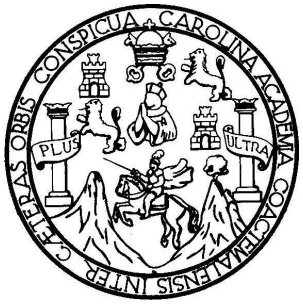
Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas

Aplicación de la Inteligencia Artificial en la Robótica: Implementación de IA en un robot, para que sea capaz de resolver laberintos.

Drixdel Gabriel Ramos Escobar
Asesorado por Inga. Floriza Ávila

Guatemala, septiembre de 2005.

UNIVERSIDAD DE SAN CAROLOS DE GUATEMALA



FACULTAD DE INGENIERÍA

APLICACIÓN DE LA INTELIGENCIA ARTIFICIAL EN LA ROBÓTICA: IMPLEMENTACIÓN DE IA EN UN ROBOT, PARA QUE SEA CAPAZ DE RESOLVER LABERINTOS.

TRABAJO DE GRADUACIÓN

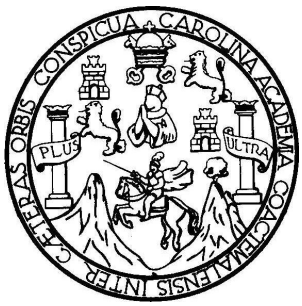
PRESENTADO A JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

DRIXDEL GABRIEL RAMOS ESCOBAR
ASESORADO POR INGA. FLORIZA AVILA

AL CONFERÍRSELE EL TÍTULO DE
INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, SEPTIEMBRE DE 2005.

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA



NOMINA DE JUNTA DIRECTIVA

Decano	Ing. Murphy Olympo Paiz Recinos
Vocal I	
Vocal II	Lic. Amahán Sánchez Álvarez
Vocal III	Ing. Julio David Galicia Celada
Vocal IV	Br. Kennet Issur Estrada Ruiz
Vocal V	Br. Eliza Yazminda Vides Leiva
Secretaria	Inga. Marcia Ivonne Véliz Vargas

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

Decano	Ing. Sydney Alexander Samuels Milson.
Examinador	Ing. Ricardo Morales Prado.
Examinadora	Inga. Elizabeth Domínguez Alvarado.
Examinadora	Inga. Virginia Victoria Tala Ayerdi.
Secretario	Ing. Carlos Humberto Pérez Rodríguez.

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

Aplicación de la Inteligencia Artificial en la Robótica: Implementación de IA en un robot, para que sea capaz de resolver laberintos

Tema que me fuera asignado por la Dirección de la Escuela de Ciencias y Sistemas, con fecha enero de 2004.

Drixdel Gabriel Ramos Escobar.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

TABLAS

GLOSARIO

Automatización Funcionamiento automático de una máquina, o conjunto de máquinas, encaminado a un fin único, lo cual permite realizar con poca intervención del hombre, una serie de trabajos industriales o administrativos o de investigación.

Autonomía Facultad de gobernarse por sus propias leyes.

Bit Unidad de medida de la cantidad de información, equivalente a la elección de una entre dos posibilidades del sistema.

Chip Diminuto trozo de cristal semiconductor, en forma de cubo, en el que se han formado diodos, transistores u otros componentes que, interconectados, constituyen un circuito integrado funcional.

Corriente eléctrica Paso de la electricidad entre dos puntos de diferente potencial, a través de un conductor. Puede ser continua, cuando fluye siempre en la misma dirección, y alterna cuando cambia periódicamente de dirección.

Efector Dispositivo que determina la producción de alguna acción.

Hardware	Conjunto de elementos materiales de un ordenador electrónico.
Humanoide	Ser con ciertos rasgos de hombre.
Informática	Conjunto de conocimientos científicos y técnicos que se ocupan del tratamiento de la información por medio de ordenadores electrónicos.
Infrarrojo	Radiación del espectro luminoso que tiene mayor longitud de onda y se encuentra más allá del rojo visible; se caracteriza por sus efectos térmicos, pero no luminosos ni químicos.
Microcontrolador	Circuito electrónico integrado que contiene muchas de las cualidades de una computadora de escritorio. Están diseñados para aplicaciones de control de máquinas.
Robot	Máquina electrónica que puede ejecutar automáticamente distintas operaciones o movimientos.
Robótica	Parte de la ingeniería que se ocupa de la aplicación de la informática al diseño y uso de máquinas que sustituyan a las personas en la realización de diferentes tareas o funciones.
Sensor	Dispositivo que sirve para determinar los valores de una dimensión física, tal como temperatura, sonido o intensidad de luz.

Servomotor	Es un motor eléctrico con capacidad de ser controlado, en velocidad y/o posición.
Sistema	Conjunto de cosas o partes coordinadas según una ley, o que, ordenadamente relacionadas entre sí, contribuyen a determinado objeto o función.
Software	Conjunto de programas de ordenador y técnicas informáticas.
Telémetro	Aparato para medir la distancia a que uno se encuentra de un objeto y que consiste esencialmente en un anteojo provisto de un micrómetro.
Telequérica	Consiste en la utilización de un manipulador remoto controlado por un ser humano. El teleoperador puede permanecer en un lugar seguro; no obstante, mirando a través de una ventana de cristal plomado o mediante televisión en circuito cerrado.
Voltaje	Diferencia de potencial eléctrico entre los extremos de un conductor, expresada en voltios.

RESUMEN

El desarrollo de este trabajo de investigación se puede dividir en dos partes, una parte teórica y una parte práctica.

Teoría

El primer capítulo es una introducción a la Robótica, se hace una clasificación de robots, dependiendo de sus características se clasifican en Robots Industriales y Robots de Investigación.

En el segundo capítulo se introduce el concepto de Agentes Inteligentes. También se habla acerca de las diferentes características que tienen los ambientes.

En el tercer capítulo se analizan las diferentes estrategias de búsqueda que ofrece el campo de la Inteligencia Artificial, las cuales se pueden clasificar en Estrategias de Búsqueda con Información y Estrategias de Búsqueda sin Información.

Práctica

En el cuarto capítulo se desarrolla un programa de computadora para simular ambientes de laberintos y probar la estrategia de búsqueda seleccionada.

El quinto capítulo se trata de la construcción del robot *Samy-Bot*(nombre con el que se bautizó al robot), en este capítulo se muestra paso a paso cómo se fue construyendo a *Samy-Bot*, se habla en detalle sobre los componentes principales de *Samy-Bot*, como lo son el microcontrolador *BASIC Stamp* 2, la Plaqueta de Educación y los Servomotores que mueven a *Samy-Bot*.

En el sexto capítulo se explica en detalle cómo fue programado el microcontrolador de *Samy-Bot*, se hace un análisis de cada módulo que componen el *software* del robot. Los principales módulos del *software* que controlan a *Samy-Bot* son: Módulo de Navegación, Módulo de Visión y Módulo de Control.

OBJETIVOS

General

Implementar Inteligencia Artificial en un robot tipo vehículo, para que sea capaz de resolver laberintos y aprender de su experiencia con el ambiente que le rodea.

Específicos

1. Mostrar los pasos necesarios para ensamblar la estructura mecánica del robot.

2. Mostrar los pasos necesarios para ensamblar la electrónica del robot.

3. Mostrar los beneficios de la programación de los Microcontroladores.

4. Desarrollar un pequeño Sistema Operativo que administre los dispositivos del Microcontrolador.

5. Implementar Algoritmos de IA, para que el robot pueda navegar de forma inteligente en el laberinto.

INTRODUCCIÓN

En este trabajo de investigación se propone un panorama de la aplicación de la Inteligencia Artificial en la Robótica. La aplicación que se pretende desarrollar a lo largo del contenido de este trabajo es: la construcción y programación de un robot, que dotado de inteligencia artificial sea capaz de resolver laberintos. El sólo hecho de construir un robot es ya de por si sumamente interesante, y lo será mucho más si se le agregan características tales como Inteligencia Artificial para resolver problemas y aprender.

La mayor parte del conocimiento que se utilizará para la construcción del robot procede de investigaciones que han realizado personas de otros países, tales como Argentina, Chile, España, Estados Unidos, Rusia y México entre otros. Así como de la colaboración de 3 comunidades en Internet dedicadas a la robótica.

Para el análisis y planteamiento se utilizarán conceptos aprendidos en los cursos de Teoría de sistemas, Sistemas Operativos, Física, Matemática, Arquitectura de computadores, Programación y Análisis y Diseño de sistemas.

El robot planteado es clasificado como robot tipo vehículo, este tipo de robot es una evolución lógica del concepto tan usual de medio de transporte. Se caracterizan porque disponen de ruedas u orugas que les permiten moverse por un entorno, igual que lo hace un tanque, un auto o una excavadora. Realmente se pueden definir como "Vehículos con inteligencia".

El contenido del trabajo se puede dividir en dos partes, una parte teórica y una parte práctica. La parte teórica está comprendida por los capítulos primero, segundo y tercero, en ellos se desarrolla la teoría necesaria para poder comprender la práctica.

La parte práctica está comprendida por los capítulos cuarto, quinto y sexto, en ellos se desarrollará paso a paso el software que se utiliza para controlar al robot.

Para una mejor ilustración del trabajo realizado, en cada capítulo se incluye fotografías de cada acción realizada. Además en cada capítulo, excepto el primero, se incluyen una introducción y un resumen con el propósito de que el lector pueda enterarse de forma rápida acerca del contenido de cada capítulo.

1. INTRODUCCIÓN A LA ROBÓTICA

1.1. Robótica

Desde que en 1,917 el escritor Karel Capek usara el término robot para referirse a unas máquinas en forma de humanoide, han tenido que aparecer numerosos avances tecnológicos y pasar casi un siglo, para que el autor de este proyecto haya podido realizar un robot en un espacio de tiempo relativamente corto y con un presupuesto mucho más reducido.

La robótica se define como una ciencia aplicada que surge de la combinación de la tecnología de las máquinas-herramienta y de la informática. Una máquina herramienta se define como una máquina que efectúa cualquier trabajo manual, y la informática como la ciencia del tratamiento automático y racional de la información. Uniendo ambos conceptos la robótica surge al automatizar de manera racional las máquinas-herramienta, es decir al permitir que un programa informático controle las operaciones que antes realizaba un operario. Ligado a la robótica aparece el robot, si lo primero es la ciencia lo segundo es el objeto. Se considera la aparición del primer robot, en su concepción moderna, como la unión del control numérico y de la telequérica. El control numérico fue una de las primeras formas de la informática y la telequérica es la ciencia que estudia los manipuladores controlados a distancia por un ser humano, o teleoperadores, que se pueden considerar como una máquina-herramienta avanzada. A partir de aquí la evolución de la robótica ha estado íntimamente relacionada con el desarrollo de la tecnología eléctrica e informática, todo en la aparición de nuevos y mejores sistemas de control.

En 1,959 se introdujo el primer robot comercial por la empresa Corporación Planeta (*Planet Corporation, por su nombre en ingles)*, estaba controlado por interruptores de fin de carrera y levas. En 1,971 la Universidad de *Stanford* desarrolló el Brazo Standford(*Stanford Arm*, por su nombre en ingles*)*, un pequeño brazo robot de accionamiento eléctrico. En 1,981 la Universidad de *Carnegie-Mellon* diseñó un robot que utilizaba motores eléctricos situados en las articulaciones del manipulador sin las transmisiones mecánicas habituales. Las primeras aplicaciones prácticas de los robots se encuentran en la industria y sobre todo en la del motor. En 1,961 la *Ford Motor Company* utilizó uno para controlar una máquina de fundición en troquel y en 1,974 *Kawasaki* instaló otro para soldar las estructuras de las motocicletas.

En paralelo, los lenguajes de programación de robots fueron mejorando. Los primeros lenguajes de programación permitían el desarrollo de programas de control utilizando gráficos interactivos en un ordenador personal que luego se podían cargar en el robot.

Al ser la industria la que adquirió la mayoría de ellos, una de las primeras definiciones del término robot la proporcionó la Asociación de Robótica Industrial (RIA, por su sigla en ingles, de ahora en adelante se hará referencia a ella por su sigla) y vino a decir:

> Un robot industrial es un manipulador multifuncional reprogramable diseñado para desplazar materiales, piezas herramientas o dispositivos especiales mediante movimientos programados y variables, para la ejecución de una diversidad de tareas.

Bajo el punto de vista anterior es fácil comprender la definición dada por la RIA, aunque actualmente se podría decir que se ha quedado pequeña, ya que actualmente existen muchas más aplicaciones que no responden a las características anteriores y no por ello dejan de ser robots. Además, se está produciendo un cambio en la actitud de la sociedad respecto a la robótica, todo por la sucesiva aparición de aplicaciones que, de no haber sido por ella, no podrían haberse llevado a cabo.

Al principio existió un sentimiento generalizado de oposición al robot, se consideraban más enemigos que amigos, siendo la razón principal el temor a ser sustituido. Evidentemente esta opinión tiene su fundamento en un momento puntual de la historia y en todas aquellas personas que perdieron sus puestos de trabajo al introducir los robots en las fábricas. Pero ahora se puede decir que este hecho permitió la evolución de los trabajos, ya que se eliminaron las tareas repetitivas que no permitían un enriquecimiento mental, y se crearon nuevos puestos dentro de las empresas que permitían una participación mayor del empleado, siendo su trabajo más humano que repetitivo. No obstante, no se quiere entrar a discutir en este proyecto si la robótica ha sido perjudicial o no desde un punto de vista social.

Otro concepto era la forma que tenía que tener un robot. Al principio la mayoría opinaba que un robot tendía un aspecto humanoide, con pies, manos, cabeza, más o menos inteligente y que estaba al servicio de su amo. Si hubiese que buscar un motivo de esto se podría decir que fueron las películas, libros series de TV, etc., las que emplearon la figura del humano y le pusieron un cerebro electrónico. Por ejemplo, se podrían citar las numerosas obras del escritor Isaac Asimov que tenían como protagonistas a robots.

Actualmente, la robótica sigue avanzando con paso firme, pero con un ritmo menor que el previsto por los expertos en los años 80. Además, la sociedad ha asimilado el papel de estos, considerándolos como herramientas, sin una forma típica y con una inteligencia mínima en comparación con la del ser humano. La evolución de la tecnología ha permitido desarrollar nuevos materiales, sensores y chips que han facilitado y abaratado considerablemente la construcción de robots. Sin embargo todavía no se ha producido la explosión que ocurrió con las computadoras personales, que pasaron de ser un privilegio de empresas multinacionales y universidades, a ser un electrodoméstico más dentro de una casa. Es más, es normal que al adquirir una computadora se le permita al comprador seleccionar los diferentes componentes que lo van a integrar. Y todo se debe al acercamiento que sufrió la informática al público no especializado, todo cuando su uso se volvió más sencillo, barato y con aplicaciones más potentes y flexibles que las tradicionales. Por ejemplo, se puede destacar el paulatino desuso de las máquinas de escribir a favor de las aplicaciones de procesamiento de textos.

La reflexión anterior induce a pensar en una posible revolución de la robótica, pasando a ser un elemento más dentro del hogar, de la empresa y de la sociedad en general. Ésta opinión es una apuesta de futuro, avalada por la aparición a lo largo de estos últimos años, de robots y proyectos muy llamativos, que han facilitado los cambios de opinión de las personas hacia los robots. Ya no se ven como los grandes enemigos sino todo lo contrario: son tratados como herramientas amigas, que tratan de ayudar y de facilitar el trabajo. Esto último es un poco exagerado, seguro que más de una vez se ha desesperado por problemas con su computadora, auto, etc. y ha pensado que cada día la sociedad lo tiene más complicado. Pero la prueba está ahí, sin ir más lejos, en 1,997 un pequeño robot móvil aterrizó en Marte permitiendo que millones de espectadores pudiesen apreciar fotografías captadas desde 191 millones de kilómetros de la Tierra. Este pequeño robot llamado *Sojourner*, más conocido como Pionero(*Pathfinder*, nombre de la misión), recordó los primeros pasos de la llegada del hombre a la Luna.

Otro caso parecido fue el robot sumergible que bajó a 3,500 metros de profundidad para introducirse en los restos del *Titanic* con objeto de fotografiarlo. En esta ocasión el robot se llamaba *Jason Junior* y se operaba por control remoto. Los ejemplos anteriores muestran aplicaciones fuera de la industria como tal, y no son las únicas: otros campos como la medicina, aeronáutica y agricultura se beneficiaron con la llegada de robots telecontrolados para realizar operaciones, aviones espías autónomos y los sistemas de recolección y envasado directo.

Aún así, estos robots son excesivamente caros y su función está muy especializada, para lograr su integración total se necesitará hacerlos cada vez más baratos y generales, de manera que sean accesibles, económicamente, a las personas ajenas a toda esta evolución y que lo que quieren es que les resuelva alguna de sus tareas cotidianas. ¿No sería fantástico un robot que limpiase el polvo, que cortara el césped y que además vigilara la casa?. Más de uno invertiría en un artilugio de este tipo. Las dos características anteriores, económico y general, se encuentran en un tipo de robot que poco a poco se está abriendo camino. El lugar dónde hay que ir a buscarlo es el entorno universitario y, más recientemente, en algunas compañías jugueteras. ¿A qué es debido esto?. La razón fundamental se comentó antes: la informática, electrónica y mecánica han pasado a de ser un privilegio a ser materia de estudio en muchos centros.

La tecnología ha evolucionado tanto que los precios se han reducido mucho. Además construir un pequeño robot se ha convertido en un pasatiempo para muchos jóvenes, llenos de imaginación y en muchas ocasiones motivados por los concursos que se realizan por el mundo. Uno de los más famosos es *ROBOCUP*(copa robot), que consiste en un partido de fútbol entre robots. También se han comenzado a celebrar concursos, sobre todo en el ámbito universitario de la ingeniería. Tal vez el más conocido sea el torneo que organiza la Asociación Aeroespacial y Sistemas Electrónicos (AESS, por su sigla en ingles) de la Universidad Politécnica de Cataluña (UPC), *SumoBot*, cuyo objetivo es un combate de sumo entre dos robots. También se ha producido un efecto curioso, la aparición de concursos ha hecho que la aplicación práctica de investigaciones informáticas y electrónicas se lleven a cabo mediante pequeños robots que realizan actividades sorprendentes. Uno de esos robots es COG, desarrollado en el Instituto Tecnológico de *Massachussets* (MIT por su sigla en ingles), que es capaz de jugar al tenis de mesa. En él se ensayan estudios de visión artificial, mecánica, elasticidad en robótica y también actitudes de comportamiento, esto último más ligado a la inteligencia artificial que a la robótica, pero que sirve para darle un carácter más humano.

1.2. Clasificación de los robots

Hoy en día la robótica como disciplina ha crecido mucho y existe una gran cantidad de robots, cada uno de un estilo totalmente diferente al otro. Es muy difícil hacer una clasificación general válida para todos ellos. Con la idea de encuadrar el objeto de este proyecto se va a realizar una clasificación, que no es en absoluto rígida y en la que las fronteras no son totalmente nítidas, pero que servirá para centrar las ideas.

Como primer criterio de clasificación se va a emplear la rentabilidad o provecho comercial que se le puede sacar al robot, que permite hacer la primera división entre **robots industriales y robots de investigación.** Los primeros tienen un tremendo impacto en la industria y son por tanto económicamente rentables. Los segundos no generan beneficios económicos, al contrario, hay que invertir dinero, pero generan conocimiento y hacen que la robótica evolucione.

Los típicos **robots industriales** son brazos mecánicos, muy pesados, con cierto número de grados de libertad y que se caracterizan por la versatilidad: ahora pintan, ahora sueldan, ahora colocan piezas, etc. según el manipulador que se les enganche. Se utilizan en las plantas de montaje, haciendo trabajos repetitivos y que pueden ser peligrosos para los humanos. En la figura 1 se muestra una foto de un robot industrial de la marca Asea, con seis grados de libertad, tres para posicionar el manipulador y tres para orientarlo.

