar en busca de comida, la hormiga que encuentre esta ruta dejará un rastro de feromona más fuerte para que otras

hormigas la sigan y así lograr tener el camino más corto en tiempo y distancia.

El concepto de uso de ACO es que un determinado espacio físico o grafo, se consideran varias hormigas considerando la colonia como el nodo i o nodo inicio y la comida como nodo j o nodo final. Las hormigas recorrerán todo el espacio del grafo y depositarán niveles de feromonas, cuanto más fuerte sea este nivel más cerca se estará de encontrar la ruta óptima.

El pseudocódigo para ACO se basa en una serie de sencillos pasos para su ejecución los cuales se explicarán con detalle y están basa en Jones (2008) más adelante:

- Paso 1 Distribución de las hormigas en el grafo
- Paso 2 Selección de ruta
- Paso 3 Intensificación de la feromona
- Paso 4 Evaporación de la feromona
- Paso 5 Nuevo tour
- Paso 6 Se repite del Paso 2 en adelante hasta llegar a encontrar el tour óptimo para este algoritmo.

9.9.5 Implementación del algoritmo de colonia de hormigas sobre GPU's

Según (Cecilia et al., 2011) el Algoritmo de Colonia de Hormigas (ACO) se basa en la población siendo intrínsecamente paralelo por lo tanto se puede llevar al paralelismo para solucionar problemas NP-Duros. El Algoritmo se divide en 2 funciones principales: la construcción del tour o inicialización de las poblaciones y el nivel de feromona, estos pasos no se adaptan del todo a la arquitectura de las GPU's ya que proporcionan resultados alternativos basado en el paralelismo de datos. Para mejorar dichos resultados se deben de evitar instrucciones atómicas (funciones vistas por el programa como indivisibles) en la ejecución de dicho programa. El uso de ACO en GPU's todavía está en una etapa inicial esperándose en un futuro la mayor compatibilidad entre ellos.

En otro trabajo, Cecilia et al. (2012) indica que la clave para poder implementar el paralelismo es utilizar las operaciones de coma flotante y que algunos operadores aritméticos suelen ocasionar problemas con la computación paralela. Además de que el Algoritmo de Colonia de Hormigas (ACO) no encaja bien con la arquitectura de CUDA por el paralelismo de datos. Estos experimentos se realizaron en una GPU con dos tarjetas gráficas: Una Tesla C1060 y C2050, los resultados indican que ACO puede ser utilizado en problemas bio-inspirados.

Tsutsui (2012) utilizó ACO y la tecnología de las GPU's para resolver otro problema NP el QAP (Problema de Asignación Cuadrática), con esto se puede observar de que ACO no solo se utiliza para resolver el TSP si no otros varios, al utilizar las GPUs se resolvió más rápido el problema, utilizando dos modelos de trabajo en las GPU's el modelo de isla y el modelo Maestro/Esclavo. En este caso en particular se obtuvieron los resultados de una

manera más rápida utilizando el modelo de isla, pero al utilizar los modelos Maestro/Esclavo, se aceleró más el algoritmo con instancias de gran tamaño, teniendo como conclusión de su investigación que si se requiere acelerar el algoritmo con instancias de un tamaño pequeño en el problema de QAP es mejor utilizar el modelo isla, pero si se trabaja con instancias de gran tamaño es mejor utilizar el modelo Maestro/Esclavo.

You (2009) en su investigación de ACO encontró una característica para poder paralelizar el algoritmo, la ubicación de las ciudades como la ubicación de las hormigas se calculan por medio de operaciones probabilística. El movimiento de las hormigas dentro del grafo, se hace de manera individual por cada hormiga siendo la única comunicación que tienen entre ellas el rastro de feromona. La actualización y evaporación de la feromona se realiza después de que la hormiga termina el tour. En CUDA es posible manejar dos tipos de operaciones para ACO: el primero es el de la inicialización y el segundo para controlar los niveles de feromona, teniendo entendido que cada thread es un tour completo dentro de un ciclo, así cuando cada hormiga termina un tour se vuelve a comenzar un tour nuevo (next_tour).

La clave para obtener los mejores resultados es el cuidado en la asignación de la memoria en la GPU, siendo la distribución de la siguiente manera:

1. Para los datos de acceso frecuente para solo una hormiga se recomienda que se guarden las coordenadas de cada cuidad en la memoria compartida.
2. Para el llenado de la lista tabú se recomienda que se guarde el proceso en la memoria global para un acceso más rápido.
3. Para los niveles y distancias totales de los tour, se recomienda que se guarden en la memoria cache ya que se tiene menor latencia que si se guardaran en la memoria global.