Figura 1. Brazo robot industrial de la marca ASEA.

Por otro lado, están los **robots de investigación**, que intentan aportar algo nuevo a la robótica: nuevos algoritmos más inteligentes, nuevas formas de movimiento, etc. dentro de estos robots se puede hacer otra clasificación, agrupándolos en tres grandes familias: **Robots humaniformes, Robots tipo vehículo y Robots que imitan animales**.

Los **robots humaniformes** no son nuevos. Se lleva hablando de ellos mucho tiempo. De hecho, la propia palabra robot viene del checo y que quiere decir "esclavo". La primera vez que se empezó a hablar de robots fue en una novela de ciencia-ficción, escrita por Karel Capek en 1,917 en la que se describe a unas criaturas mecánicas, robots, creados por un científico con el fin de realizar todo el trabajo duro y pesado para el hombre. Estas criaturas se revelan al final contra su creador. La novela marcó un hito y dio comienzo a una tendencia anti-robot a la que Isaac Asimov bautizó con el nombre de "Complejo de *Frankenstein*", queriendo indicar el miedo que existe a crear criaturas artificiales que puedan llegar a revelarse contra sus creadores.

Isaac Asimov, el creador del término **robótica** como ciencia de los robots, escribió infinidad de cuentos cortos y novelas sobre el tema de los robots humaniformes, pero desde un punto de vista no destructivo, tratando a los robots como máquinas inteligentes que realizan un trabajo muy útil para el hombre. Para garantizar la seguridad definió sus famosas **tres leyes de la robótica**, las cuales se enuncian como sigue:

- Un robot no puede hacer daño a un ser humano, o, por medio de la inacción, permitir que sea lesionado.
- Un robot debe obedecer las órdenes recibidas por los seres humanos, excepto si estas órdenes entrasen en conflicto con la Primera Ley.
- Un robot debe proteger su propia existencia en la medida en que esta protección no sea incompatible con la Primera y la Segunda Ley.

La idea de tener una máquina tan versátil como un ser humano es muy buena. Si todas las herramientas están hechas para que las manejen humanos, ¿no es una buena idea el crear una máquina que maneje esas herramientas en vez de automatizar las propias herramientas? Por ejemplo, una excavadora podría convertirse en un robot autónomo, o se podría sustituir el conductor por un robot humaniforme, sin tener que cambiar la excavadora. Este razonamiento lo deja intuir muy claramente Isaac Asimov en sus obras sobre robots.

Los robots humaniformes de hoy en día, no tienen todavía utilidad comercial, pero son desarrollados por grandes empresas para mostrar su elevada tecnología y conseguir que los medios hablen sobre ellas. Son un reclamo publicitario. El mejor ejemplo está en el robot Asimo, desarrollado por Honda y que se puede ver en las figuras 2 y 3.

Figura 2. Vista lateral del robot Asimo fabricado por Honda.

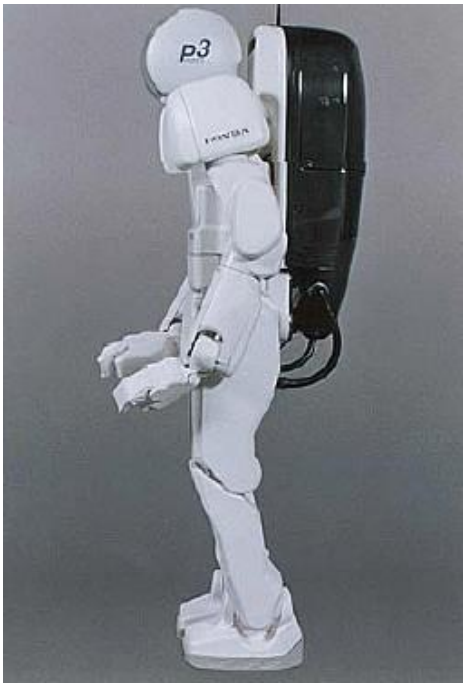
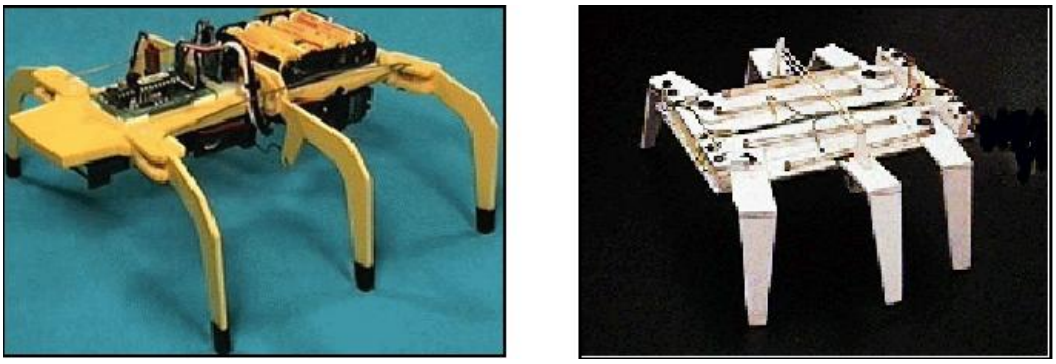


Figura 3. Vista frontal de Asimo.

Con la idea de buscar robots que puedan sustituir a los vehículos con ruedas, de manera que puedan moverse por superficies donde éstos no pueden, ingenieros y científicos están imitando animales, que se encuentran muy bien adaptados al medio en el que viven. Esta idea de imitar a la naturaleza es una fuente de inspiración muy fuerte para los técnicos. El problema es que los animales son tan tremendamente complejos que el hombre sólo puede hacer burdas imitaciones, de momento. Qué más quisieran los ingenieros de la Agencia Espacial de Estados Unidos (NASA, por su sigla en ingles) que crear un vehículo de exploración que fuese un “mono robot” y que pudiese realizar sus mismos movimientos. No existirían obstáculos para la exploración de entornos hostiles fuera de la Tierra.

Una manera de clasificar los animales robots es atendiendo al número de extremidades. Están los robots con patas y los "sin patas". Los **robots con patas** intentan heredar las características de sus animales replicados. Al tener patas son rápidos, aunque no tanto como los vehículos ágiles, pueden moverse por terrenos complicados y tienen la capacidad de poderse impulsar para dar pequeños saltos. Esto es en teoría lo que podrían llegar a hacer. En la práctica son lentos y bastantes torpes, como si fuesen crías recién nacidas. Conseguir movimiento con patas no es una tarea nada trivial. La coordinación entre las diferentes articulaciones es un problema que todavía no está resuelto de una forma elegante. En la figura 4 se muestran dos robots hexápodos. Se puede ver que su estructura es muy simple y dan la impresión de ser lentos y poco ágiles. Distan mucho todavía de moverse como una hormiga real.

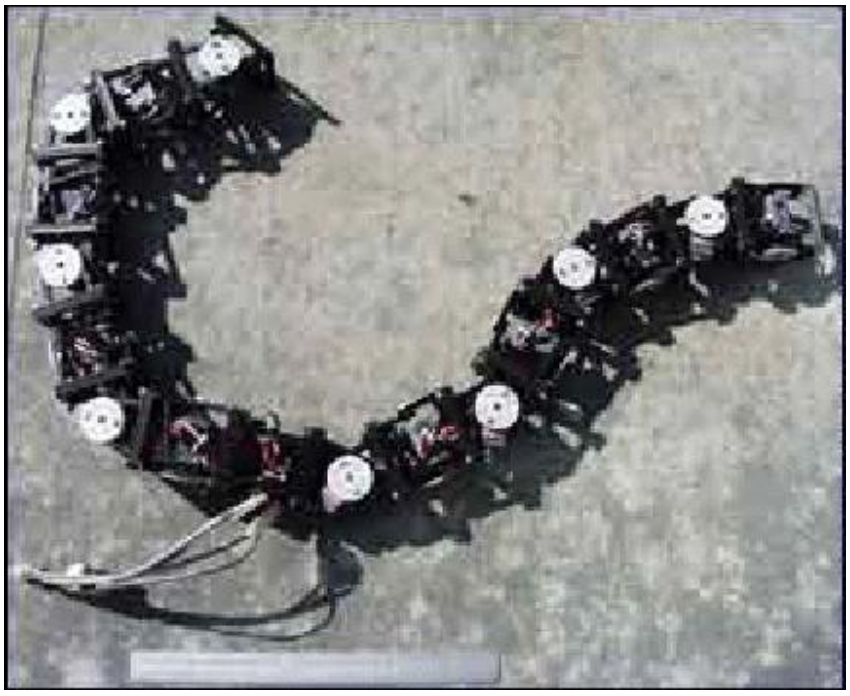
Figura 4. Dos robots hexápodos muy simples.



Los robots sin patas son conocidos como **robots gusano** o **robots serpiente**. Estos no son tan veloces ni tan ágiles como los que tienen patas, pero tienen una serie de ventajas que justifican su investigación. Al tener una forma alargada, pueden penetrar por sitios a los que otro tipo de robot no tiene acceso, como por ejemplo una tubería. Pueden adoptar la forma de la superficie por la que se desplazan, de manera que pueden hacerlo por sitios muy tortuosos. Están compuestos por segmentos iguales, que se enganchan unos a otros, pudiendo conseguirse robots de cualquier longitud.

En la figura 5 se muestra un robot serpiente, diseñado por la *NASA*, en el que se pueden apreciar claramente los diferentes segmentos idénticos que la componen. También se puede ver la facilidad que tienen para adoptar cualquier forma. El desplazamiento se consigue mediante ondas que recorren el robot, para lo cual las articulaciones tienen que estar perfectamente coordinadas.

Figura 5. Robot serpiente diseñado en la NASA.



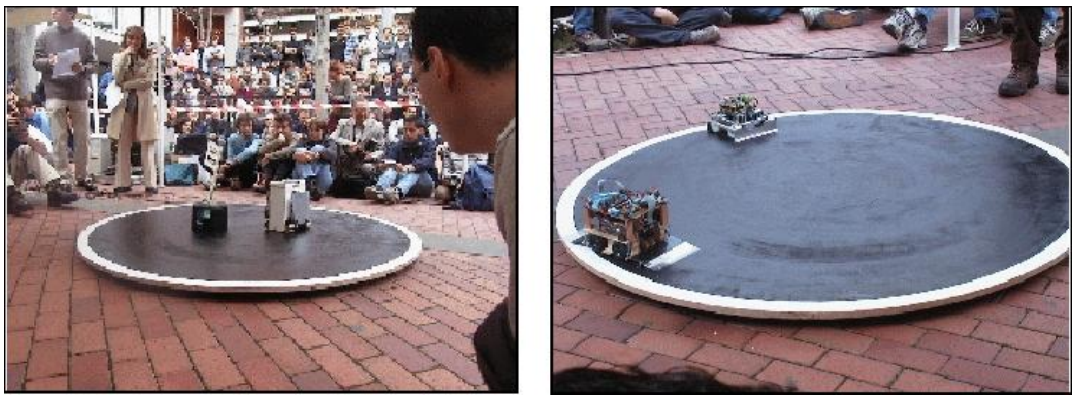
1.3. Robots tipo vehículo

Los **robots tipo vehículo** son una evolución lógica del concepto tan usual de medio de transporte: la mecánica ya existe, luego se utiliza para construir un robot. Se caracterizan porque disponen de ruedas u orugas que les permiten moverse por un entorno, igual que lo hace un tanque, un automóvil o una excavadora. Realmente se pueden definir como "vehículos con inteligencia". Son robots más sencillos de construir que los humaniformes, más baratos y la mecánica está mucho más estudiada.

Actualmente hay dos tipos de vehículos robóticos. Por un lado los modelos creados por estudiantes y aficionados. En el ambiente universitario estos robots se emplean para que los alumnos asimilen los conceptos y aprendan electrónica, mecánica y programación. Existen numerosos concursos en los que se puede participar, como *Robocup*, donde los robots juegan un partido de fútbol, luchadores de sumo, donde se enfrentan en terribles luchas sin tregua, carreras(*microracing*), y rastreadores son otras categorías en las que se compite, como por ejemplo el concurso de *Alcabot* en la Universidad de Alcalá de Henares.

En la figura 6 se muestran escenas de un concurso de luchadores de sumo celebrado en la UPC. Dos robots se sitúan frente a frente en un tatami circular, de color negro, estando en el borde delimitado por un color blanco. El objetivo es echar fuera del tatami al adversario. Para ello los robots deben estar dotados de sensores que le permitan detectar si ellos mismos se salen fuera, analizando el color del suelo, y sensores para detectar al adversario. Además de esto deben disponer de mecanismos para expulsar al enemigo.

Figura 6. Concurso de luchadores de sumo celebrado en la UPC.



En la figura 7 se muestran en la parte de la izquierda el concurso de "robots limpiadores de superficie" también celebrado en la UPC. En este caso el robot lucha contra el reloj y tiene que recoger de un recinto cerrado tantos granos de arroz como pueda. Si además los deposita en la "basura" obtendrá más puntos. En la foto de la derecha se muestra un robot que concursó en Sumo. Se trata de un toro que en cuanto detecta al enemigo comienza a embestir hasta que lo expulsa del tatami.

Figura 7. Concurso de limpiadores de superficie(izquierda), UPC. Robot luchador de sumo Tauro-Viper(derecha).

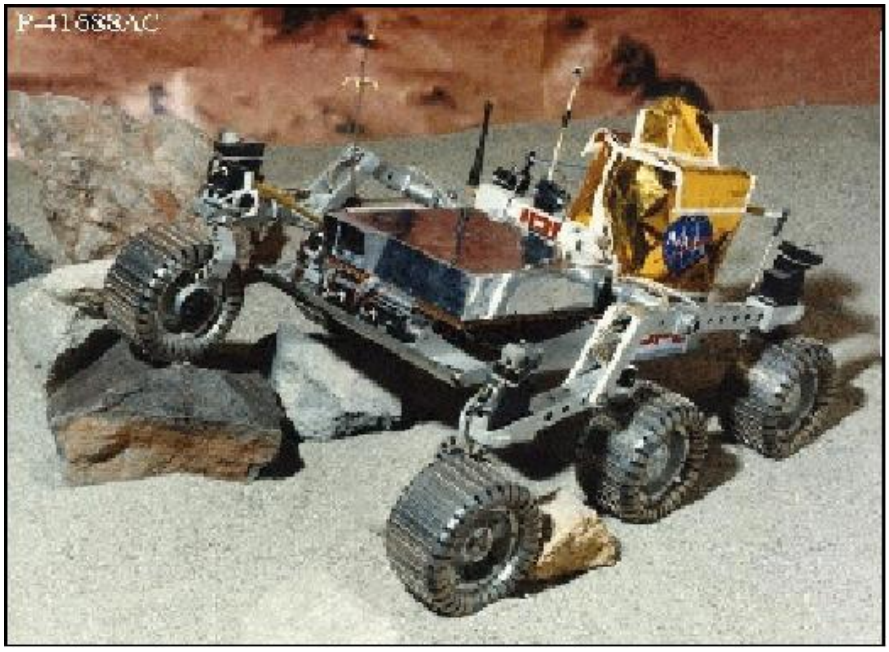
Otro tipo de vehículos más avanzados son los de exploración, que sirven para reconocer un entorno hostil, donde el hombre no puede llegar, y así obtener datos y realizar medidas. La *NASA* está trabajando en este tipo de robots. La sonda ***Mars Pathfinder*** que se envió a Marte llevaba en su interior un vehículo autónomo para recorrer el entorno y tomar muestras de la superficie de Marte. Aunque gran parte del control de este robot se realizaba remotamente, estaba dotado de una cierta inteligencia que lo diferencia mucho de un simple robot por "radio control".

La misión a Marte fue un éxito, aunque el vehículo se quedó a veces "atascado", teniendo que hacer muchas maniobras para poder avanzar por el pedregoso terreno. Esto puso de manifiesto dos aspectos importantes: por un lado la viabilidad de enviar robots en vez de hombres, que es mucho más barato y menos arriesgado. Por otro lado la necesidad de enviar robots cada vez más versátiles y que se adapten a cualquier superficie.

En la figura 8 se muestra una foto del *Microrover* sobre la superficie de Marte, tomada desde la sonda. Está analizando la roca "*Yogi*". Puede verse la cantidad de piedras que hay sobre el terreno y lo complicado que resultó para el robot poder acceder a la roca. En la figura 9 se muestra una versión prototipo del *Microrover* que se está probando en los laboratorios de la *NASA*, en un entorno que simula la superficie marciana.

Figura 8. Vehículo *MICROROVER* analizando la roca "*yogi*" en Marte.

Figura 9. Prototipo del *MICROROVER* en los laboratorios de la *NASA*.



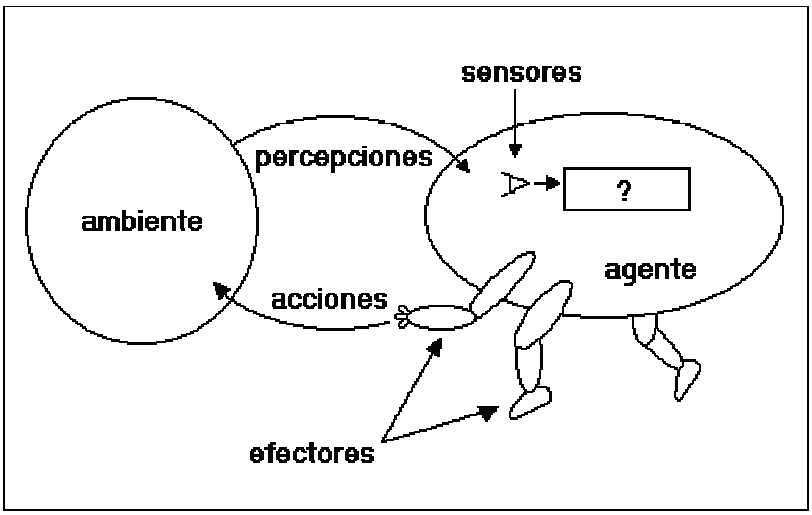
2. AGENTES INTELIGENTES

2.1. Introducción

Un agente es todo aquello que puede considerarse que percibe su ambiente mediante sensores y que responde o actúa en tal ambiente por medio de efectores. Los agentes humanos tienen ojos, oídos y otros órganos que les sirven de sensores, así como manos, piernas, boca y otras partes de su cuerpo que les sirven de efectores. En el caso de los agentes robóticos, los sensores son sustituidos por cámaras y telémetros infrarrojos, y los efectores son reemplazados mediante motores. En el caso de un agente de software, sus percepciones y acciones vienen a ser las cadenas de bits codificados. En la figura 10 puede observarse el diagrama de una agente genérico.

En este capítulo se mostrarán los diferentes diseños de agentes, para seleccionar el diseño que logre un buen desempeño en los ambientes tipo laberinto. Primero se intentará precisar qué se entiende por un buen desempeño y luego se hablará de los diversos diseños de agentes satisfactorios: la respuesta al signo de interrogación de la figura 10. Se mencionarán algunos de los principios generales aplicados al diseño de agentes. Finalmente se enseñará a conjuntar un agente y un ambiente determinados y se explicarán diversos tipos de ambientes.

Figura 10. Los agentes interactúan con los ambientes a través de sensores y efectores.



2.2. Agentes inteligentes

2.2.1. Cómo debe proceder un agente

Un agente racional es aquel que hace lo correcto. Obviamente, esto es preferible a que haga algo incorrecto, pero ¿qué significa? Como un primer intento de aproximación, se afirmará que lo correcto es aquello que permite al agente obtener el mejor desempeño. Dicho lo anterior, ahora será necesario decidir cómo y cuándo evaluar ese buen desempeño del agente.