En el trabajo de O'Neil (2011) se puede observar que el TSP de manera paralela no solo se soluciona con ACO, el muestra otra alternativa de solución con el algoritmo IHC (Algoritmo para acelerar Intercomunicaciones). Los resultados obtenidos no solo dan soluciones alta calidad sino que también se ejecutan muy rápidamente. Tomando en cuenta que la velocidad a comparación de un procesador de 64 bits de una CPU y una GPU muestra una mejora de 62 veces más rápido, aun teniendo 32 CPU con 8 núcleos cada una. Llegando a la conclusión de que al ejecutar el algoritmo de manera paralela en una GPU la respuesta será más rápida, y no sólo con el algoritmo IHC sino que también se puede probar con el algoritmo ACO es posible resolver el TSP.

9.9.6 Identificación de procesos paralelizables en ACO

Antes de empezar a explicar este apartado, tomando en cuenta la bibliografía actual no todo el código es factible de paralelizar, pero si aquel que contenga dos ciclos for anidados como se muestra en el siguiente análisis.

El segmento de código secuencial que se pasó a paralelo fue el de la función precompute_distance (véase código siguiente), el cual contiene cálculos susceptibles de ser paralelizados por encontrarse con dos ciclos for anidados.

```
void precomputedistancesfunction(void) \{    //funcion
   para precalcular las distancias
    int from, to;
    float dx, dy;
    /* Precalculo de las distancias entre las ciudades
       */
    for (from = 0 ; from $<$ NUM\_CITIES ; from++) \{
        for (to = 0 ; to $<$ NUM\_CITIES ; to++) \{
            dx = abs(cities\_x[from] - cities\_x[to]);
            dy = abs(cities\_y[from] - cities\_y[to]);
            precomputed\_distance[from][to] =
               precomputed\_distance[to][from] = float
               (sqrt((dx*dx)+(dy*dy)));
            pheromone[from][to] = pheromone[to][from] =
                BASE\_PHEROMONE;
        }
    }
    return;
}
```

Función secuencial precompute_distance que incluye la inicialización de pheromone

Los arreglos precompute_distance y pheromone fueron utilizados como base para crear 2 kernel's o núcleos paralelos, siendo el primero, el kernel denominado precompute_distance el cual nos ayudará a calcular la distancia entre los nodos véase la siguiente función:

```
\_\_global\_\_ void kernelPrecomputedistances(float *
   cities\_x, float *cities\_y, float *precomputed\
   _distance, int n) \{   //funcion paralelizada \\
   para precalcular las distancias entra las ciudades.
    float dx, dy, resultado;
        int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;
    int index= i * n + j;

    dx = abs(cities\_x[i] - cities\_x[j]);// Se
       reformateo
    dy = abs(cities\_y[i] - cities\_y[j]);// Se
       reformateo
    resultado = sqrt(float ((dx*dx)+(dy*dy)));
```

```
    precomputed\_distance[index]= resultado;
}
```

Kernel paralelo de la función Precompute_distance

Este kernel recibe un vector flotante de la coordenada de x, otro vector flotante de la coordenada y, y un vector flotante denominado precompute_distance el cual contendrá el resultado del cálculo, y el número de ciudades en una variable entera n.

Los valores i y j son identificadores para un hilo dentro un bloque con su respectivas coordenadas en "x" e "y".

El cálculo se lleva acabo teniendo dos valores flotantes denominados dx y dy, los cuales contienen un valor absoluto del punto inicial y el punto final de dichas coordenadas, estos valores se elevan al cuadrado, se suman y se saca la raíz cuadrada la cual representa la distancia que hay entre los dos puntos guardando este valor en la variable resultado.

El valor de index es donde se van a posicionar el vector en la memoria de la GPU, siendo este el que recorrerá todos los espacios del vector, eliminando los ciclos for anidados ya que estas operaciones se ejecutan de manera paralela resolviéndolas más rápido.

El segundo kernel (kernelPheromone) queda de la siguiente manera:

```
\_\_global\_\_ void kernelPheromone(double *pheromone,
   int n) \{    //funcion paralelizada para el calculo
   de la cantidad de la feromona
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;
    int index= i * n + j;

    pheromone[index] = BASE\_PHEROMONE;
}
```

Kernel paralelo de la función Pheromone

Este kernel recibe un vector doble denominado pheromone y el número de ciudades en una variable entera n.