El término **medición del desempeño** se aplica al *cómo*: es el criterio que sirve para definir qué tan exitoso ha sido un agente. Desde luego que no existe una medida fija que se pueda aplicar por igual a todos los agentes. Se le podría preguntar al agente su opinión subjetiva de cuándo le satisface a él mismo su desempeño; sin embargo, algunos de ellos no estarían en posibilidad de responder y otros, simplemente eludirían responder. (En particular, los agentes humanos se caracterizan por responder que "las uvas están verdes" para justificar el no haber obtenido lo que originalmente deseaban).

Por lo anterior, se debe insistir en la necesidad de contar con una medición objetiva del desempeño, medida que deberá ser propuesta por una autoridad. En otras palabras, un observador externo, define la norma de lo que se considera un satisfactorio desempeño en un ambiente y lo emplea en la medición del desempeño de los agentes. Un ejemplo sería el caso de un agente al que se le encomienda limpiar con una aspiradora un piso sucio. Una posible medida de su desempeño sería la cantidad de mugre eliminada en un turno de ocho horas. Una medida más elaborada consistiría en correlacionar la cantidad de electricidad consumida y la cantidad de ruido generada. Una tercera, otorgaría la máxima calificación a un agente que no sólo limpiase silenciosa y eficientemente el piso, sino que también se diera tiempo para ir a pasear el fin de semana.

El **cuándo** evaluar el desempeño es importante también. Si sólo se midiera cuánta mugre eliminó un agente durante la primera hora del día, aquellos agentes que empiezan a trabajar rápidamente resultarían premiados, en tanto que aquellos que laboran todo el turno de manera consistente resultarían castigados. Es decir, lo importante es medir el desempeño a largo plazo, sea éste un turno de ocho horas o una vida.

Hay que dejar claro que existe una diferencia entre racionalidad y **omnisciencia**. Un agente omnisciente es aquel que sabe el resultado **real** que producirán sus acciones, y su conducta es congruente con ello; sin embargo, en la realidad, no existe la omnisciencia. Considérese el siguiente ejemplo: Un día un agente humano va caminando por la avenida Reforma y ve que al otro lado de la calle está un antiguo amigo. No hay tránsito en las cercanías y no lleva ninguna prisa; así que, actuando racionalmente, empieza a cruzar la calle. Al mismo tiempo, a una altura de 10,000 metros se desprende la puerta de la sección de carga aérea de un avión de pasajeros, y, antes de poder alcanzar la otra acera, queda completamente aplastado. ¿Se consideraría como irracional el haber atravesado la calle?.

Del ejemplo anterior se puede deducir que la racionalidad tiene más bien que ver con un éxito esperado, tomando como base lo que se ha percibido. La decisión de atravesar la calle fue racional dado que la gran mayoría de las veces el cruce habría tenido éxito y no había forma de que se hubiese previsto la caída de la puerta. Un agente que hubiera contado con un radar para detectar puertas en caída, o que estuviera dentro de una caja de acero que lo protegiera, habría tenido más éxito, pero no por ello habría sido más racional.

En otras palabras, no se puede culpar a una agente por no haber tomado en cuenta algo que no podía percibir, o por no emprender una acción (como repeler la puerta) de la que es incapaz. Sin embargo, la tolerancia en relación con la exigencia de perfección no es algo que tenga que ver con una actitud justa a favor de los agentes. Lo importante es que si se especifica que un agente inteligente siempre debe hacer lo que realmente es correcto, será imposible diseñar un agente para que satisfaga esta especificación.

En resumen, el carácter de racionalidad de lo que se hace un momento dado dependerá de cuatro factores:

- De la medida con la que evalúa el grado de éxito logrado.
- De todo lo que hasta ese momento haya percibido el agente. A esta historia perceptual completa se le denomina la **secuencia de percepciones**.
- Del conocimiento que posea el agente acerca del medio.
- De las acciones que el agente puede emprender.

De lo anterior se puede definir lo que es un **agente racional ideal**: en todos los casos de posibles secuencias de percepciones, un agente racional deberá emprender todas aquellas acciones que favorezcan obtener el máximo de su medida de rendimiento, basándose en las evidencias aportadas por la secuencia de percepciones y en todo conocimiento incorporado en tal agente.

Es necesario examinar cuidadosamente la definición anterior. Aparentemente, con esta definición se le permite a una gente emprender actividades definitivamente carentes de inteligencia. Por ejemplo, si un agente no se fija en el tráfico de ambos sentidos de una calle antes de proceder a cruzarla, su secuencia de percepciones no le podrá informar que un enorme camión carguero se aproxima a alta velocidad. En apariencia la definición afirmaría que es correcto que el agente proceda a atravesar la calle. Sin embargo, la interpretación anterior está equivocada por dos razones. La primera es que no sería racional atravesar la calle: el riesgo de hacerlo así es enorme. Segundo, un agente racional ideal invariablemente elegiría la acción de "verificar el tráfico" antes de lanzarse a la calle, ya que esta verificación le permite obtener lo mejor de su desempeño esperado. El emprender acciones con el fin de obtener información útil es parte importante de la racionalidad.

El concepto de agente permite pensar en él como herramienta para el análisis de sistemas, no como una caracterización absoluta que tajantemente divida al mundo en agentes y no agentes. Tómese el caso de un reloj. Puede considerarse como un objeto inanimado, o bien, como un agente sencillo. En cuanto éste último, todos los relojes hacen siempre lo correcto: mueven las manecillas de reloj (o presentan dígitos) de la manera adecuada. Los relojes, en cierto sentido, son agentes degenerados, dado que su secuencia de percepciones está vacía; independientemente de lo que suceda en el exterior, las acciones del reloj no se ven afectadas.

En realidad, lo anterior no es completamente cierto. Si el reloj y su propietario partieran de viaje de California a Australia, lo adecuado sería que el reloj se retrasara seis horas. Si los relojes no hicieran lo anterior, los propietarios no estarían molestos con ellos puesto que están conscientes de que están actuando racionalmente, considerando que no cuentan con un mecanismo de percepción.

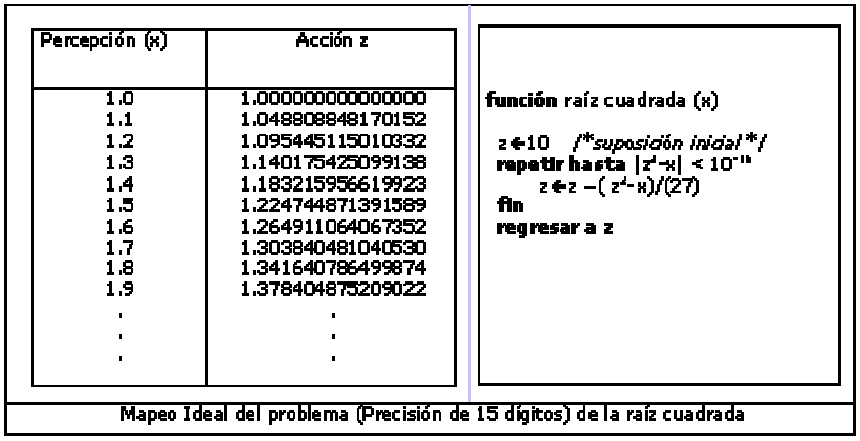
2.2.2. El mapeo ideal de las secuencias de percepciones para acciones

Una vez que se ha determinado que el conocimiento de una agente depende exclusivamente de la secuencia de sus percepciones en un momento dado, se sabe que es posible caracterizar cualquier agente en particular elaborando una tabla de las acciones que éste emprende como respuesta a cualquier secuencia de percepciones posible. La anterior tabla se denomina un mapeo de secuencias de percepciones para acciones. En principio, es posible determinar qué mapeo describe acertadamente a una agente, ensayando todas las secuencias de percepciones posibles y llevando un registro de las acciones que en respuesta emprende el agente. Si mediante los mapeos se caracteriza a los agentes, los mapeos ideales caracterizan a los agentes ideales. El especificar qué tipo de acción deberá emprender un agente como respuesta a una determinada secuencia de percepciones constituye el diseño de un agente ideal.

Lo anterior no implica, desde luego, que hay que crear una tabla explícita con una entrada por cada posible secuencia de percepciones. Es vez de ello, se puede definir una especificación del mapeo sin tener que enumerarlo exhaustivamente.

Considérese el caso de un agente muy sencillo: la función raíz cuadrada de una calculadora. La secuencia de percepciones del agente anterior es una secuencia de pulsaciones de tecla que representan un número; y la acción consiste en presentar un número en la pantalla. El mapeo ideal se produce cuando la percepción es un número positivo x, y la acción correcta consiste en presentar un número positivo z tal que $z^2 \approx x$, con una precisión de 15 cifras decimales. La anterior especificación del mapeo ideal no implica que el diseñador construya realmente una tabla de raíces cuadradas. Ni tampoco la función raíz cuadrada tiene que utilizar una tabla para responder correctamente: en la figura 11 se muestra parte del mapeo ideal y un sencillo programa que permite la implantación del mapeo utilizando el método de Newton.

Figura 11. Parte del mapeo ideal del problema de la raíz cuadrada y el programa correspondiente para implantar el mapeo ideal.



Percepción (x)	Acción z
1.0	1.000000000000000
1.1	1.048808848170152
1.2	1.095445115010332
1.3	1.140175425099138
1.4	1.183215956619923
1.5	1.224744871391589
1.6	1.264911064067352
1.7	1.303840481040530
1.8	1.341640786499874
1.9	1.378404875209022
.	.
.	.
.	.

```
función raíz cuadrada (x)

 z←10   /*suposición inicial*/
repetir hasta |z²-x| < 10⁻¹⁵
     z←z –( z²-x)/(2z)
fin
regresar a z
```

Mapeo Ideal del problema (Precisión de 15 dígitos) de la raíz cuadrada

2.2.3. Autonomía

En la definición de agente racional ideal hay un elemento más al que se debe prestar atención: la parte del "conocimiento integrado". Si las acciones que emprende el agente se basan exclusivamente en un conocimiento integrado, con lo que se hace caso omiso de sus percepciones, se dice que el agente no tiene autonomía. Por ejemplo, si el fabricante de relojes tuviera la capacidad de saber con antelación que el propietario del reloj viajará a Australia en una fecha determinada, incorporaría un mecanismo para ajustar automáticamente las manecillas seis horas en el momento preciso. Sin lugar a dudas la anterior es una conducta satisfactoria, pero la inteligencia respectiva es mérito del diseñador del reloj, no del reloj mismo.

La conducta de un agente se basa tanto en su propia experiencia como en el conocimiento integrado que sirve para construir al agente para el ambiente específico con el cual va a operar. Un sistema será autónomo en la medida en que su conducta está definida por su propia experiencia. Sin embargo, sería excesivo el esperar una total autonomía tan sólo con mencionar la palabra "adelante": si el agente cuenta con poca o ninguna experiencia, tendrá que comportarse de manera aleatoria a menos que el diseñador le dé algún tipo de ayuda. Y así como la evolución ha dotado a los animales con una dotación suficiente de reflejos incorporados, a fin de que sobrevivan lo suficiente hasta que sean capaces de aprender por sí mismos, también es razonable dotar a un agente de inteligencia artificial con ciertos conocimientos iniciales y de capacidad para aprender.

La autonomía no sólo tiene cabida en lo que indica la intuición, también es un ejemplo del sólido manejo de la práctica. Cuando un agente opera basándose en suposiciones en él integradas, su actuación será satisfactoria sólo en la medida en que tales suposiciones sean vigentes y, por ello, carecerá de flexibilidad.

Considérese, por ejemplo, el caso del humilde escarabajo del estiércol. Luego de cavar su nido y depositar en él sus huevecillos, toma una bola de estiércol de una pila cercana para tapar con ella la entrada; si durante el trayecto se le quita la bola, el escarabajo continuará su recorrido y hará como si estuviera tapando la entrada del nido, no obstante que ya no tenga consigo la bola y sin que se dé cuenta de ello. La evolución incorporó una suposición en la conducta del escarabajo; si tal suposición se viola, se produce la consecuente conducta insatisfactoria. El auténtico agente inteligente autónomo debe ser capaz de funcionar satisfactoriamente en una amplia gama de ambientes, considerando que se le da tiempo suficiente para adaptarse.

.

2.2.4. Estructura de los agentes inteligentes

Hasta ahora se ha referido a los agentes mediante la descripción de su conducta: aquellas acciones que se producen después de una determinada secuencia de percepciones. Es tiempo ya de ver cómo funcionan las cosas desde adentro. El cometido de la IA es el diseño de un programa de agente: una función que permita implantar el mapeo del agente para pasar de percepciones a acciones. Se da por sentado que este programa se ejecutará en algún tipo de dispositivo de cómputo, al que se denominará arquitectura. Desde luego, el programa elegido debe ser aquel que la arquitectura acepte y pueda ejecutar. La arquitectura puede ser una computadora sencilla o hardware especial para la ejecución de ciertas tareas. Podría incluir también un software que ofrezca cierto grado de aislamiento entre la computadora y el programa de agente, lo que permitiría la programación a un nivel superior. En general, la arquitectura pone al alcance del programa las percepciones obtenidas mediante los sensores, lo ejecuta y alimenta el efector con las acciones elegidas por el programa conforme éstas se van generando. La relación entre agentes, arquitectura y programas podrían resumirse de la siguiente manera:

agente = arquitectura + programa

Antes de proceder al diseño de un programa de agente, es necesario contar con una idea bastante precisa de las posibles percepciones y acciones que intervendrán, qué metas o medidas de desempeño se supone lleve a cabo el agente, así como del tipo de ambiente en que tal agente operará. Éstos pueden ser muy variados. En la tabla I se muestran los elementos básicos que se consideran en la elección de los tipos de agente.

Tabla I. Ejemplos de diversos tipos de agentes y sus correspondientes descripciones PAMA.

Tipo de agente	**Percepciones**	**Acciones**	**Metas**	**Ambientes**
Sistema para diagnósticos médicos	Síntomas, evidencias y respuestas del paciente	Preguntas, pruebas, tratamientos	Paciente saludable, reducción al mínimo de costos	Paciente, hospital
Sistema para el análisis de imágenes de satélite	Píxeles de intensidad y colores diversos	Imprimir una clasificación de escena	Clasificación correcta	Imágenes enviadas desde un satélite en órbita
Robot clasificador de partes	Píxeles de intensidad variable	Recoger partes y clasificarlas poniéndolas en botes	Poner las partes en el bote que les corresponda	Banda transportadora sobre la que se encuentran las partes
Controlador de una refinería	Lecturas de temperatura y presión	Abrir y cerrar válvulas; ajustar la temperatura	Lograr pureza, rendimiento y seguridad máximos	Refinería
Asesor interactivo de inglés	Palabras escritas a máquina	Ejercicios impresos, sugerencias y correcciones	Que el estudiante obtenga la máxima calificación en una prueba	Grupo de estudiantes

En contraste, con algunos muy ricos e ilimitados ámbitos se utilizan agentes de software (o robots de software, o *softbots*). Imagine un *softbot* diseñado para pilotear el simulador de vuelo de un 747. El simulador constituye un ambiente muy detallado y complejo; en él, el agente de software debe elegir entre una amplia gama de acciones en tiempo real. O, imagine por ejemplo un *softbot* diseñado para que revise fuentes de noticias en línea y escoja entre ellas las que sean del interés de sus clientes. Para lograrlo, deberá estar dotado con la capacidad de procesamiento del lenguaje natural, tendrá que aprender qué es lo que interesa a cada uno de los clientes y es necesario que pueda modificar en forma dinámica sus planes, cuando, por ejemplo, se interrumpa la conexión con una de las fuentes noticiosas o una nueva se incorpore a la línea.

En algunos ambientes se diluye la diferencia entre "real" y "artificial". En el ambiente ALIVE, los agentes reciben como percepción la imagen digitalizada de una cámara colocada en una habitación en la que deambula un ser humano. El agente procesa la imagen de la cámara y decide la acción que va a emprender. El ambiente presenta la imagen producida por la cámara en una pantalla grande, a la vista del humano y sobrepone a tal imagen una gráfica de computadora que es una representación del agente de software. Esta imagen puede ser un perro de caricatura, programado para aproximarse al humano, para dar la pata o saltar animadamente cuando el humano hace ciertos gestos.

2.2.4.1. Programas de agentes

En la figura 12 se muestra un esqueleto básico de los agentes, consiste en la aceptación de percepciones originadas en un ambiente y la generación de acciones respectivas. En los programas de agentes se emplean algunas estructuras de datos internos que se irán actualizando con la llegada de nuevas percepciones. Tales estructuras de datos se operarán mediante los procedimientos de toma de decisiones de un agente para generar la elección de una acción, elección que se transferirá a la arquitectura para proceder a su ejecución.

Figura 12. Esqueleto de un agente.

```
function SKELETON-AGENT(percept) returns action
función ESQUELETO-AGENTE(percepción) responde con una acción
        Estática: memoria, la memoria del mundo del agente

        memoria← ACTUALIZACIÓN-MEMORIA(memoria,percepción)
        acción← ESCOGER-LA-MEJOR-ACCIÓN(memoria)
        memoria ← ACTUALIZACIÓN-MEMORIA(memoria, acción)
        Responde con una acción
```

En relación con este programa esqueleto hay dos aspectos que es necesario comentar. Primero, si bien se definió el mapeo de un agente como una función para pasar de secuencias de percepciones a acciones, el programa de agente recibe como entrada sólo una percepción. Es decisión del agente construir la secuencia de percepciones en la memoria. Algunos ambientes permiten funcionar perfectamente sin necesidad de almacenar la secuencia de percepciones; en algunos dominios complejos no es factible el almacenaje de la totalidad de secuencia.

Segundo, la meta o la medición del desempeño no forma parte del programa esqueleto. La razón es que la medición del desempeño se efectúa desde afuera, a fin de evaluar la conducta del agente; es frecuente que se obtenga un alto desempeño sin contar con un conocimiento explícito de la medición del desempeño.

2.2.4.2. Un ejemplo

Sería conveniente ahora considerar un ambiente en particular con el fin de hacer más concreta esta explicación. Principalmente por lo familiar que resulta y también por la enorme gama de destrezas que entraña. Se escogerá la tarea del diseño de un conductor de taxis automatizado. La tarea de conducir un automóvil, en su totalidad, es extremadamente ilimitada: no hay límite en cuando a nuevas combinaciones de circunstancias que pueden surgir.

Habrá que considerar las percepciones, acciones, metas y ambientes que corresponden a un taxi. Están resumidos en la tabla II y se procederá a explicarlos.

Tabla II. El tipo de agente conductor de taxi.

Tipo de agente	Percepciones	Acciones	Metas	Ambiente
Conductor de taxi	Cámaras, velocímetro, sistema satélite global de ubicación, sonar, micrófono	Manejo del volante, acelerar, frenar, hablar con el pasajero	Un viaje seguro, rápido, sin infracciones, cómodo, obtención máxima de ganancias	Caminos, tráfico, peatones, clientes

El taxi necesita saber dónde se encuentra, quién más circula por su vía y a qué velocidad corre. Esta información se obtiene de las percepciones que ofrecen una o más cámaras de televisión controlables, el velocímetro y el contador de kilómetros. A fin de tener un adecuado control del vehículo, especialmente en las curvas, es necesario contar con un acelerómetro; también se necesita conocer el estado mecánico del vehículo, para ello se debe contar con sensores para los sistemas de la maquinaria y eléctrico. Posiblemente se le dotaría de instrumentos con los que no cuenta un conductor humano promedio: un sistema satélite global para determinar la ubicación, mediante el que se generará información precisa sobre su ubicación en relación con un mapa electrónico; o sensores infrarrojos o de sonar para detectar las distancias que lo separan de otros carros y obstáculos. Por último, deberá estar provisto de un micrófono o de un teclado para que por medio de éstos los pasajeros le puedan informar cual es el destino deseado.