Los valores i y j son identificadores para un hilo dentro un bloque con su respectivas coordenadas en "x" e "y".

El valor de index indica donde se van a posicionar el vector denominado pheromone en la memoria de la GPU, siendo éste el que recorrerá todos los espacio del vector, eliminando los ciclos for anidados ya que estas operaciones se ejecutan de manera paralela resolviéndose más rápido.

El resultado de dicho cálculo se guarda en el vector BASE_PHEROMONE.

Estos dos kernel's se utilizan para simplificar el cálculo de las matrices bidimensionales convirtiéndolas en matrices unidimensionales o vectores.

9.10 ESPECIFICACIÓN DE LA INTERFAZ EN TÉRMINOS DE PATRONES DE INTERACCIÓN

A partir de la información obtenida en el análisis anterior y de las clasificaciones de patrones de interacción de Van Welie (Welie and Trætteberg, 2000) y de Muñoz (Muñoz et al. 2004) se identificaron 2 patrones de interacción donde se aplican prácticas exitosa en el diseño de la interfaz de usuario del simulador de señales.

9.11 ESPECIFICACIÓN DE LA INTERFAZ EN TÉRMINOS DE PATRONES DE SOFTWARE

A partir de la información obtenida en la etapa de análisis (sección 3.1) y de la clasificación de patrones de diseño propuesto por (Gama et al.1995) se identificaron tres patrones para aplicar las buenas prácticas para el diseño del estado, la vista y los servicios que ofrece en un momento de la interacción el ambiente de simulación de señales.

9.12 EJERCICIOS RESUELTOS

Por favor describa trabajos relacionados al tema Incluir el otro Kernel

A pesar que el paradigma de patrones es reciente en la literatura del diseño de AMS existe cierto número de trabajos que preconizan su diseño en base a cierto tipo de patrones. Los tipos de patrones más frecuentes tal como lo muestra la siguiente tabla, son los patrones de interacción y los patrones de software.

Tabla 3 Comparación de trabajos de ambientes de modelado y simulación en base en el paradigma de patrones

Trabajos/Patrones de diseño de	(Kreutzer, 1996)	(Blilie, 2002)	(Sanz et al. 2003)	(Ekström 2001)	(Boyko et al. 2002)	(Rodriguez et al. 2004)	InOutSpec
Interfaz					*		*
Software	*		*	*			*

El trabajo aquí propuesto titulado In&OutSpec (Especificación de la parte interna y externa de una aplicación interactiva) aparece en la tabla 3 como el único que propone especificar la interfaz de usuario y la funcionalidad de un AMS utilizando respectivamente los patrones de interacción y los patrones de diseño de software.

9.13 EJERCICIOS A RESOLVER

La autoevaluación sobre la implementación de algoritmos en paralelo usando GPUs se presenta través de diversos tipos de ejercicios y pruebas, a saber:

1. Implemente las otras funciones identificadas como susceptibles de ser paralelizadas.
   (a) ¿Se pueden realmente paralelizar todas las funciones?
   (b) ¿Cómo son los tiempos de ejecución secuencial con respecto a los tiempos de ejecución del algoritmo en paralelo? Pruebe con varias instancias del tsplib.
2. Busque la implementación del código secuencial para resolver el Juego de las Torres de Hanói propuesto por Jones (2009), y realice los siguientes ejercicios:
   (a) a) La solución propuesta es sólo para tres discos, intente generalizar esta solución a 4 discos y a n discos, ¿Qué elementos es necesario modificar?¿El programa arroja soluciones válidas?
   (b) b) Identifique el código que es susceptible a ser paralelizado y paralelícelo empleando la metodología propuesta en este capítulo,
   (c) c) Compare el tiempo de ejecución secuencial contra el paralelo
   (d) d) Realice un análisis de sensibilidad, comparando los resultados obtenidos de la implementación secuencial contra la paralela cambiando el tamaño de la población.
3. Anímese e identifique un código secuencial con ciclos for anidados, cuyos cálculos sean independientes y paralelice el código.
   (a) Compare los resultados secuencial contra paralelo
   (b) Cambie las condiciones de terminación del ciclo y analice los resultados obtenidos.
4. De acuerdo a sus resultados indique cuándo conviene paralelizar los algoritmos secuenciales y cuando no.