Las acciones que puede producir este conductor de taxi son más o menos las mismas que las de un conductor humano: control de la máquina mediante pedal para combustible y control del volante y de los frenos. Además, necesitará una salida en pantalla o sintetizador de voz que le permita contestar a los pasajeros y, quizás, de algún medio para comunicarse con otros vehículos.

¿Cuál sería la medida de desempeño que el conductor automatizado desearía satisfacer? Entre las cualidades deseables están la de llegar al destino correcto, reducir al mínimo el consumo de combustible, desgaste del vehículo, tiempo de recorrido, su costo, violaciones al reglamento de tránsito y las molestias ocasionadas a otros conductores; ofrecer el máximo de seguridad y comodidad al pasajero y producir el máximo de ganancias. Desde luego que el logro de algunos de estos objetivos está en conflicto con el logro de otros, por lo que será necesario implicar concesiones.

Por último, en caso de que le anterior fuese un proyecto real, sería necesario definir qué tipo de ambiente de conducción enfrentará el taxi. ¿Operaría en calles comunes o en vías de alta velocidad? ¿Se conducirá siempre a la derecha, o se tendrá la flexibilidad de permitirle también conducir a la izquierda, en caso de que se quieran operar taxis en Gran Bretaña o Japón? Obviamente, cuanto más restringido sea el ambiente, menos complicado será el problema del diseño.

Es tiempo ahora de decidir cómo se construirá un programa real para implantar el mapeo que permitirá pasar de percepciones a acciones. Nos damos cuenta de que los diversos aspectos del conducir un auto dan pie a proponer diversos tipos de agente. Consideremos cuatro tipos de estos programas:

- Agentes de reflejo simple
- Agentes bien informados de todo lo que pasa
- Agentes basados en metas
- Agentes basados en utilidad

2.3. Tipos de agentes

2.3.1. Agentes de reflejo simple

El recurso de utilizar una tabla de consulta está fuera de toda consideración. La señal de entrada visual producida por una sola cámara tiene una frecuencia de 50 *megabytes* por segundo (25 fotogramas por segundo, 1000*1000 píxeles con 8 bits de color y 8 bits de información de intensidad). La tabla de consulta correspondiente a una hora tendría $2^{60*60*50M}$ entradas.

No obstante, es posible resumir fragmentos de la tabla si se presta atención a ciertas asociaciones entre entradas/salidas que se producen con relativa frecuencia. Por ejemplo, si el carro de adelante frena, enciende las luces de alto correspondientes, el conductor de atrás lo percibe e inicia, a su vez, el frenado. Es decir, se realiza un proceso como respuesta a la entrada visual y se establece la condición "el carro de enfrente está frenando"; esto activa la conexión ya definida en el programa del agente y la acción correspondiente "empiece a frenar". A esta conexión se le denomina regla de condición-acción y se describe:

Si *el carro de adelante está frenando* **entonces** *empiece a frenar*

Los humanos funcionan también en muchos casos recurriendo a estas conexiones, algunas de las cuales son respuestas aprendidas (como en el caso del manejo de un auto) y otras son reflejos innatos (como el parpadeo del ojo cuando se le aproxima algún objeto).

En la figura 13 se presenta la estructura de un agente reflejo sencillo en forma esquematizada. En ella se puede observar cómo las reglas condición-acción permiten al agente establecer la conexión entre percepciones y acciones. Los rectángulos se usan para indicar el estado interno en un momento dado del proceso de decisión del agente; los óvalos representan la información de base utilizada en el proceso. El programa de agente, también muy sencillo, se muestra en la figura 14. La función INTERPRETAR-ENTRADA genera una descripción abstracta del estado prevaleciente de la percepción; y la función REGLA-COINCIDENCIA produce la primera regla en el conjunto de reglas que hace juego con la descripción del estado ofrecida. Si bien agentes como el anterior se pueden implantar con bastante eficiencia, la extensión del ámbito en donde se pueden aplicar es bastante limitada, como se verá.

Figura 13. Diagrama esquematizado de un agente reflejo simple.

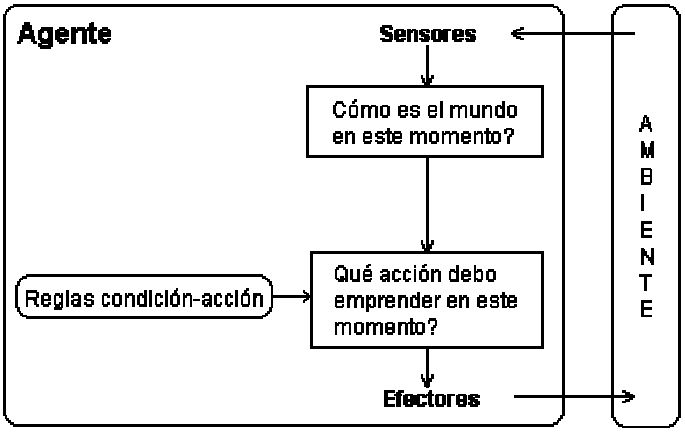


Diagrama esquematizado de un agente reflejo simple

Figura 14. Programa de un agente reflejo simple.

```
function SIMPLE-REFLEX-AGENT(percept) returns action
función AGENTE-REFLEJO SIMPLE(percepción) responde con una acción
        estático: reglas, un conjunto de reglas de condición-acción
        estado ← INTERPRETAR-ENTRADA(percepción)
        regla ← REGLA-COINCIDENCIA(estado, reglas)
        responder con una acción
```

2.3.2. Agentes bien informados de todo lo que pasa

El agente reflejo simple explicado anteriormente funcionará sólo si se toma la decisión adecuada con base en la percepción de un momento dado. Si el carro que va delante es un modelo reciente, y tiene instalada la luz central indicadora de frenado ahora obligatoria en Estados Unidos, bastará con una imagen para darse cuenta que esta frenando ese carro. Lamentablemente, en los modelos más antiguos el diseño de luces, luces de frenado y luces direccionales es distinto, por lo que no siempre es posible decir con precisión si el carro está frenando. Es decir aún en el caso de la regla de frenado más sencilla, el conductor tiene que mantener cierto tipo de estado interno, con el fin de estar en condiciones de optar por una acción. En este caso el estado interno no es demasiado extenso: lo único que necesita es la imagen anterior de la cámara para detectar cuándo se encienden o se apagan simultáneamente dos luces rojas ubicadas en los extremos del carro.

Considérese el siguiente caso, más evidente: de vez en vez, el conductor mira al espejo retrovisor para percibir la ubicación de los vehículos cercanos. Cuando el conductor no ve el espejo, los vehículos del carril contiguo resultan invisibles; pero si el conductor quiere decidir una maniobra de cambio de cambio de carril será necesario que sepa si hay vehículos en el carril contiguo.

El problema aquí ejemplificado surge debido a que los sensores no informan acerca del estado total del mundo. En casos así, el agente necesita actualizar algo de información en el estado interno que le permita discernir entre estados del mundo que generan la misma entrada de percepciones pero que, sin embargo, son totalmente distintos. Al decir “totalmente distintos” significa que para cada uno de los estados se necesitan acciones distintas.

La actualización de esta información sobre el estado interno conforme va pasando el tiempo exige la codificación de dos tipos de conocimiento en el programa de agente. En primer lugar, se necesita cierta información sobre cómo está evolucionando el mundo, independientemente del agente; por ejemplo, un carro que va alcanzando a otro automóvil generalmente estará más cercano a la parte trasera de éste que lo que estaba hace un momento. En segundo lugar, se necesita información sobre cómo las acciones del agente mismo afectan al mundo; por ejemplo, cuando el agente cambia de carril a la derecha, queda un lugar vacío (por lo menos temporalmente) en el carril en donde se encontraba, o que después de conducir durante cinco minutos en dirección norte en una autopista por lo general se encuentra a unas cinco millas al norte de donde se entraba hace cinco minutos.

En la figura 15 se puede observar la estructura del agente reflejo y también cómo se combinan las percepciones prevalecientes con el estado interno anterior para generar la descripción actualizada del estado prevaleciente. El programa de agente se muestra en la figura 16. Lo más interesante es la función ACTUALIZAR-ESTADO, que es la responsable de crear la nueva descripción del estado interno. Así como interpretar la nueva percepción a la luz del conocimiento disponible sobre el estado, también utiliza la información referente a la manera como evoluciona el mundo y así mantenerse informada acerca de esas partes no visibles de él, y además debe estar bien enterada en lo tocante a cómo las acciones del agente están afectando al estado del mundo.

Figura 15. Un agente reflejo con un estado interno

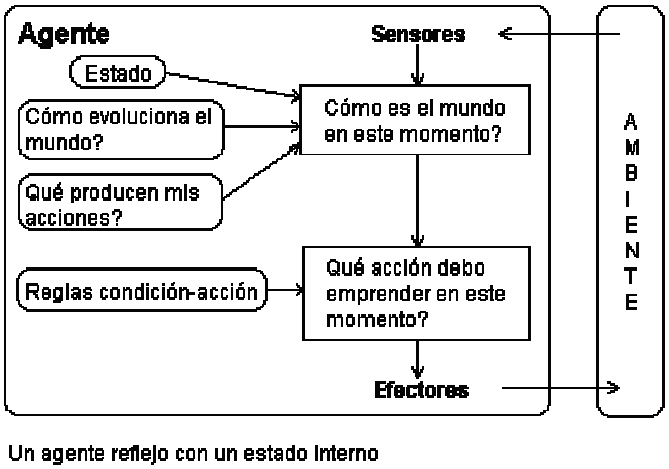


Figura 16. Programa de un agente reflejo con un estado interno

function RFLEX-AGENT-WITH-STATE(*percept*) ***return*** *action*

función AGENTE-REFLEJO-CON-ESTADO(*percepción)* **responde** con una *acción*

estático: estado, una descripción prevaleciente del estado del mundo

reglas, conjunto de reglas condición-acción

estado ← ACTUALIZAR-ESTADO(*estado,percepción*)

regla ← REGLA-COINCIDENCIA(*estado,reglas*)

acción ← REGLA-ACCION[*regla*]

estado ← ACTUALIZAR-ESTADO(*estado,acción*)

responder con una *acción*

2.3.3. Agentes basados en metas

Para decidir qué hay que hacer no siempre basta con tener información acerca del estado que prevalece en el ambiente. Por ejemplo, al llegar a un crucero, el taxi puede optar por dar vuelta a la derecha, a la izquierda o seguir de frente. La decisión adecuada dependerá de adónde desee llegar el taxi. Es decir, además de una descripción del estado prevaleciente, el taxi también requiere de cierto tipo de información sobre su meta, información que detalle las situaciones deseables. Por ejemplo, llegar al punto de destino del pasajero. El programa de agente puede combinar lo anterior con la información relativa al resultado que producirán las posibles acciones que se emprendan (esta misma información se utilizó para actualizar el estado interno en el caso del agente reflejo) y de esta manera elegir aquellas acciones que permitan alcanzar la meta. En ocasiones esto es muy sencillo, cuando el agente tenga que considerar largas secuencias de giros y vueltas hasta que logre encontrar la vía que le lleve a la meta. La búsqueda y planificación son los subcampos de la IA que se ocupan de encontrar las secuencias de aplicaciones que permiten alcanzar las metas de un agente.

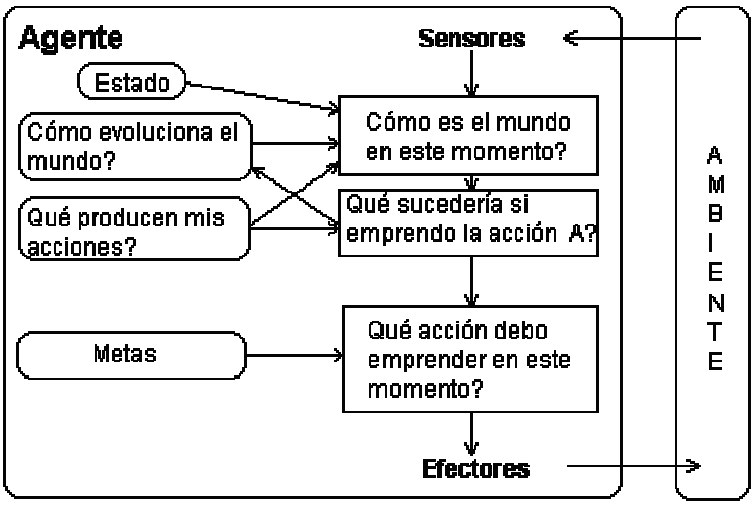
Nótese que la toma de decisiones de este tipo difiere radicalmente de las reglas condición-acción explicadas anteriormente, en que implica el tomar en cuenta el futuro: tanto el "¿qué sucederá si se hace tal o cual cosa?" con el "¿y esto hará feliz al agente?" en el caso del diseño del agente reflejo simple esta información no se utiliza explícitamente puesto que el diseñador calcula previamente la acción correcta correspondiente a diversos casos. El agente reflejo frena en cuanto ve que se encienden las luces de frenado.

Un agente basado en metas, en principio, razonaría que si en el carro de enfrente se han encendido las luces de frenado, deberá disminuir velocidad. Considerando la forma como normalmente evolucionan las cosas en el mundo, la única acción que permitiría alcanzar la meta de no chocar con otro vehículo es la de frenar.

Si bien el agente basado en metas es menos eficiente, también es cierto que es más flexible. Si empieza a llover, el agente tendrá la posibilidad de actualizar su conocimiento de qué tan eficiente podrán funcionar sus frenos; esto trae como consecuencia la alteración automática de todas las conductas anteriores con el fin de adaptarse a las nuevas condiciones. Por el contrario, en el caso del agente reflejo habría que reelaborar una gran cantidad de reglas condición-acción. Asimismo, el agente basado en metas es también más flexible con respecto a dirigirse a diferentes destinos. Tan sólo con especificar un nuevo destino, se produce en el agente basado en metas una nueva conducta. Las reglas de un agente de reflejo acerca de cuándo dar vuelta y cuándo seguir derecho funcionarán tan sólo en el caso de un destino; siempre que surja uno nuevo habrá que sustituirlas.

En la figura 17 se muestra la estructura del agente basado en metas.

Figura 17. Un agente con metas explícitas.



Un agente con metas explícitas

2.3.4. Agentes basados en utilidad

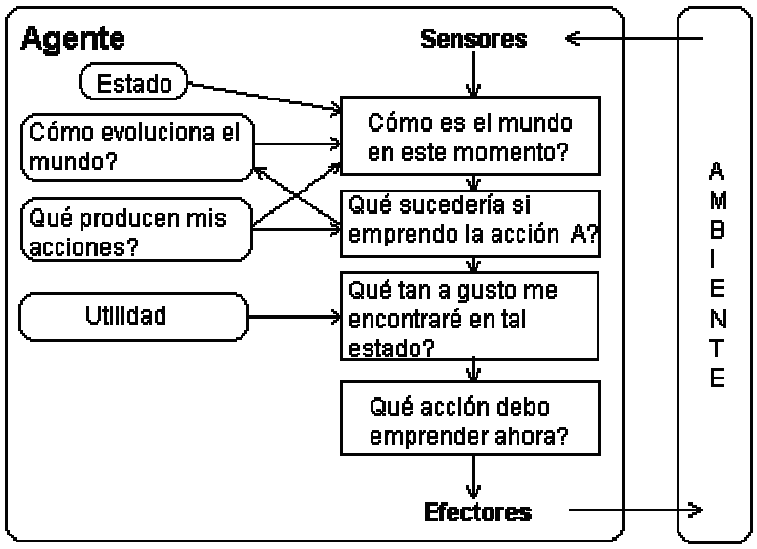
Las metas no bastan por sí mismas para generar una conducta de alta calidad. Por ejemplo, son muchas las secuencias de acciones que permitirían al taxi llegar a su destino, con lo que se habría logrado la meta, pero de todas ellas, algunas son más rápidas, seguras y confiables, o baratas que las demás. Las metas permiten establecer una tajante distinción entre estados "felices" e "infelices", en tanto que mediante una medida de desempeño más general sería posible establecer una comparación entre los diversos estados del mundo (o secuencias de estados) de acuerdo a cómo exactamente harían feliz al agente en caso de poder lograrlos. Puesto que el término "feliz" no suena muy científico, la terminología que se acostumbra a utilizar es afirmar que si se prefiere un estado del mundo por otro, entonces ese estado ofrece mayor utilidad al agente.

Por lo tanto, la utilidad es una función que correlaciona un estado y un número real mediante el cual se caracteriza el correspondiente grado de satisfacción. La completa especificación de la función de utilidad permite la toma de decisiones racionales en dos tipos de casos en los que las metas se encuentran con problemas. El primero, cuando el logro de algunas metas implica un conflicto, y sólo algunas de ellas se pueden obtener (por ejemplo, la velocidad y la seguridad), la función de utilidad definirá cuál es el compromiso adecuado por el que hay que optar. Segundo, cuando son varias las metas que el agente podría desear obtener, pero no existe la certeza de poder lograr ninguna de ellas, la utilidad es una vía para ponderar la posibilidad de tener éxito considerando la importancia de las diferentes metas. Así pues, un agente que tiene una función de utilidad explícita está en posibilidad de tomar decisiones racionales, aunque quizás tenga que comparar las utilidades obtenidas mediante diversas acciones.

Las metas, no obstante su inflexibilidad, permiten al agente optar de inmediato por una acción cuando ésta permite alcanzarlas. Además, en algunos casos, se puede traducir la función de utilidad en un conjunto de metas, de manera que las decisiones adoptadas por el agente basado en metas tomando como base tal conjunto de metas resulten idénticas a las que haría el agente basado en la utilidad.

La estructura general de un agente basado en utilidad se muestra en la figura 18.

Figura 18. Un agente completo basado en la utilidad.



Un agente completo basado en la utilidad

2.4. Ambientes

En la sección anterior se presento los diversos tipos de agentes y sus ambientes. En todos los casos, la relación que existe entre ellos es siempre la misma: es el agente quien ejerce acciones sobre el ambiente, el que, a su vez, aporta percepciones al agente. Primero se explicará cuáles son los diferentes tipos de ambientes y cómo condicionan el diseño de los agentes. Luego se explicarán los programas de ambientes que sirven como campo de prueba de los programas de agentes.

2.4.1. Propiedades de los ambientes

Los ambientes vienen en varios sabores. Las diferencias básicas son las siguientes:

2.4.1.1. Accesibles y no accesibles

Si el aparato sensorial de un agente le permite tener acceso al estado total de un ambiente, se dice que éste es accesible a tal agente. Un ambiente es realmente accesible si los sensores detectan todos los aspectos relevantes a la elección de una acción. Los ambientes accesibles son cómodos, puesto que no es necesario que el agente mantenga un estado interno para estar al tanto de lo que sucede en el mundo.

2.4.1.2. Deterministas y no deterministas

Si el estado siguiente de un ambiente se determina completamente mediante el estado actual y las acciones escogidas por los agentes, se dice que el ambiente es determinista. En principio, un agente no tiene por qué preocuparse sobre la incertidumbre en un ambiente accesible y determinista. Pero si el ambiente es inaccesible, entonces podría parecer que es no determinista.

2.4.1.3. Episódicos y no episódicos

En un ambiente episódico, la experiencia del agente se divide en "episodios". Cada episodio consta de un agente que percibe y actúa. La calidad de su actuación dependerá del episodio mismo, dado que los episodios subsecuentes no dependerán de las acciones producidas en episodios anteriores. Los ambientes episódicos son más sencillos puesto que el agente no tiene que pensar por adelantado.

2.4.1.4. Estáticos y dinámicos

Si existe la posibilidad de que el ambiente sufra modificaciones mientras el agente se encuentra deliberando, se dice que tal ambiente se comporta de forma dinámica en relación con el agente; de lo contrario, se dice que es estático. Es más fácil trabajar con ambientes estáticos puesto que el agente no tiene que observar lo que sucede en el mundo al mismo tiempo que decide sobre el curso de una acción, ni tampoco tiene que preocuparse por el paso del tiempo. Si el ambiente no cambia con el paso del tiempo, pero sí se modifica la calificación asignada al desempeño de un agente, se dice que el ambiente es semi-dinámico.