9.14 CONCLUSIONES

El presente trabajo propone una metodología de análisis y diseño para la reutilización y el fácil mantenimiento de la parte interna y externa de una Ambiente de Modelado y Simulación (AMS). A nivel de análisis establece una relación de los requerimientos de la interfaz gráfica y la funcionalidad de un AMS. A nivel de diseño se propone una especificación más cerca al código en términos de patrones de diseño y un especificación más cercana a los requerimientos del usuario en términos de de patrones de interacción. Uno de los objetivos alcanzados con los patrones es facilitar la fluidez de la

comunicación entre las partes, donde cada una tiene su propia experiencia y conocimientos. Los patrones de diseño permiten especificar el código de la funcionalidad de una aplicación dando soluciones al diseñador para reutilizar y mantener fácilmente el código. Por su parte los patrones interacción permiten especificar las experiencias exitosas en el diseño de la interfaz grafica tomando en cuenta las necesidades del usuario y los componentes gráficos. El modelo aquí propuesto puede ser extendido a aplicar los patrones de software par la generación de código de un AMS. Esto con el fin de llevar a cabo el prototipaje rápido y realizar así pruebas en conjunto con el usuario.

9.15 BIBLIOGRAFIA

Blilie, C.: Computing in Science & Engineering Patterns in scientific software: an introduction (2002)

Ekström, U.: Design Patterns for Simulations in Erlang/OTP . PhD Thesis, Uppsala University,Sweden (2001)

Gamma, E., Helm, R., Johnson, R., Vlissides, J. and Booch, G.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing (1995)

Kreutzer, Wolfgang: Some Patterns for Building Discrete Event Models and Simulators. Urbana Illinois., USA (1996)

Muńoz, A.J., Reyes, G.C.A. and Pérez, G.H.G.: Designing Direct Manipulation User Interfaces by Using Interaction Patterns. WSEAS Transactions on Computer 3 (2004)

Rodriguez, G.G., Muńoz, A.J. and Fernandez, d.B.R.: Scientific Software Design Through Scientific Computing Patterns. 4th IASTED International Conference on Modeling, Simulation, and Optimization -MSO04- (2004) 493-498.

Sanz, R. and Zalewski, J.: Pattern-Based Control Systems Engineering in IEEE Control System Vol. 23, No. 3, (2003) 43-60.

Peter Forbrig, Q.L.B.U.&.J.V., (ed.): User Interface Desing Patterns for Interactive Modeling in Demography and Biostatics. Rostock, Germany: Springer-Verlag. (2002)

Welie, van Martijn and Trætteberg, Hallvard. Interaction Patterns in User Interfaces, 7th. Pattern Languages of Programs Conference; Allerton Park Monticello, Illinois, USA. (2000)

Hernández-Aguilar, J.A.; Zavala, J.C.; and Vakhnia, N. (2011). Explotación de las Capacidades de Procesamiento en Paralelo de Computadoras Multiprocesador y de las Unidades Gráficas de Procesamiento “GPUs” mediante

PythonOpenCL en ambientes Linux in Construcción de Soluciones a Problemas de Administración de Operaciones e Informática Administrativa. Vol. 1(1) pp. 25-39, CA INVOP, pp. 25-39.

Pospichal, P., Jaros, J., & Schwarz, J. (2010). Parallel Genetic Algorithm on the CUDA Architecture.

Curriculums de los Autóres

Dra. Casali Ana. Es Doctora en Tecnologías de la Información de la Universidad de Girona UdG (Girona, España), obtuvo el Diploma de Estudios Avanzados (equivalente a un Magister) en Tecnologías de la Información en la misma universidad española y Licenciada en Matemática por la Universidad Nacional de Rosario, UNR (Rosario, Argentina). Actualmente es Directora del Departamento de Ciencias de la Computación de la Facultad de Cs. Exactas, Ingeniería y Agrimensura, FCEIA-UNR; Profesora responsable de la Cátedras Introducción a la Inteligencia Artificial, e Ingeniería del Conocimiento de la FCEIA-UNR; e investigadora del Centro Internacional Franco Argentino de Ciencias de la Información y de Sistema, CIFASIS. Ha dictado numerosos Cursos de Posgrado. Como investigadora ha dirigido y participado en proyectos nacionales e internacionales en el área de Agentes y Sistemas Inteligentes y aplicaciones a la educación. Es autora de un libro y tiene numerosas publicaciones en su línea de investigación en los últimos años. Ha participado en comisiones ejecutivas y evaluadoras de distintos Congresos Nacionales, Internacionales y Revistas. Ha dictado diferentes Conferencias sobre temas vinculados a su área de investigación. Además, ha sido Miembro de Jurados de Tesis de Doctorado.

Dr. Hernandez Jose Alberto. Obtuvo el Doctorado Directo en Ingeniería y Ciencias Aplicadas (Especialidad en Ingeniería Eléctrica) por la Universidad Autónoma del Estado de Morelos (UAEM) CIICAp en el año 2008. Es profesor de tiempo completo en la UAEM en materias de Licenciatura, Maestría en las áreas de Programación, Matemáticas, Base de Datos, Miembro de diversos comités de diseño y rediseño curricular para programas educativos de Licenciaturas y Posgrado. Responsable de proyectos de Investigación de la PROMEP. Miembro titular del cuerpo académico En CONSOLIDACIÓN (PROMEP): "•Investigación de Operaciones e Informática". Áreas de Investigación en las que ha publicado: Inteligencia Artificial, Minería de Datos, Ingeniería de Software, Uso de las TICs en la Educación. Cuenta con más de 40 publicaciones en congresos y revistas a nacional e internacional, 10 Capítulos de libros publicados en editoriales e instituciones como IGI-Global, InTech, Nova Publisher.

Dr. Ornelas Zapata Francisco Javier. Obtuvo el Doctorado en Ciencias de la Computación por parte de la Universidad Autónoma de Aguascalientes (UAA) en el año 2010. Es profesor de medio tiempo interino en el Departamento de Ciencias de la Computación en la UAA a nivel licenciatura así como profesor a nivel medio superior en áreas de Programación, Inteligencia Artificial y Fundamentos y Teorías Computacionales. Áreas de Investigación en las que ha publicado: Inteligencia Artificial, Minería de Datos. Cuenta con publicaciones en congresos y revistas a nacional e internacionales y con varios Capítulos de libros.

Dr. Ponce Gallegos Julio Cesar. Obtuvo el Doctorado en Ciencias de la Computación por parte de la Universidad Autónoma de Aguascalientes (UAA) en el año 2010. Es profesor de Tiempo Completo en el Departamen-

to de Ciencias de la Computación en la UAA, en materias de Licenciatura, Maestría y Doctorado en las áreas de Programación, Inteligencia Artificial y Fundamentos y Teorías Computacionales, Miembro de diversos comités de diseño curricular para programas educativos de Licenciaturas y Posgrado. Responsable de proyectos de Investigación de la UAA y PROMEP. Miembro titular del cuerpo académico CONSOLIDADO (PROMEP): "Sistemas Inteligentes". Áreas de Investigación en las que ha publicado: Inteligencia Artificial, Minería de Datos, Ingeniería de Software, Uso de las TICs en la Educación. Cuenta con más de 50 publicaciones en congresos y revistas a nacional e internacional, 11 Capítulos de libros y 2 Libros publicados en editoriales e instituciones como IGI-Global, InTech, Nova Publisher, Textos Universitarios (UAA), SMIA.

Dra. Quezada Aguilera Fatima Sayuri. Obtuvo el Doctorado en Ciencias de la Computación por parte de la Universidad Autónoma de Aguascalientes (UAA) en el año 2010. Es profesora de Asignatura en el Departamento de Ciencias de la Computación en la UAA, en materias de Licenciatura, en las áreas de Programación, Inteligencia Artificial y Fundamentos y Teorías Computacionales. Áreas de Investigación en las que ha publicado: Inteligencia Artificial, Minería de Datos, Ingeniería de Software. Cuenta con publicaciones en congresos y revistas a nacional e internacionales, Capítulos de libros y 1 Libro.

Dra. Scheihing Eliana. Doctora en Estadística de la Universidad Católica de Lovaina, Bélgica e Ingeniero Civil Matemático de la Universidad de Chile. Actualmente Académica del Instituto de Informática de la Universidad Austral de Chile y Directora Alterna del Centro de Investigación, Desarrollo e Innovación Kelluwen (http://www.kelluwen.cl). Profesora Responsable de la asignatura Inteligencia Artificial para la carrera de Ingeniería Civil en Informática de la Universidad Austral de Chile. Autora de diversas publicaciones en revistas y congresos en las áreas de TIC en Educación, Minería de Datos Educacionales y Sistemas Recomendadores. Directora del proyecto FONDEF D08i-1074 Kelluwen y actualmente de un proyecto de financiamiento regional (FIC-R) sobre Didáctica 2.0 para el Emprendizaje.