2.4.1.5. Discretos y continuos

Si existe una cantidad limitada de percepciones y acciones distintas y claramente discernibles, se dice que el ambiente es discreto. El ajedrez es discreto: existe una cantidad fija de posibles jugadas en cada ronda. La conducción de un taxi es continua: la velocidad y la ubicación del taxi y de los demás vehículos se extiende a través de un rango de valores continuos.

Se mostrará como para los distintos tipos de ambientes se necesitan programas de agente relativamente diferentes para así poder trabajar con ellos de manera eficiente. El resultado, como era de esperar, es que el caso más difícil se caracteriza por ser inaccesible, no episódico, dinámico y continuo.

Otro resultado será el hecho de que puesto que la mayoría de situaciones reales son tan complejas que el decidir si son realmente deterministas es tema de controversia; para efectos prácticos, se les deberá considerar como no deterministas.

En la tabla III se listan las propiedades de diversos ambientes que son familiares. Nótese que las respuestas podrán variar dependiendo de cómo conceptúe usted ambientes y agentes. Por ejemplo, el póquer es determinista cuando el agente está en la posibilidad de obtener información del orden de las cartas en el mazo, de lo contrario sería no determinista. Por otra parte, muchos ambientes son episódicos en niveles más altos que las acciones individuales de los agentes. Por ejemplo, un torneo de ajedrez consta de una secuencia de juegos: cada uno de ellos es un episodio, puesto que (en gran medida) la contribución de los movimientos en juego, al desempeño general del agente, no se ve afectada por las jugadas de la siguiente sesión. por otra parte, las jugadas de una misma sesión indudablemente que interactúan entre sí, y por ello es que el agente tiene que anticipar varias de ellas.

Tabla III. Ejemplos de ambientes y sus características.

Ambiente	Accesible	Determinista	Episódico	Estático	Discreto
Ajedrez con reloj	Si	Si	No	Semi	Si
Ajedrez sin reloj	Si	Si	No	Si	Si
Póquer	No	No	No	Si	Si
Backgammon	Si	No	No	Si	Si
Conducir un taxi	No	No	No	No	No
Sistema de diagnóstico Médico	No	No	No	No	No
Sistema de análisis de imágenes.	Si	Si	Si	Semi	No
Robot clasificador de partes	No	No	Si	No	No
Controlador de refinería	No	No	No	No	No
Asesor de inglés	No	No	No	No	Si

2.4.2. Programas de ambientes

El programa genérico de ambiente de la figura 19 ilustra la relación básica que existe entre agentes y ambientes. De esta forma, el ambiente se define por el estado inicial y la función de actualización. Desde luego, un agente que opera en un simulador deberá ser capaz también de operar en un ambiente real que le proporcione el mismo tipo de percepciones y que acepte el mismo tipo de acciones.

Figura 19. Programa básico del simulador de ambiente.

```
procedimiento PROBAR-AMBIENTE(estado, FUNCION-ACTUALIZACIÓN, agentes,
terminación)
entradas: estado, el estado inicial del ambiente
        FUNCION-ACTUALIZACIÓN, función para modificar el ambiente
        agentes, un conjunto de agentes
        terminación, un predicado para probar cuando se concluya.
repetir
        por cada agente dentro los agentes responden
                PERCEPCIÓN[agente]←OBTENER-PERCEPCIÓN(agente, estado)
        terminar
        por cada agente dentro los agentes responden
                ACCIÓN[agente]←PROGRAMA[agente](PERCEPCIÓN[agente])
        terminar
        estado←FUNCIÓN-ACTUALIZAR(acciones, agentes, estado)
hasta   terminación(estado)
```

El procedimiento PROBAR-AMBIENTE permite la adecuada ejercitación de los agentes en un ambiente. En el caso de algunos tipos de agentes, como los que participan en un diálogo de lenguaje natural, es suficiente con observar su conducta. Para obtener información más detallada acerca del desempeño de un agente, es necesario insertar algún tipo de código para medición de desempeño. La función RUN-EVAL-ENVIROMENT, mostrada en la figura 20 sirve para eso; aplica una medición de desempeño a cada agente y produce una lista de calificaciones obtenidas. Mediante la variable *calificaciones* se está al tanto de la calificación de cada uno de los agentes.

En general, lo que se use como medición de desempeño dependerá de la secuencia total de estados ambientales generados durante la operación del programa. Sin embargo, por lo general, la medida del desempeño se realiza a través de una simple acumulación, recurriendo ya sea a una adición, a un promedio o tomando un máximo. Por ejemplo, si la medida de desempeño de un agente que hace limpieza con aspiradora es la cantidad total de suciedad eliminada durante su turno, las calificaciones se limitarán a mantener al tanto de qué tanta mugre se ha eliminado hasta un momento dado.

Figura 20. Función RUN-EVAL-ENVIRONMENT.

Function RUN-EVAL-ENVIRONMENT(*state*, UPDATE-FN, *agents*, *termination*, PERFORMANCE-FN) **returns** *scores*
Función EJECUCIÓN-EVALUACIÓN(*estado*, FUNCIÓN-ACTUALIZAR, *agentes*, *terminación*, FUNCIÓN-DESEMPEÑO) **responde** con *calificaciones*
Variables locales: *calificaciones*, un vector del mismo tamaño que los agentes, todos 0

Repetir
Por cada *agente* **dentro** *los agentes* **responden**
PERCEPCIÓN[*agente*] ← HA OBTENER-PERCEPCIÓN(*agente,estado*)
Fin
Por cada *agente* **dentro** *los agentes* **responden**
ACCIÓN[*agente*] ← PROGRAMA[*agente*](PERCEPCIÓN[*agente*])
Fin
Estado ← FUNCIÓN-ACTUALIZAR(*acciones,agentes,estado*)
Calificaciones ← FUNCIÓN-DESEMPEÑO(*calificaciones,agentes,estado*)
Hasta *terminación(estado)*
Responde con *calificaciones*

La función RUN-EVAL-ENVIROMENT produce la medida de rendimiento correspondiente a un solo ambiente, definida como un solo estado inicial y una función de actualización en particular. Por lo general, los agentes se diseñan para que funcionen dentro de una clase ambiental, es decir, un conjunto de ambientes diversos.

Por ejemplo, el diseño de un programa de ajedrez capaz de enfrentarse a una amplia gama de oponentes humanos y máquinas. De haberlo diseñado para un solo oponente, habría sido posible aprovechar las debilidades específicas de tal oponente, pero no se habría logrado un buen programa para utilizarlo en cualquier juego. En rigor, para poder medir el desempeño de un agente es necesario contar con un generador de ambientes que seleccione ambientes particulares (con ciertas características parecidas) en los que se pueda probar el agente. Lo que interesa es el desempeño promedio del agente inmerso en determinada clase ambiental. La implantación de lo anterior se puede hacer relativamente sin mayor complicación en ambientes simulados.

Es posible que exista cierta confusión entre lo que es la variable de estado en el simulador ambiental y lo que es la variable de estado en el agente mismo (véase AGENTE-REFLEJO-CON-ESTADO). Cuando un programador tiene a cargo la tarea de implantar tanto el simulador de ambiente como el agente, se ve tentado a permitir que el agente fisgonee la variable de estado del simulador de ambiente. ¡A todo costo debe resistir la tentación! La versión que el agente proponga para el estado deberá construirse partiendo exclusivamente de sus percepciones, sin acceso alguno a toda la información del estado.

2.5. Resumen

Este capítulo ha sido una especie de agitado viaje de vista por la IA, considerada ésta como la ciencia del diseño de agentes. Los puntos más importantes que hay que recordar son los siguientes:

Un agente es algo que percibe y actúa en un ambiente. Internamente lo componen la arquitectura y el programa de agente.

El agente ideal es aquel que siempre emprende acciones que se esperan permitirán obtener la máxima medición de desempeño, tomando en cuenta la secuencia de percepciones hasta ese momento recibida

Los agentes se consideran autónomos en la medida que el tipo de acciones que eligen emprender dependan de su propia experiencia, no del conocimiento sobre el ambiente que les haya sido incorporado por el diseñador.

Mediante programa de agente se pasa de percepciones a acciones, al tiempo que se actualiza un estado interno.

Existe gran diversidad de diseños de programas de agente básicos, y su elección dependerá del tipo de información que se hace explícita y se utiliza en el proceso de decisión. Los diseños varían en cuanto eficiencia, concisión y flexibilidad. El adecuado diseño de un programa de agente dependerá de las percepciones, de las acciones y del ambiente.

Los agentes reflejos responden de inmediato a las percepciones, los agentes basados en metas actúan en función del logro de una meta y los agentes basados en utilidad se esfuerzan por obtener un máximo de “felicidad”.

El proceso de la toma de decisiones razonando con lo que se conoce, es fundamental en la IA y en el satisfactorio diseño del agente. Lo anterior es una indicación de la gran importancia de la representación del conocimiento.

Algunos ambientes son más dinámicos que otros. Los ambientes que ofrecen más retos son los inaccesibles, no deterministas, no episódicos, dinámicos y continuos.

3. PROCEDIMIENTOS PARA LA SOLUCIÓN DE PROBLEMAS

3.1. Introducción

En el capítulo anterior se vio que los agentes reflejos sencillos no son capaces de planificar con anticipación. Lo que pueden hacer es limitado, puesto que a sus acciones las determina sólo la percepción que existe en un momento dado. Además, ignoran el resultado producido por sus acciones y también lo que están esforzándose por lograr.

En este capítulo se presenta un tipo de agente basado en metas, denominado agente para la solución de problemas. Este tipo de agentes determina lo que deberán hacer al determinar secuencias de acciones que les permitan estados deseables. Se abordará el tema acerca de cómo deberá enfocar tal agente a los problemas. El tipo de problema que se derive como consecuencia del proceso de formulación dependerá del conocimiento que de dicho problema tenga el agente: básicamente, de si conoce su estado actual y el resultado de sus acciones. A continuación se definirán con más precisión aquellos elementos que constituyen el "problema" y su "solución", y serán ofrecidos varios ejemplos de tales definiciones. Una vez que se cuenta con definiciones precisas de un problema, se puede pasar relativamente de manera directa a la construcción de un proceso para la búsqueda de soluciones. Se explicarán seis estrategias de búsqueda y se mostrará cómo utilizarlas en una diversidad de problemas.

3.2. Solución de problemas mediante la búsqueda

3.2.1. Agentes que resuelven problemas

Los agentes inteligentes deben actuar de manera que el entorno experimente una serie de estados tales que permitan obtener un máximo en la medida de rendimiento. En términos generales, es difícil lograr la plena traducción de esta especificación en un satisfactorio diseño de un agente. Como se mencionó en el capítulo 2, el logro de este objetivo se simplifica cuando el agente tiene una meta y se esfuerza por lograrla.

Supóngase que el agente se encuentra en la ciudad de Arad, Rumania, casi al final de un viaje turístico durante sus vacaciones. El agente abordará un avión en Bucarest al día siguiente. El boleto respectivo no es reembolsable, la visa del agente está a punto de vencer y, a partir del día siguiente el de su partida, no habrá lugares en el avión durante seis semanas. En estas circunstancias, en la medida del desempeño intervienen muchos factores, además del costo del boleto y de no desear ser arrestado y deportado. Por ejemplo, el agente también quiere acentuar el bronceado de su piel, mejorar su conocimiento del rumano, visitar algunos sitios, etc. Los factores anteriores indican la existencia de toda una gama de posibles acciones que emprender. Pero, considerando la gravedad de la situación, debe fijarse como meta la de dirigirse conduciendo un auto a Bucarest. Toda aquella acción cuyas consecuencias impidan llegar a tiempo a Bucarest deberán descartarse sin mayor consideración. Este tipo de metas permiten organizar el curso de las acciones al limitar los objetivos que el agente se empeña en alcanzar. La formulación de metas, tomando como base la situación de un momento dado, es el primer paso en la solución de problemas. Además de lo anterior, el agente quizás quiera tomar una determinación en relación con otros factores que afectan lo deseable de las diversas maneras de alcanzar una meta.

Para efectos de los propósitos de este capítulo, se considerará que las metas son conjuntos de estados del mundo: Sólo aquellos que permiten el logro de la meta. Se puede considerar que las acciones son las causantes de la transición entre uno y otro estado del mundo; por ello, obviamente el agente tiene que determinar qué acciones permiten obtener el estado correspondiente a una meta. Pero, para ello, primero tiene que decidir qué tipos de acciones y estados habrá que tomar en consideración. Si sólo se manejaran acciones del tipo de "desplace el pie izquierdo hacia adelante 36 centímetros" o "gire el volante seis grados a la izquierda", no sería posible ni siquiera salir del estacionamiento, no se diga llegar hasta Bucarest, pues el construir una solución con tal grado de detalle constituiría un problema intratable.

La formulación de un problema es el proceso que consiste en decidir que acciones y estados habrán de considerarse y es el paso que sigue a la formulación de metas. Se hablará con más detalle de este proceso. Por ahora, considérese que las acciones que el agente emprenderá serán del tipo de trasladarse de una población grande a otra. Es decir, el tipo de estados que tomará en cuenta corresponden al estar en una población determinada.

La meta del agente en cuestión es ahora la de dirigirse en auto a *Bucarest* y delibera a qué pueblo ir desde *Arad*. Hay tres caminos que salen de esta población: uno hacia *Sibiu*, otro a *Timisoara* y otro más a *Zerind*. Por ninguno de ellos se puede alcanzar la meta, por lo que, a menos que el agente no esté familiarizado con al geografía de *Rumania*, ignorará qué camino tomar. Es decir, el agente no sabe cuál de sus posibles acciones podría ser la mejor puesto que carece del suficiente conocimiento del estado producido como consecuencia de cada una de las acciones. De no contar con mayor conocimiento, el agente quedará en un callejón sin salida. Lo mejor que puede hacer es decidirse al azar por una de las acciones.

Pero, supóngase ahora que el agente cuenta con un mapa de *Rumania*, sea en el papel o en la memoria, el propósito del mapa es dotar al agente de información acerca de los diversos estados en los que podría encontrarse, así como sobre las acciones que puede emprender. El agente puede apoyarse en esta información para evaluar las posteriores etapas de un viaje hipotético por cada uno de los tres poblados, con el fin de encontrar una ruta que finalmente le lleve a *Bucarest*. Una vez que logra encontrar en el mapa una ruta que vaya de *Arad* a *Bucarest*, procede a lograr su meta conduciendo el auto a través de los diversos tramos de dicha ruta. Es decir, en términos generales, cuando un agente tiene ante sí diversas opciones inmediatas cuyo valor ignora, para decidir lo que debe hacer primero tiene que evaluar las diversas secuencia de acciones posibles que le conducen a estados cuyo valor se conoce, y luego decidirse por la mejor. Al anterior proceso de tratar de hallar tal secuencia se le conoce como búsqueda.

En un algoritmo de búsqueda la entrada es un problema y la respuesta es una solución que adopta la forma de una secuencia de acciones. Una vez encontrada una solución, se procede a ejecutar las acciones que ésta recomienda. A la anterior se le denomina fase de ejecución. Es decir, el diseño del agente se reduce simplemente a "formular, buscar y ejecutar", como se puede observar en la figura 21. Luego de formular una meta y el problema que hay que resolver, el agente solicita un procedimiento de búsqueda que le permita resolverlo. Procede a utilizar la solución para guiar sus acciones, y así hace todo lo que la solución le indica como el siguiente paso que se debe dar y, hecho esto, procede a eliminar este paso de la secuencia. Una vez ejecutada la solución, el agente busca una nueva meta.

Figura 21. Agente sencillo para la solución de problemas.

función AGENTE-SENCILLO-PARA LA SOLUCIÓN DE PROBLEMAS(*p*) **responde** con una acción
entradas: *p*, una percepción.
estático: *s*, una secuencia de acciones, originalmente vacía.
estado, una descripción del estado actual del mundo.
g, una meta, originalmente nula.
problema, la formulación de un problema.
estado ← ACTUALIZAR-ESTADO(*estado*,*p*)
si *s* está vacío, **entonces**
g ← FORMULAR-META(*estado*)
problema ← FORMULAR-EL-PROBLEMA(*estado*,*g*)
s ← BUSCAR(*problema*)
acción ← RECOMENDACIÓN(*s*,*estado*)
s ← RESTO(*s*,*estado*)
responder con *acción*

En este capítulo no se hablará más de las funciones ACTUALIZAR-ESTADO y FORMULAR-META. En las dos secciones siguientes se explica el proceso de la formulación de un problema, y la parte restante del capítulo se dedica a las diversas versiones de la función BUSCAR. Por lo general la fase de la ejecución es bastante sencilla en el caso de un agente para la solución de problemas sencillo: con base en la RECOMENDACIÓN se emprende la primera acción de la secuencia y PARTE RESTANTE devuelve lo que queda.

3.2.2. Formulación de problemas

En esta sección se abordará con mayor detalle el proceso de la formulación de un problema. Se verá primero los distintos grados de conocimiento que un agente puede tener de sus acciones y del estado en el que este agente se encuentra. Lo anterior dependerá de cómo está relacionado el agente con su ambiente a través de sus percepciones y acciones. Se verá que, básicamente, son cuatro los diversos tipos de problemas: problemas de un solo estado, problemas de estado múltiple, problemas de contingencia y problemas de exploración. Se dará una definición precisa de estos tipos, en preparación de lo que se abordará en secciones posteriores, en donde se trata lo referente al proceso de solución.

3.2.2.1. Conocimiento y tipos de problemas

Considérese ahora un ambiente distinto al de Rumania: el mundo de una aspiradora. Para facilitar la explicación, se le simplificará aún más. En este mundo hay dos posibles ubicaciones. En ellas puede o no haber mugre y el agente se encuentra en una de las dos. El mundo puede asumir ocho posibles estados, como se muestra en la figura 22. Las tres acciones que el agente puede emprender en esta versión del mundo de la aspiradora son: a la Izquierda, a la Derecha y Aspirar.

Por el momento, supóngase que la eficiencia del aspirado es de 100%. La meta es eliminar toda la mugre. Es decir, la meta equivale al conjunto de estados {7,8}.

Figura 22. Los ocho posibles estados del mundo simplificado de la aspiradora.

Para empezar supóngase que los sensores del agente le proporcionan información suficiente como para determinar con exactitud el estado en el que se encuentre (es decir, si el mundo es accesible); supóngase que conoce con exactitud el resultado producido por cada una de sus acciones. Por lo tanto, podrá calcular con exactitud en qué estado se encontrará luego de una determinada secuencia de acciones. Por ejemplo, si su estado inicial es 5, será capaz de calcular que la secuencia de acciones [*Derecha, Aspirar*] le permitirá alcanzar el estado meta. El anterior es el caso más sencillo, al que se denomina problema de un solo estado.

Segundo, supóngase que el agente sabe cuál es el resultado de cada una de sus acciones pero que su acceso al estado del mundo es limitado. Por ejemplo, el caso extremo sería el de carecer por completo de sensores. En este caso, sólo sabe que su estado inicial es uno de los del conjunto {1,2,3,4,5,6,7,8}. Aunque pudiera parecer que el agente no tiene ayuda, en realidad puede arreglárselas bastante bien. Gracias a que sabe cuál es el efecto de sus acciones, puede, por ejemplo, calcular que la acción *Derecha* le conducirá a uno de los estados {2,4,6,8}. De hecho, el agente puede descubrir que la secuencia de acciones [*Derecha, Aspirar, Izquierda, Aspirar*] le garantiza alcanzar un estado meta, independientemente de cual sea el estado inicial.

Resumiendo: si el mundo no es totalmente accesible, el agente deberá razonar en términos de los conjuntos de estados a los que puede llegar, en vez de pensar en función de estados únicos. A lo anterior se acostumbra a denominar **problema de estado múltiple**.

Aunque pudiera parecer diferente, el caso de cuando se ignoran los efectos de las acciones puede ser manejado de manera similar. Supóngase, por ejemplo, que el ambiente al parecer no es determinista puesto que sigue la ley de *Murphy*: la acción denominada *Aspirar*, a veces deja mugre en la alfombra, pero solamente sin en ésta no había mugre. Por ejemplo, si el agente sabe que está en el estado 4, sabrá que si aspira llegará a uno de los estados {2,4}. A cada estado inicial conocido le corresponde una secuencia de acciones que garantiza el logro de un estado meta.

Hay veces que la ignorancia impide al agente encontrar una secuencia de solución garantizada. Por ejemplo, si el agente está en el mundo de la ley de *Murphy*, cuenta con un sensor de ubicación y un sensor local de mugre, pero no tiene un sensor que le permita detectar mugre en otros cuadrantes. Además, los sensores le dicen que está en uno de los estados {1,3}. El agente podría formular la secuencia de acciones [*Aspirar, Derecha, Aspirar*]. El aspirar modificaría el estado al de {5,7} y el desplazarse a la derecha modificaría el estado a uno de {6,8}. Si en realidad está en el estado 6, la secuencia de acciones tendrá éxito, pero si se trata del estado 8, el plan fracasará. Si el agente hubiera escogido la secuencia de acción más sencilla [*Aspirar*], tendría éxito a veces, pero no siempre. Resulta así que no hay una secuencia fija de acciones que garantice la solución de este problema.

Obviamente, el agente sí tiene una forma de resolver el problema empezando del {1,3}: primero aspirar, luego desplazarse a la derecha, luego aspirar sólo si allí está sucio. Es decir, para resolver este problema es necesario utilizar el sensor durante la fase de ejecución. Nótese que el agente ahora debe calcular todo un árbol de acciones, en vez de una sola secuencia de ellas. Por lo general, cada una de las ramas del árbol se refiere a una posible contingencia que pudiera surgir. Por ello, al anterior se le denomina problema de contingencia. Muchos de los problemas el mundo real, físico, son problemas de contingencia, puesto que es imposible hacer predicciones exactas. Esta es la razón por la que la mayoría de las personas presten bastante atención cuando va caminando o al conducir.

Los problemas de un sólo estado y los de estado múltiple se resuelven con las mismas técnicas de búsqueda, de las que se hablará en éste y en el siguiente capítulo. En los problemas de contingencia, por otra parte, es necesario emplear algoritmos más complejos. Por otra parte, se prestan a un diseño de agente un poco distinto, en el que el agente está en posibilidad de actuar antes de encontrar un plan garantizado. Lo anterior es muy útil, porque en vez de considerar por anticipado toda posible contingencia que pueda surgir durante la ejecución, muchas veces es mejor proceder a la ejecución y ver qué contingencias van surgiendo. Con base en la información adicional, el agente puede seguir resolviendo el problema.

Por último, considérese el predicamento de un agente que no cuenta con información sobre los efectos de sus acciones. Constituye la tarea más difícil con la que se enfrenta un agente inteligente. El agente tendrá que experimentar, descubrir poco a poco qué tipo de acciones emprender y que tipo de estados existen. La anterior es un tipo de búsqueda, pero en el mundo real, no en un modelo. Ejecutar un paso en el mundo real, en vez de hacerlo en un modelo entraña grandes peligros para un agente ignorante. Si logra sobrevivir, el agente aprenderá un “mapa” del ambiente, mapa que puede utilizar para resolver otros problemas que se le presenten.

3.2.2.2. Problemas bien definidos y soluciones

Un problema en realidad es un conjunto de información que el agente utiliza para decidir lo que va a hacer. Se comenzará primero por especificar qué información se necesita para definir un problema de un solo estado. Se ha visto ya que los elementos básicos que configuran la definición de un problema son los estados y las acciones. Para capturarlos formalmente necesitamos lo siguiente:

- El **estado inicial**, que el agente sabe que allí es donde se encuentra.

- El conjunto de las posibles acciones que el agente puede emprender. El término **operador** denota la descripción de una acción en función de la cual se alcanzará un estado al emprender una acción en un estado particular. (Otra forma de definirlo es mediante la **función subsecuente** S. Teniendo un estado particular x, S(x) responde con un conjunto de estados obtenibles a partir de x mediante una sola acción).

- Los dos términos anteriores definen el **espacio de estados** del problema: el conjunto de todos los estados que pueden alcanzarse a partir del estado inicial mediante cualquier secuencia de acciones que permiten pasar de un estado a otro. El siguiente elemento de un problema es:

- La **prueba de meta**, es lo que el agente aplica a la descripción de un solo estado para decidir si se trata de un estado meta. Hay ocasiones en las que existe un conjunto explícito de posibles estados meta, y la prueba sólo sirve para cerciorarse si se ha alcanzado alguno de ellos. A veces la meta se especifica mediante una propiedad abstracta, en vez de un conjunto d estados enumerados explícitamente. Por ejemplo, en el ajedrez la meta es alcanzar un estado denominado "jaque mate", en el que el rey contrincante se capturará en la siguiente jugada, haga lo que haga éste.

- Por último hay casos en los que una solución es preferible a la otra, no obstante que con ambas pueda lograrse la meta. Por ejemplo, son preferibles las rutas que entrañan menos acciones o acciones menos costosas. La función **costo de ruta** es una función mediante la que se asigna un costo a una ruta determinada. En todos los casos que consideremos, este costo es la suma de los costos de cada una de las acciones individuales que se emprendan a lo largo de la ruta. Esta función se denota como g.

Conjuntamente, el estado inicial, el conjunto del operador, la prueba de meta y la función costo de ruta definen un problema. Ahora, desde luego, es posible definir un machote de datos que sirva para representar problemas:

Machote PROBLEMA
Componentes: INICIAL-ESTADO,OPERADORES,META-EXAMEN,COSTO-RUTA-FUNCIÓN

Los casos específicos de este machote de datos serán la entrada de los algoritmos de búsqueda. La salida producida por un algoritmo de búsqueda se denomina **solución**, es decir, una ruta que va del estado inicial al estado que satisface la prueba de meta.

En el caso de los problemas de estado múltiple sólo hay que hacer algunas ligeras modificaciones: un problema está formado por un conjunto de estados iniciales; un conjunto de operadores especifican cada una de las acciones del conjunto de estados que se alcanza a partir de un estado determinado; la prueba de meta y la función del costo de ruta siguen funcionando como antes. Se aplica un operador a un conjunto de estados mediante la unión de los resultados obtenidos al aplicar el operador a cada uno de los estados del conjunto. Ahora mediante una ruta se conectan los *conjuntos* de estados y la solución es ahora una ruta que conduce a un conjunto de estados *todos los cuales* son estados meta. El espacio de estados se sustituye con un **espacio de conjunto de estados.**

3.2.2.3. Cómo medir la eficiencia para resolver problemas

Existen por lo menos tres formas de medir la eficiencia de una búsqueda. Primera: ¿permite encontrar una solución? Segunda: ¿es una buena solución (con un bajo costo de ruta)? Tercera: ¿cuál es el **costo de búsqueda** correspondiente al tiempo y memoria necesarios para encontrar una solución? El **costo total** de la búsqueda es la suma del costo de la ruta y el costo de la búsqueda.

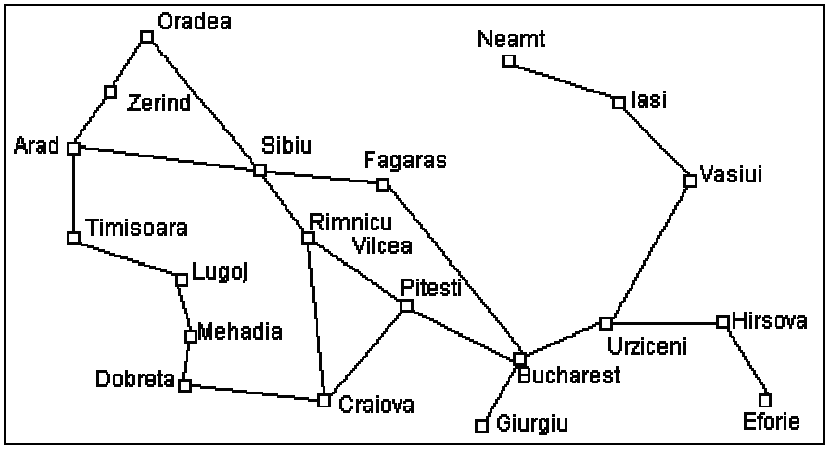
En el caso del problema de encontrar una ruta que vaya de Arad a Bucarest, el costo de la ruta podría ser proporcional al kilometraje total de la ruta, quizás con algo que incorpore el desgaste producido por las diversas superficies de las carreteras. El costo de búsqueda dependerá de las circunstancias. En un ambiente estático, será igual a cero puesto que la medición del desempeño es independiente del tiempo. Si hubiera cierta prisa por llegar a Bucarest, el ambiente sería semi-dinámico puesto que el tardarse más tiempo deliberando implicará más costo. En este caso, el costo de búsqueda podría variar aproximadamente en forma lineal con el tiempo requerido en los cálculos (al menos, en pequeñas cantidades de tiempo). Así al calcular el costo total tendríamos que sumar kilómetros y milisegundos.

Lo anterior no siempre es sencillo, pues no existe un "tipo de cambio oficial" para ambas unidades. El agente tendrá que ver alguna forma que le permita decidir qué recursos invertir en la búsqueda y qué recursos dedicar a la ejecución. En el caso de problemas cuyos espacios de estados son muy pequeños, no es difícil encontrar la solución que implica el mínimo costo de ruta. Sin embargo, en el caso de problemas grandes y complicados, hay que hacer un compromiso: o el agente busca por un tiempo prolongado para así obtener una solución óptima, o el agente busca durante un tiempo más corto y obtiene una solución, aunque con un costo de ruta un poco mayor.

3.2.2.4. Como escoger estados y acciones

Una vez, concluido lo referente a definiciones, se empezará con la investigación sobre los problemas a través de uno muy sencillo: "ir en auto de Arad a Bucarest utilizando las carreteras del mapa de la figura 23". En un espacio de estados adecuado hay 20 estados, cada uno de los cuales se define exclusivamente por ubicación, especificada como una ciudad. Por lo tanto, el estado inicial es "en Arad" y la prueba de meta es "¿aquí es Bucarest?" Los operadores consisten en manejar por las carreteras entre una ciudad y otra.

Figura 23. Un mapa simplificado de Rumania.



Una solución es la ruta que va de *Arad* a *Sibiu*, a *Rimmicu Vilcea*, a *Pitesti* y a *Bucarest*. Son muchas las otras rutas que también son soluciones por ejemplo, vía *Lugoj* y *Craiova*. Para decidir cuál de estas soluciones es la mejor, hay que saber qué es lo que se está midiendo con la función costo de trayectoria: puede ser el kilometraje total o el tiempo de recorrido estimado. Dado que en el mapa con el que contamos no se especifica ninguna de las dos, como función de costo se utilizará la cantidad de etapas. Así pues, la ruta que va de *Sibiu* a *Fagaras*, cuyo costo de ruta es 3, es la mejor de las soluciones.

El verdadero arte de la solución de problemas consiste en saber decidir qué es lo que servirá para describir los estados y operadores y qué no. Compárese la sencilla descripción de estado que se ha elegido, "en *Arad*", con un viaje real a través del país, caso éste en el que el estado del mundo está integrado por muchas cosas: los compañeros de viaje, la velocidad de tal vehículo, si hay policías de caminos en las proximidades, la hora del día, si el chofer tiene hambre o está cansado, si ya se está acabando la gasolina, cuánto falta para la próxima parada para descansar, las condiciones de la carretera y del tiempo. En las descripciones de estado no se toman en cuenta todas las consideraciones anteriores, puesto que son irrelevantes en el problema de encontrar una ruta que lleve a *Bucarest*. El proceso de la eliminación de detalles de una representación se denomina **abstracción**.

Así como se abstrae en el caso de la descripción de estado, también hay que hacerlo en el caso de las acciones. Una acción produce efectos diversos. Además de modificar la ubicación del vehículo y de sus ocupantes, consume tiempo y combustible, genera contaminación y cambia al agente. En nuestra formulación sólo tomamos en cuenta el cambio en ubicación. Por otra parte, hay muchas acciones que se omiten: prender el radio, mirar por la ventana, disminuir la velocidad cuando se pasa ante la policía de caminos, etc.

¿Se Podría precisar más en cuanto a la definición del nivel adecuado de abstracción? Considérese que los estados y acciones elegidas corresponden a conjuntos de estados del mundo y conjuntos de secuencias de acciones detallados. Ahora considérese una solución al problema abstracto: por ejemplo la ruta que va de *Arad* a *Sibiu*, *Rimnicu Vilcea*, *Pitesti* y *Bucarest*. Esta solución pertenece a una gran cantidad de rutas más detalladas. Por ejemplo, podríamos viajar con el radio prendido entre *Sibiu* y *Rimnicu Vilcea* y luego apagarlo por todo el resto del viaje. Cada una de este tipo de rutas más detalladas sigue siendo una solución para la meta, por lo que la abstracción es válida.

La abstracción también es útil puesto que el emprender cada una de las acciones de la solución, como sería manejar de *Pitesti* a *Bucarest*, es un poco más fácil que el problema original. Así pues, para escoger una buena abstracción hay que eliminar todos los detalles que sea posible siempre y cuando se conserve la validez y se garantice que es fácil emprender las acciones abstractas. Si no fuera por su capacidad para construir abstracciones útiles, el mundo real dejaría abrumados a los agentes inteligentes.

3.3. Búsqueda de soluciones

Se ha visto ya cómo definir un problema y cómo identificar una solución. La parte restante, la búsqueda de la solución, se logra mediante una búsqueda realizada a través del espacio de estados. La idea consiste en mantener y ampliar un conjunto de secuencias de solución parciales. En esta sección se indica cómo generar tales secuencias y cómo mantenerse al tanto de éstas a través de las estructuras de datos adecuadas.

3.3.1. Generación de secuencias de acciones

Para resolver el problema de la determinación de la ruta de *Arad* a *Bucarest*, por ejemplo, empezaremos por el estado inicial, *Arad*. Primero hay que determinar si éste es un estado meta. Es evidente que no es así, pero es importante probarlo y estar así en condiciones de resolver engañosos problemas como "partir de *Arad* para llegar a *Arad*". Puesto que el anterior no es un estado meta, habrá que evaluar otros estados. Para ello hay que aplicar los operadores al estado de un momento determinado, con lo que se genera un nuevo conjunto de estados. Al proceso anterior se le denomina **expansión** del estado.

En nuestro ejemplo hay tres nuevos estados: "en *Sibiu*", "en *Timisoara*" y "en *Zerind*", puesto que existe una ruta directa de un solo trayecto que va de Arad a las tres ciudades anteriores. Si sólo hubiera una posibilidad, bastaría con optar por ella y continuar; pero cuando las posibilidades son múltiples hay que escoger cuál es la que conviene.

En esto radica la búsqueda: en escoger una opción, haciendo a un lado las demás para considerarlas posteriormente en caso de no obtener respuesta alguna mediante la primera opción. Supóngase que se escoge *Zerind*. Se verifica si se trata de un estado meta (lo que no es), y luego se le expande para llegar a "*Arad*! Y a "*Oradea*". Se debe escoger ahora entre una de éstas, o regresar y elegir una entre *Sibiu* o *Timisoara*. Se sigue eligiendo, probando y expandiendo, hasta dar con la solución, o hasta que ya no haya más estados que expandir. La elección del estado que se desea expandir primero se realiza a través de una **estrategia de búsqueda**.

Para facilitar la comprensión, conviene concebir el proceso de búsqueda como si fuera la construcción de un **árbol de búsqueda** sobrepuesto al espacio de estados. La raíz del árbol de búsqueda es el **nodo de búsqueda** y corresponde al estado inicial. Los nodos hoja del árbol corresponden a estados que no tienen ningún sucesor en el árbol, ya sea porque todavía no se han expandido, o porque ya lo fueron pero generaron un conjunto vacío. En cada paso, el algoritmo de búsqueda escoge un nodo hoja y lo expande. En la figura 24 se muestra parte de las expansiones del árbol de búsqueda obtenidas durante la búsqueda de una ruta que vaya de *Arad* a *Bucarest*. El algoritmo general de búsqueda está detallado de manera informal en la figura 25.

Es importante tener clara la diferencia entre el espacio de estados y el árbol de búsqueda. En el problema de la determinación de la ruta, en el espacio de estados sólo hay 20 de éstos, uno por cada ciudad, en tanto que la cantidad de rutas en este mismo espacio es "infinito", por lo que la cantidad de nodos del árbol de búsqueda también es "infinita". Por ejemplo, en la figura 24, la rama que *Arad-Sibiu-Arad* continúa con *Arad-Sibiu-Arad-Sibiu-Arad*, y así, indefinidamente. Es obvio que un buen algoritmo de búsqueda evitará rutas como la anterior.

Figura 24. Árbol de búsqueda parcial para encontrar una ruta que vaya de *Arad* a *Bucarest*.

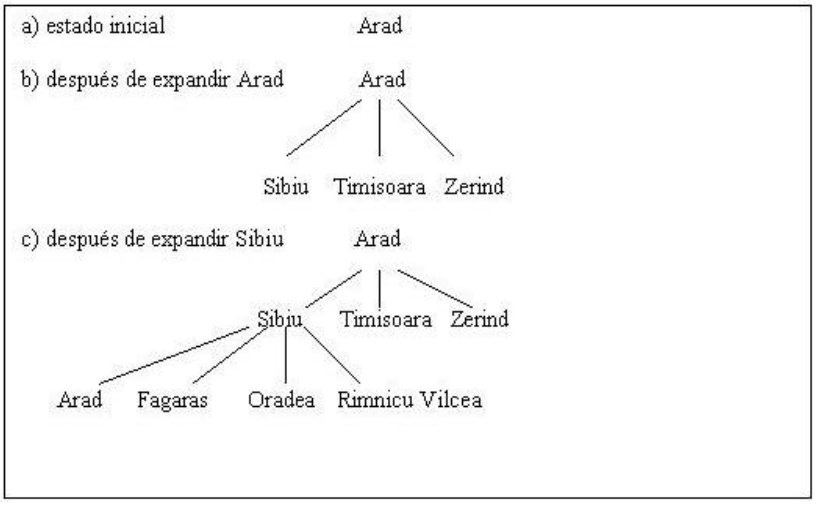


Figura 25. Descripción informal del algoritmo general de búsqueda.

```
función BÚSQUEDA-GENERAL(problema,estrategia) produce una solución o falla
  inicializa el árbol de búsqueda empleando el estado inicial del problema
  bucle hacer
    si no hay candidatos para la expansión, responda con falla
    escoja un nodo hoja para hacer la expansión, de conformidad con la estrategia
    si el nodo contiene un estado meta, responda con la solución respectiva
    o bien expanda el nodo y añada los nodos resultantes al árbol de búsqueda
  fin
```

3.3.2. Estructuras de datos para los árboles de búsqueda

Son muchas las formas de representar a los nodos, pero en este capítulo se supondrá que un nodo es una estructura de datos con cinco componentes:

- El estado en el espacio de estados al que corresponda el nodo;
- El nodo del árbol de búsqueda que generó este nodo (denotado **nodo padre**);
- El operador que se aplicó para generar el nodo;
- Cuántos nodos de la ruta hay desde la raíz hasta dicho nodo; (la **profundidad** del nodo);
- El costo de ruta de la ruta que va del estado inicial al nodo.