M.C. Silva Sprock Antonio. Estudiante del doctorado en Ciencias de la Computación de la Universidad Central de Venezuela. Magister en Ingeniería del Conocimiento por la Universidad Politécnica de Madrid. Docente investigador de la Escuela de Computación de la Universidad Central de Venezuela. Investigador acreditado categoría A del Programa de Estímulo a la Investigación del ONCTI del MPPCTI. Trabaja en Base de Datos, Sistemas de Información, Objetos de Aprendizaje y Tecnologías Educativas. Desde 2012 coordina, regionalmente para Venezuela, el proyecto "LATIn" (iniciativa para la creación de libros abiertos en Latinoamérica), ALFA auspiciado por la Unión Europea. Coordina el proyecto "Desarrollo de una Herramienta para Crear Recursos Educativos Abiertos para la Enseñanza en Lenguas Indígenas", PEII auspiciado por el FONACIT. Participa en el Proyecto "Generación de herramientas tecnológicas para la producción distribución y almacenamiento de OACA para personas con discapacidad visual, auditiva y

cognitiva", PEII auspiciado por el FONACIT. Es autor de múltiples trabajos publicados en revistas nacionales e internacionales.

Dra. Torres Soto Aurora. Obtuvo el Doctorado en Ciencias de la Computación por parte de la Universidad Autónoma de Aguascalientes (UAA) en el año 2010. Es profesor de Tiempo Completo en el Departamento de Ciencias de la Computación en la UAA, donde imparte materias de Pregrado y Posgrado en las áreas de Programación, Inteligencia Artificial y Fundamentos y Teorías Computacionales. Actualmente es coordinadora de la Ingeniería en Computación Inteligente del Centro de Ciencias Básicas. Miembro titular del cuerpo académico CONSOLIDADO (PROMEP): "Sistemas Inteligentes". Áreas de Investigación en las que ha publicado: Inteligencia Artificial, Metaheurísticas, Diseño automático de Circuitos Analógicos. Cuenta con publicaciones en congresos y revistas a nivel nacional e internacional y con varios Capítulos de libros en editoriales como Global e InTech.

Dra. María Dolores Torres Soto. Obtuvo el Doctorado en Ciencias de la Computación por parte de la Universidad Autónoma de Aguascalientes (UAA) en el año 2010. Es profesor de Tiempo Completo en el Departamento de Sistemas de Información en la UAA, donde imparte materias de Pregrado y Posgrado en las áreas de Programación, Bases de Datos y Sistemas y Tecnologías de la Información. Actualmente es coordinadora de la Maestría en Informática y Tecnologías Computacionales del Centro de Ciencias Básicas. Miembro titular del cuerpo académico CONSOLIDADO (PROMEP): "Sistemas Inteligentes". Áreas de Investigación en las que ha publicado: Inteligencia Artificial, Metaheurísticas, Diseño automático de Circuitos Analógicos. Cuenta con publicaciones en congresos y revistas a nivel nacional e internacional y con varios Capítulos de libros en editoriales como Global e InTech.

Dr. Túpac Valdivia Yván Jesús. Profesional del área de Ingeniería Eléctrica, con Doctorado en Ingeniería Eléctrica (PUC-Rio, 2005), Maestría en Ingeniería Eléctrica (PUC-Rio 2000), y pregrado en Ingeniería Electrónica (UNSA, 1995), con experiencia en Ciencia de la Computación, énfasis en Soft Computing (Inteligencia Artificial: redes neuronales, algoritmos evolutivos, lógica difusa, aprendizaje por refuerzo, procesamiento de imágenes, sistemas clasificadores). Su actuación principal es en proyectos de Investigación y Desarrollo en Ingeniería de Petróleo, Energía Eléctrica, Evaluación de Proyectos, Análisis de Riesgo, Control y Automatización de Procesos, Sistemas de redes de Comunicación, Computación Distribuida, Sistemas Electrónicos y Digitales, Análisis y Procesamiento de Imágenes. Miembro del IEEE/CIS (Peru Chapter vicechair 20913-2014), IEEE/CS, SPE (Society of Petroleum Engineers), SPC (Sociedad Peruana de Computación). CV online en www.ucsp.edu.pe/˜ytupac y http://lattes.cnpq.br/4210156169619645

Edición: Marzo de 2014.
Este texto forma parte de la Iniciativa Latinoamericana de Libros de Texto abiertos (LATIn), proyecto financiado por la Unión Europea en el marco de su Programa ALFA III EuropeAid.