Así pues, la tipología de los datos del nodo es la siguiente:

Nodo de **machote**

Componentes: ESTADO, NODO-PADRE, OPERADOR, PROFUNDIDAD, COSTO DE RUTA

Es importante tener presente la diferencia entre nodos y estados. Los primeros son estructuras de datos contables que sirven para representar el árbol de búsqueda de un problema en particular, estructura generada mediante un algoritmo en particular. Los segundos representan una configuración (o conjunto de configuraciones) del mundo. Es decir, los nodos tienen profundidades y padres, en tanto que los estados, no. La función EXPANDIR se encarga de calcular cada una de las componentes de los nodos que éste genera.

También es necesario representar al grupo de nodos que están en espera de ser expandidos: se les conoce como el **margen** o **frontera**. La más sencilla de las representaciones sería la de un conjunto de nodos. La estrategia de búsqueda sería una función mediante la que se escoge el siguiente nodo de este conjunto que se va a expandir. Si bien teóricamente lo anterior es un procedimiento directo, su cálculo podría resultar costoso, puesto que mediante la función de estrategia habría que examinar cada uno de los elementos del conjunto y así escoger el mejor. De acuerdo con lo anterior, se supondrá que el grupo de nodos se implanta como si se tratara de una **lista de espera**. Las operaciones de una lista de espera son las siguientes:

- HACER-LISTA DE ESPERA (Elementos) crea una lista de espera con base en los elementos dados.
- ¿ESTA VACÍA? (Lista de espera) contestará afirmativamente sólo cuando ya no haya más elementos en la lista.
- QUITAR-REFERENTE (Lista de espera) elimina el elemento que encabeza la lista y lo devuelve.
- FUCNIÓN-PNER EN LISTA (Elementos, lista de espera) pone un conjunto de elementos en la lista.

Cada variedad de la función lista de espera producirá distintas variedades del algoritmo de búsqueda.

Con la ayuda de las definiciones anteriores ahora es posible dar una versión más formal del algoritmo general de búsqueda. Lo anterior aparece en la figura 26.

Figura 26. Descripción informal del algoritmo general de búsqueda.

Función BÚSQUEDA-GENERAL(*problema,* FUNCIÓN-LISTA DE ESPERA) **responde** con una solución o falla.
  Nodos ← HACER-LISTA DE ESPERA(HACER-NODO(ESTADO-INICIAL[*problema*]))
  Bucle hacer
    Si *nodos* está vació, **conteste** con falla
    Nodo ← ELIMINAR-DEL FRENTE(*nodos*)
    Si PRUEBA-META[*problema*] se aplica a ESTADO(*nodo*) y se tiene éxito **contestar** con *nodo*
    Nodos ← FUNCIÓN LISTA DE ESPERA(nodos, EXPANDIR(nodo,OPERADORES[*problema*]))
  fin

3.4. Estrategias de búsqueda

Buena parte de los esfuerzos invertidos en el área de la búsqueda han quedado en la determinación de la **estrategia de búsqueda** adecuada para un problema. Al estudiar este campo se evaluarán las estrategias en función de los cuatro criterios siguientes:

- **Completez:** ¿La estrategia garantiza encontrar una solución, si es que ésta existe?
- **Complejidad temporal:** ¿Cuánto tiempo se necesitará para encontrar una solución?
- **Complejidad espacial:** ¿Cuánta memoria se necesita para efectuar la búsqueda?
- **Optimidad:** ¿Con esta estrategia se encontrará una solución de la más alta calidad. En caso de que existan varias soluciones?

En esta sección se presentan seis estrategias de búsqueda agrupadas genéricamente bajo el nombre de **búsqueda sin contar con información**. Es decir, no existe información acerca de la cantidad de pasos necesarios o sobre el costo de ruta para pasar del estado de un momento dado a la meta: lo único que permiten hacer es diferenciar entre un estado meta del otro que no lo es. A la búsqueda sin contar con información también se le conoce como búsqueda ciega.

Considérese nuevamente el problema sobre la determinación de una ruta. Partiendo del estado inicial en *Arad*, hay tres acciones que conducen a tres nuevos estados: *Sibiu*, *Timisoara* y *Zerind*. Una búsqueda que no cuenta con información no tendrá preferencia entre éstas; un agente más listo se daría cuenta de que la meta, *Bucarest,* se encuentra al sudeste de *Arad,* y que sólo *Sibiu* se encuentra en esa dirección, por lo que es muy probable que sea la mejor opción.

Las estrategias en donde se hace este tipo de consideraciones se denominan estrategias de **búsqueda respaldada con información** o estrategias de **búsqueda heurística**. Obviamente, la búsqueda sin contar con información es menos eficiente que la búsqueda que cuenta con información. Con todo, la primera es importante, ya que son muchos los problemas en los que no se puede contar más que con cierta información.

Las seis estrategias de búsqueda sin contar con información se diferencian entre sí por el *orden* en el que se realiza la expansión de los nodos. Esta diferencia es determinante, como se verá en breve.

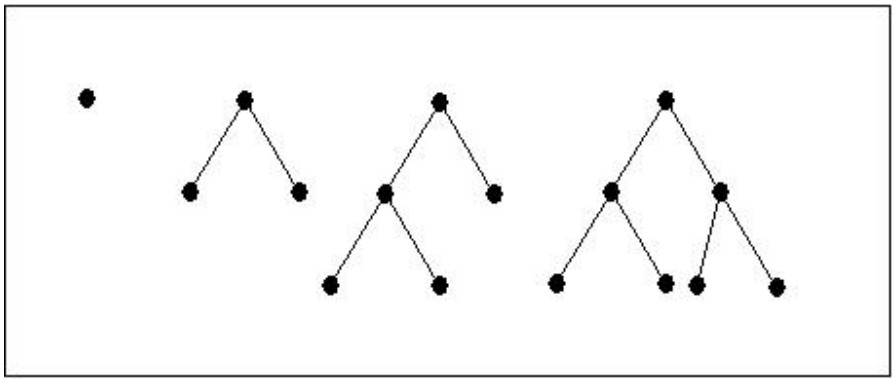
3.4.1. Búsqueda preferente por amplitud

Una de las estrategias más sencillas es la de la búsqueda preferente por amplitud. En este caso, primero se expande el nodo raíz, y luego todos los nodos generados por éste; luego, sus sucesores, y así sucesivamente. En general, todos los nodos que están en la profundidad *d* del árbol de búsqueda se expanden antes de los nodos que estén en la profundidad *d*+1. Para implantar una búsqueda preferente por amplitud se utiliza un algoritmo BÚSQUEDA-GENERAL con una función de lista de espera mediante la que se van poniendo los estados recién generados al final de la lista, a continuación de todos los estados generados previamente:

Function BREADTH-FIRST-SEARCH(*problem*) ***return*** a *solution or failure* **Función** BÚSQUEDA-PREFERENTE-POR-AMPLITUD(problema) **responde** con una solución o una falla **Responde** con BÚSQUEDA-GENERAL(problema) EN-LISTA DE ESPERA AL FINAL

La de búsqueda preferente por amplitud es una estrategia bastante sistemática, pues primero toma en cuenta todas las rutas de longitud 1, luego las de longitud 2, etc. en la figura 27 se muestra el avance de la búsqueda efectuada en un sencillo árbol binario. En cada caso de haber solución, es seguro que ésta se encontrará mediante la búsqueda preferente por amplitud; si son varias las soluciones, este tipo de búsqueda permitirá siempre encontrar primero el estado meta más próximo. En función de los cuatro criterios, la búsqueda preferente por amplitud es muy completa y óptima, *siempre y cuando el costo de ruta sea una función que no disminuya al aumentar la profundidad del nodo.*

Figura 27. Árboles de búsqueda preferente por amplitud.



Hasta ahora las noticias sobre la búsqueda preferente por amplitud han sido alentadoras. Pero, para mostrar por qué no siempre es la estrategia de opción, considérese la cantidad de tiempo y memoria necesarias para realizar una búsqueda. Tómese en cuenta un espacio de estados hipotético, en el que la expansión de cada uno de los estados produce *b* nuevos estados. Se dice que el **factor de ramificación** de estos estados (y del árbol de búsqueda) es *b*. La raíz del árbol de búsqueda genera *b* nodos en el primer nivel, cada uno de los cuales genera *b* nodos más, con un total de b^2 en el segundo nivel. Cada uno de éstos genera *b* nodos más, y producen b^3 nodos en el tercer nivel, etc. Supóngase ahora que en la solución de este problema hay una ruta de longitud *d*. Así pues, la cantidad máxima de nodos expandidos antes de poder encontrar una solución es

$$1+b+b^2+b^3+.....+b^d$$

Esta es la cantidad máxima, aunque la solución podrá aparecer en cualquiera de los puntos del *d*-avo nivel. En el mejor de los casos, la cantidad será menor a éste.

Aquellos familiarizados con el análisis se ponen nerviosos (o emocionados si son personas que les gustan los retos) cuando aparece una relación de complejidad exponencial como O(b^d). En la figura 28 podrá observarse el porqué. Allí se muestra el tiempo y memoria necesarios para una búsqueda preferente por amplitud cuyo factor de ramificación es b=10 y para varios valores de profundidad de solución d. La complejidad del espacio es igual a la complejidad temporal, dado que todos los nodos hojas del árbol deben mantenerse en la memoria al mismo tiempo.

De la figura 28 se pueden aprender dos lecciones. Primera: *en el caso de la búsqueda preferente por amplitud, el consumo de memoria es un problema más serio que el del tiempo de ejecución.*

Figura 28. Tiempo y memoria necesarios en la búsqueda preferente por amplitud.

Profundidad	Nodos	Tiempo	Memoria
0	1	1 milisegundo	100 byte
2	111	.1 segundos	11 kbytes
4	11,111	11 segundos	1 Mb
6	10000000	18 segundos	111 Mb
8	1000000000	31 horas	11 GB
10	1E+11	128 días	1 TB
12	1E+13	35 años	111 TB
14	1E+15	3500 años	11,111 TB

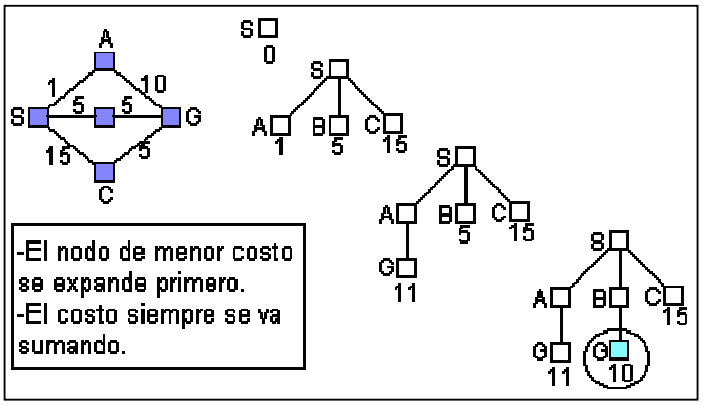
La segunda lección es que el tiempo necesario sigue siendo un factor determinante. Si un problema tiene solución en la profundidad 12, con base en lo supuesto, se necesitarían 35 años para encontrarla a través de una búsqueda sin información. E*n general, los problemas de búsqueda de complejidad exponencial son irresolubles, sino en las instancias más pequeñas.*

3.4.2. Búsqueda de costo uniforme

Mediante la búsqueda preferente por amplitud se encuentra el estado meta más próximo a la superficie, sin embargo ésta no siempre es la solución de costo mínimo de la función general de costo de ruta. En el caso de la **búsqueda de costo uniforme** se modifica la estrategia preferente por amplitud en el sentido de expandir siempre el nodo de menor costo en el margen (medido por el costo de la ruta g(n)) en vez del nodo de menor profundidad. No es difícil darse cuenta de que la búsqueda preferente por amplitud no es sino una búsqueda de costo uniforme en la que g(n)=PROFUNDIDAD(n).

Si se cumplen ciertas condiciones, es seguro que la primera solución encontrada será la más barata, puesto que si hubiera una ruta más barata que fuese solución, ya se habría expandido anteriormente y ya habría sido encontrada. Esto se comprenderá más fácilmente considerando una estrategia en plena acción. Tómese el caso del problema de la determinación de la ruta de la figura 29. El problema radica en ir de S a G, y va marcándose el costo de cada uno de los operadores. Con esta estrategia primero se expande el estado inicial, lo que produce rutas en dirección a A, B y C. Como la ruta que lleva a A es la más barata, se procede a expandirla y así generar la ruta SAG, que de hecho es una solución, si bien no es la óptima. Sin embargo, el algoritmo no admite a ésta como solución, ya que cuesta 11 y se le deja en la lista de espera debajo de la ruta SB, cuyo costo es de 5. Es una pena crear una solución para dejarla olvidada en la lista, pero así debe ser si lo que se desea es encontrar la solución óptima, no cualquier solución. El paso siguiente es expandir SB, generar SBG, y que ahora es la ruta más barata que queda en la lista; se verifica la meta y se decide que es la solución.

Figura 29. Un problema de determinación de ruta.



Mediante la búsqueda por costo uniforme se puede encontrar la solución más barata, siempre y cuando se satisfaga un requisito muy sencillo: el costo de la ruta nunca debe ir disminuyendo conforme avanzamos por la ruta. En otras palabras, es importante que:

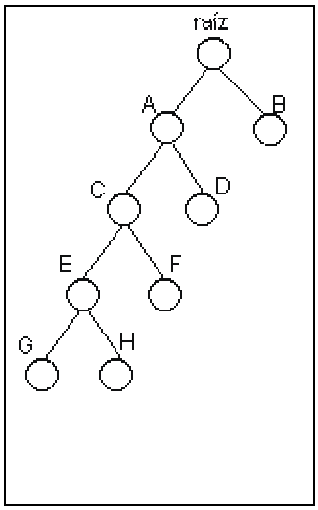
$g(SUCESOR(n)) \geq g(n)$
en todos los nodos n.

La restricción de que el costo de la ruta no vaya disminuyendo tiene sentido si se considera como costo de la ruta de un nodo a la suma de los costos de los operadores que configuran la ruta. Si el costo de todos los operadores no es negativo, el costo de una ruta nunca irá disminuyendo conforme se avanza, y mediante la búsqueda de costo uniforme será posible encontrar la ruta más barata sin tener que explorar el árbol de búsqueda en su totalidad. Pero si el costo de uno de los operadores es negativo, para hallar la solución óptima será necesario efectuar una búsqueda exhaustiva por todos los nodos, puesto que nunca será posible saber en qué momento una ruta está a punto de estar en un paso con un elevado costo negativo t, en consecuencia, convertirse en la mejor de todas las rutas.

3.4.3. Búsqueda preferente por profundidad

En la búsqueda preferente por profundidad siempre se expande uno de los nodos que se encuentre en lo más profundo del árbol. Sólo si la búsqueda conduce a un callejón sin salida, se revierte la búsqueda y se expanden los nodos de niveles menos profundos. La implantación de la anterior estrategia se realiza mediante BÚSQUEDA-GENERAL, con una función de lista de espera que va poniendo los estados recién generados a la cabeza de tal lista. Puesto que el nodo expandido fue el más profundo, los respectivos sucesores estarán a profundidades cada vez mayores. El desarrollo de la búsqueda se ilustra en la figura 30.

Figura 30. Árbol de búsqueda preferente por profundidad de un árbol de búsqueda binario.



En la búsqueda preferente por profundidad el volumen de memoria necesario es bastante modesto. Como se muestra en la figura 30, sólo es necesario guardar la ruta que vaya del nodo raíz al nodo hoja, junto con los nodos restantes no expandidos, por cada nodo de la ruta. Si un espacio de estados tiene un factor de ramificación *b* y profundidad máxima *m*, la cantidad de memoria necesaria en una búsqueda preferente por profundidad será sólo de *bm* nodos, en contraste con la cantidad b^d necesaria en una búsqueda preferente por amplitud, si la meta más inmediata se encuentra a profundidad *d*. Si se considera lo supuesto en la figura 28, para una búsqueda preferente por profundidad se necesitan 12 *kilobytes* en vez de los 111 *terabytes* para una profundidad 12, un factor de 10,000 millones de veces de menor espacio.

La complejidad temporal en una búsqueda preferente por profundidad es de $O(b^m)$. Si la cantidad de soluciones de un problema es excesiva, la búsqueda preferente por profundidad será mucho más rápida que la búsqueda preferente por amplitud, puesto que tiene más probabilidades de poder encontrar una solución luego de explorar tan sólo una pequeña porción de todo el espacio. Con la búsqueda preferente por amplitud todavía será necesario examinar todas las rutas de longitud *d*-1 antes de proceder a revisar las de longitud *d*. La búsqueda preferente por profundidad, aún en le peor de los casos, se reduce a $O(b^m)$.

La desventaja de la búsqueda preferente por profundidad es la posibilidad de que se quede estancada al avanzar por una ruta equivocada. En muchos problemas, los árboles de búsqueda son muy profundos, o hasta infinitos, por lo que en una búsqueda preferente por profundidad nunca será posible recuperarse de alguna desafortunada opción en uno de los nodos que están cerca de la parte superior del árbol. La búsqueda proseguirá por siempre en sentido descendente, sin hacia atrás, aún en el caso de que exista una solución próxima. Por lo tanto, en estos problemas, la búsqueda preferente por profundidad o se queda atorada en un bucle infinito y nunca es posible regresar al encuentro de una solución, o a la larga encontrará una ruta de solución más larga que la solución óptima. *Cuando hay árboles de búsqueda con prolongadas o infinitas profundidades máximas hay que evitar el empleo de la búsqueda preferente por profundidad*.

Es muy fácil implantar la búsqueda preferente por profundidad mediante BÚSQUEDA-GENERAL:

Function DEPTH-FIRST-SEARCH(*problem*) ***return*** *a solution, or failure* **Función** BÚSQUEDA-PREFERENTE-POR-PROFUNDIDAD(*problema*) **responde** con una solución o falla BÚSQUEDA-GENERAL(*problema*) EN-LISTA DE ESPERA AL FRENTE

Es frecuente también implantar la búsqueda preferente por profundidad utilizando una función recursiva, la cual se solicita a sí misma por cada uno de sus hijos en turno. En este caso, la lista de espera se guarda automáticamente en el estado local cada vez que se solicite la pila de espera.

3.4.4. Búsqueda limitada por profundidad

Con la búsqueda limitada por profundidad se eliminan las dificultades que conllevan la búsqueda preferente por profundidad al imponer un límite a la profundidad máxima de una ruta. Para implantar tal límite se utiliza un algoritmo especial de búsqueda limitada por profundidad, o utilizando el algoritmo general de búsqueda con operadores que se informan constantemente de la profundidad. Por ejemplo, si en el mapa de Rumania hay 20 ciudades, sabemos que en el caso de que exista una solución esta deberá tener como máximo 19 ciudades. La implementación del límite de profundidad se lleva a cabo utilizando operadores del tipo: "si usted se encuentra en la ciudad A y hasta allí ha recorrido una ruta que incluya 19 pasos, proceda a generar un nuevo estado en la ciudad B con una longitud de ruta que sea una unidad mayor". Este nuevo conjunto de operadores garantiza que, de existir, se encontrará la solución; lo que no se garantiza es que la primera solución encontrada sea necesariamente la más breve: la búsqueda limitada por profundidad es completa. Requiere un tiempo de $O(b^l)$ y un espacio $O(bl)$, en donde l es el límite de la profundidad.

3.4.5. Búsqueda por profundización iterativa

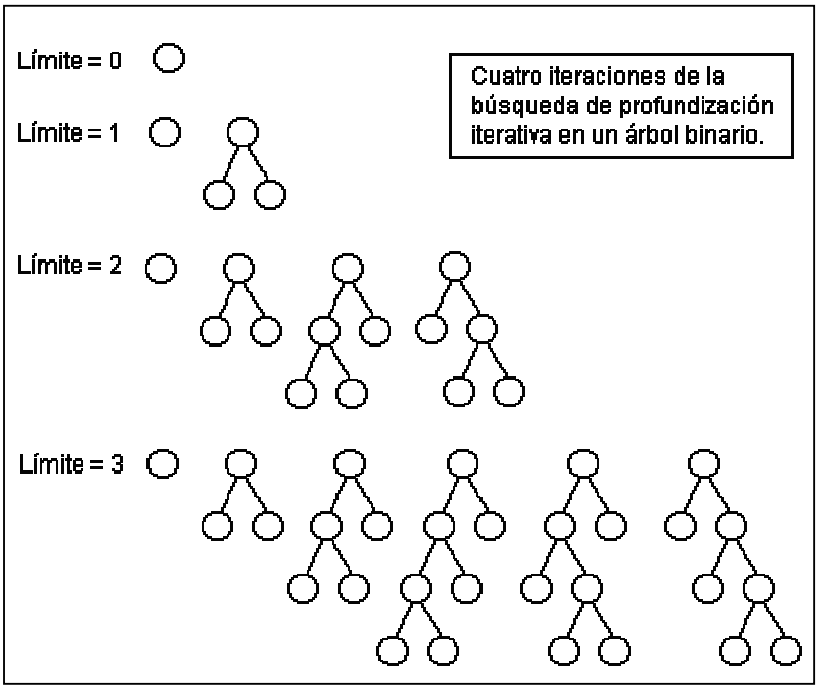
Lo difícil en una búsqueda limitada por profundidad es la elección del límite adecuado. En el caso del problema de Rumania el límite de 19 se escogió porque se trataba de un límite "obvio", aunque en realidad, de haber observado atentamente el mapa, se habrá dado cuenta de que es posible llegar a cualquier ciudad a partir de cualquiera otra en un máximo de 9 pasos. Este número, conocido como el **diámetro** del espacio de estados, constituye el mejor límite de profundidad, lo que, a su vez, favorece una búsqueda limitada por profundidad más eficiente. Sin embargo, en la mayoría de los problemas, no es posible determinar un adecuado límite de profundidad, hasta no haber resuelto el problema.

La búsqueda por profundización iterativa es una estrategia que esquiva el tema de la elección del mejor límite de la profundidad probando todos los límites de profundidad posibles: la primera profundidad 0, luego la profundidad 1, luego la profundidad 2, etc. El algoritmo respectivo se muestra en la figura 31. En efecto, en la profundización iterativa se combinan las ventajas de las búsquedas preferente por profundidad y preferente por amplitud. Es óptima y completa, como la búsqueda preferente por amplitud, al tiempo que la memoria que necesita es sólo la de la búsqueda preferente por profundidad. El orden de la expansión de los estados es semejante a la preferente por amplitud, excepto que algunos de los estados se expanden varias veces. En la figura 32 se muestra la primera de cuatro iteraciones de BÚSQUEDA-POR PROFUNDIZACIÓN-ITERATIVA en un árbol de búsqueda binario. Cuando mayor sea el factor de ramificación menor será el exceso de repetición de estados expandidos. Es decir que la complejidad temporal de la profundización iterativa sigue siendo $O(b^d)$ y la complejidad espacial es *O(bd). Por lo general, la profundización iterativo es el método idóneo para aquellos casos donde el espacio de búsqueda es grande y se ignora la profundidad de la solución*.

Figura 31. Algoritmo de búsqueda por profundización iterativa.

Function ITERATIVE-DEEPENDING-SEARCH(*problem*) ***return*** *a solution sequence*
Función BÚSQUEDA-POR-PROFUNDIZACIÓN-ITERATIVA(*problema*) **responde** con una secuencia de solución
Entradas: *problema*, un problema
Para *profundidad* ← 0 a infinito **hacer**
  Si BÚSQUEDA-LIMITADA-POR-PROFUNDIDAD(*problema*,*profundidad*) tiene éxito, **entregue** el resultado obtenido
Fin
Responda falla

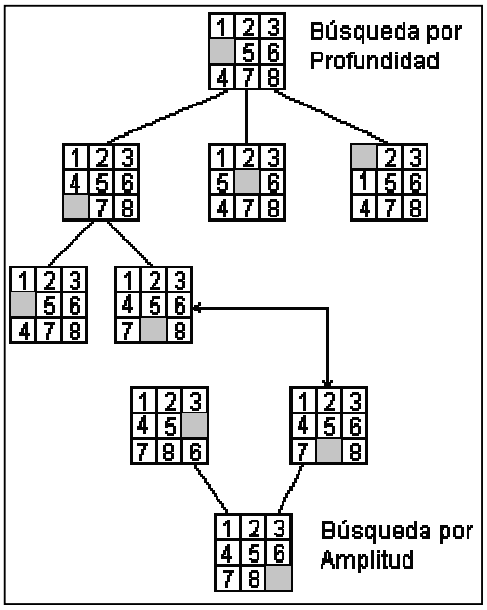
Figura 32. Búsqueda de profundización iterativa en un árbol binario.



3.4.6. Búsqueda bidireccional

La búsqueda bidireccional es, básicamente una búsqueda simultánea que avanza a partir del estado inicial y que retrocede a partir de la meta y que se detiene cuando ambas búsquedas se encuentran en algún punto intermedio figura 33. En el caso de los problemas cuyo factor de ramificación es *b* en ambas direcciones, la búsqueda bidireccional puede ser muy útil. Si, como de costumbre, se supone que existe una solución cuya profundidad es *d*, entonces la solución será $O(2b^{d/2})=O(b^{d/2})$ pasos, puesto que en las búsquedas hacia delante y hacia atrás solo se recorre la mitad del trayecto. Un ejemplo concreto: si b=10 y d=6, una búsqueda preferente por amplitud produce 1,111,111 nodos, en tanto que una búsqueda bidireccional tiene éxito cuando cada una de la direcciones de búsqueda están a profundidad 3, en cuyo caso se generan 2,222 nodos.

Figura 33. Búsqueda bidireccional.



En teoría esto es fabuloso, pero antes de poder implantar el algoritmo hay que resolver varias cuestiones.

- Lo más importante es: ¿qué significa buscar hacia atrás a partir de la meta?. Se definen los **predecesores** de un nodo *n* como todos aquellos nodos cuyo sucesor en *n*. La búsqueda hacia atrás implica la sucesiva generación de predecesores a partir del nodo meta.

- Si todos los operadores son reversibles, los conjuntos de predecesor y sucesor son idénticos; en algunos problemas, sin embargo, el calculo de los predecesores puede resultar muy difícil.

- ¿Qué se hace cuando son varios los posibles estados meta? En caso de contar con una lista *explicita* de los estados metas, como los estado meta de la figura 22, podemos aplicar una función de predecesor al conjunto de estado exactamente como se aplicó la función del sucesor en una búsqueda de estado múltiple. Si sólo contamos con una *descripción* del conjunto, aunque es posible darse una idea de los "conjuntos de estados que podría generar el conjunto meta", resulta un procedimiento engañoso. Por ejemplo, ¿cuáles son los estados que se consideran predecesores a la meta jaque mate en un juego de ajedrez?

- Es necesario contar con una manera eficiente de verificar cada uno de los nodos nuevos para ver si ya están en el árbol de búsqueda de la otra mitad de la búsqueda.

- Hay que definir qué tipo de búsqueda se realizará en cada una de las mitades.

En la cifra de complejidad $O(b^{d/2})$ se supone que el procedimiento para probar la intersección de las dos fronteras se puede efectuar en tiempo constante (es decir, es independiente de la cantidad de estados). Para ello se emplea una embrollada tabla. Con el fin de que ambas búsquedas lleguen a encontrarse en algún momento, los nodos de por lo menos uno de ellos deberá quedar retenido en la memoria. Es decir, la complejidad espacial de una búsqueda bidireccional que no cuenta con información es $O(b^{d/2})$.

3.5. Resumen

En éste capítulo se han estudiado estrategias de búsqueda que se pueden utilizar para resolver problemas. Se ha considerado que una buena estrategia debe cumplir con cuatro requerimientos:

- **Completez:** ¿La estrategia garantiza encontrar una solución, si es que ésta existe?
- **Complejidad temporal:** ¿Cuánto tiempo se necesitará para encontrar una solución?
- **Complejidad espacial:** ¿Cuánta memoria se necesita para efectuar la búsqueda?
- **Optimidad:** ¿Con esta estrategia se encontrará una solución de la más alta calidad. En caso de que existan varias soluciones?

Se presento seis estrategias de búsqueda agrupadas genéricamente bajo el nombre de **búsqueda sin contar con información.**

- Búsqueda preferente por amplitud
- Búsqueda preferente por lo mejor
- Búsqueda de costo uniforme
- Búsqueda preferente por profundidad
- Búsqueda limitada por profundidad
- Búsqueda por profundización iterativa
- Búsqueda bidireccional

4. MODELO VIRTUAL DE SAMY-BOT

4.1. Introducción

Antes de comenzar a ensamblar y probar los componentes del robot real, se va a desarrollar una aplicación que mostrará en un ambiente virtual como deberá interactuar *Samy-Bot* en los ambientes reales de los laberintos, para esto se va a utilizar los conceptos de Agente Inteligente, Ambientes y Estrategias de búsquedas tratados en el capítulo 3 de este trabajo de investigación.

Realizar pruebas en ambientes virtuales tiene la ventaja de que se deja por un lado la arquitectura y componentes que tendrá el robot y permite concentrarse exclusivamente en el sistema de control, es decir en el comportamiento o personalidad que tendrá el robot real.

El tipo de Agente que se utilizará es el Agente basado en metas, es así debido a que al robot le interesa encontrar la meta no importa la distancia que deba recorrer, una vez que ha localizado la meta entonces su objetivo se habrá cumplido. Al Agente se le agregará capacidad de memoria, es decir, aparte de poder encontrar la meta también podrá recordar la ruta correcta que lo lleva directo de la línea de inicio hacia la línea de meta.

Para programar el comportamiento o personalidad del Agente, se utilizará la estrategia de búsqueda preferente por profundidad, éste método permite encontrar la única solución que existe en el laberinto, y lo hace de forma que garantiza los cuatro requerimientos que debe cumplir una estrategia de búsqueda. Otra de las razones por las cuales se seleccionó esta estrategia de búsqueda es que el agente no tiene información previa al entrar al laberinto por lo tanto debe hacer uso de una estrategia de búsqueda sin información o búsqueda ciega.

4.2. Descripción de la aplicación *Samy-Bot* virtual

Se desea crear un ambiente virtual similar al que tendrá que enfrentar *Samy-Bot*, para ello se diseñará y desarrollará un programa de computadora que permita generar laberintos, la configuración de los laberintos debe ser modificable, es decir, el programa debe permitir que el usuario cambie la cantidad de filas y columnas que va a tener el laberinto a generar, estos laberintos serán completamente aleatorios, es decir el estado meta y el estado inicial van a ser seleccionados al azar así como las rutas dentro del laberinto. El programa debe tener un módulo o rutina que se encargue de resolver el laberinto, este módulo actuará como el agente inteligente que resolverá los laberintos, dicho módulo no podrá conocer en ningún momento la solución del laberinto únicamente se le proporcionará el estado inicial y el estado meta. Si el agente es capaz de resolver los laberintos entonces se tendrán pruebas suficientes para implementarlo en los circuitos del robot físico.

La secuencia básica de cada prueba será como sigue:

- Generar el laberinto.
- Pedir que el programa resuelva el laberinto.
- El programa muestra la solución encontrada.

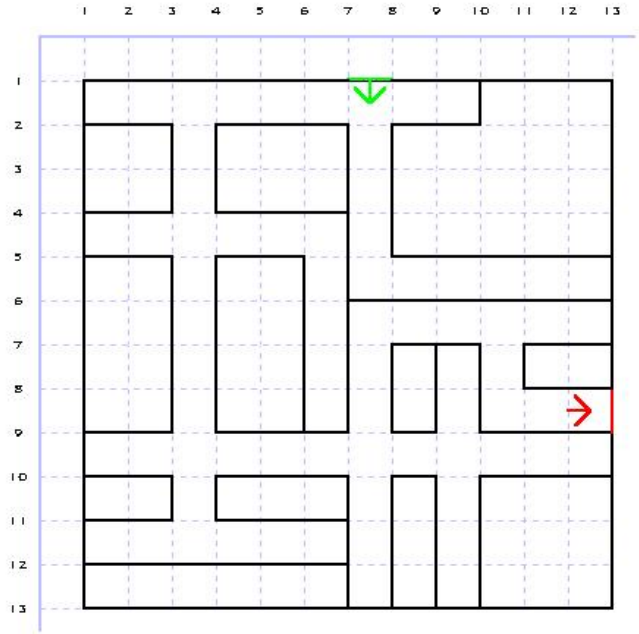
4.3. Análisis y diseño

El análisis y diseño expuesto aquí se tomará de referencia para el programa que será instalado en los circuitos del robot real.

4.3.1. Estructuras de datos

Para poder representar el ambiente en el cual operará *Samy-Bot* se va a utilizar como base una cuadricula generada por dos ejes perpendiculares como la que se utiliza en el plano cartesiano, cada cruce de caminos en el laberinto va a ser identificado en una posición en el cuadrante tal como se muestra en la figura 34, de esta forma se sabrá fácilmente la longitud de los caminos, el diseño que se va a hacer corresponde a laberintos que no tienen ciclos en sus caminos. Para representar este ambiente en la computadora se hará uso de una estructura híbrida compuesta por una matriz ortogonal y un árbol, la matriz ortogonal se utilizará para representar los cruces de caminos en el laberinto, es decir cada nodo de la matriz va a ser un cruce de caminos en el laberinto, así mismo los nodos de la matriz dispersa se utilizarán para formar un árbol que indique todas las rutas por las que el robot puede transitar para conseguir resolver los laberintos.

Figura 34. Laberinto de ejemplo.



4.3.1.1. Descripción del nodo híbrido

Estos registros se utilizarán para almacenar información acerca de los cruces de caminos y a la vez formarán parte del árbol de rutas que existen dentro de los laberintos.

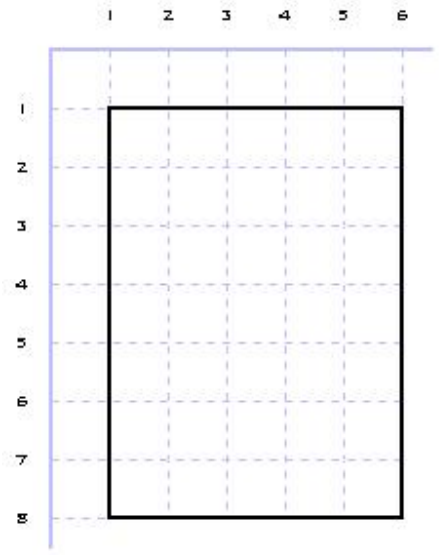
Registro NodoHibrido
  PosicionX numero
  PosicionY numero
  ContRamas numero
  EsMeta numero
  Ramas vector de apuntadores a NodoHibrido
  Dirección vector de apuntadores a NodoHibrido
  Padre apuntador a NodoHibrido
Fin de registro

- Los atributos o campos PosicionX y PosicionY se utilizarán para almacenar la posición del cruce de camino.
- El atributo ContRamas se utilizará para almacenar información acerca de la cantidad de ramas del árbol que salen del nodo.
- El atributo EsMeta se utilizará para almacenar información acerca de si el nodo es o no la meta.
- El atributo Ramas es un vector que se utilizará para almacenar información acerca de los punteros hacia las ramas que salen del nodo en cuestión y que forman parte del árbol.
- El atributo Dirección es un vector que se utilizará para almacenar información acerca de los punteros hacia sus nodos vecinos en la matriz ortogonal.
- El atributo Padre se utilizará para almacenar información acerca del nodo padre.

4.3.2. Módulo generador de laberintos

Para poder generar los laberintos se utilizará la estructura de los nodos híbridos descrita anteriormente. El primer paso es tomar los parámetros bajo los cuales se generará el laberinto estos parámetros son cantidad de filas y cantidad de columnas que podrá tener el laberinto, tomando la idea anterior del cuadrante formado por dos ejes perpendiculares, se colocaran las columnas sobre el eje horizontal y las filas sobre el eje vertical. La cantidad de columnas definirá en cuantos segmentos se debe dividir el eje horizontal y la cantidad de filas definirá la cantidad de segmentos en la cual se dividirá el eje vertical, por ejemplo si se desea tener 7 filas y 5 columnas la cuadricula quedará como se muestra en la figura 35.

Figura 35. Ejemplo de cuadrícula de los laberintos.



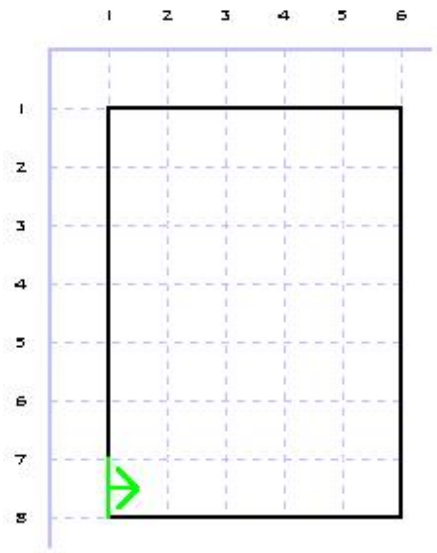
Cada segmento define el ancho de las rutas, de tal forma que si se hicieran únicamente caminos horizontales entonces la cantidad máxima de rutas sería igual a la cantidad de filas, y por el contrario si se hicieran únicamente caminos verticales entonces la cantidad máxima de rutas sería igual a la cantidad de columnas.

Este proceso básicamente se puede descomponer en tres procesos que se ejecutaran de forma secuencial; El proceso de identificar el estado inicial, el proceso de generación de rutas y finalmente el proceso de colocación de metas, cada uno de los procesos se explica detalladamente en los párrafos siguientes.

La cuadrícula resultante de las filas y columnas servirá de base para comenzar a generar los laberintos, como todo debe ser aleatorio primero se selecciona aleatoriamente el lado por el cual se iniciará a generar el laberinto para este efecto se nombrará al lado de arriba, Norte, al lado de abajo, Sur, al lado de la derecha desde el punto de vista del lector, Este, al lado izquierdo desde el punto de vista del lector, Oeste.

Luego de tener identificado el lado por el cual se iniciará el laberinto, entonces se selecciona aleatoriamente el segmento dentro del lado en el cual se va a colocar el estado inicial, por ejemplo si aleatoriamente se ha seleccionado el lado Oeste y luego se selecciona el segmento siete el nodo inicial quedará ubicado como se muestra en la figura 36.

Figura 36. Ejemplo de línea de inicio de un laberinto.



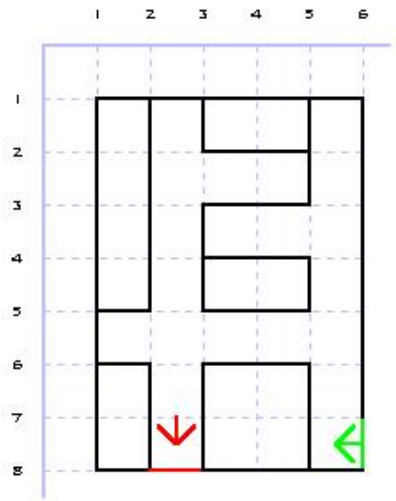
Este ha sido el primer paso en proceso de generación del laberinto, ahora es necesario enfrentar el paso de comenzar a generar rutas, antes de pasar al siguiente paso, el nodo inicial que se ha creado es ingresado a una cola, esta cola almacenará los nodos a ser expandidos.

Para generar las rutas dentro del laberinto se tomará como base la cola de nodos a ser expandidos, cuando esta cola se encuentre sin ningún elemento entonces el trabajo de generar rutas habrá terminado. El proceso es como sigue:

```
Mientas la cola de nodos a ser expandidos no este vacía
   Se saca un nodo de la cola de nodos a ser expandidos
   Se genera aleatoriamente el numero de rutas que parten de ese nodo
   Ciclo para de 1 a cantidad de rutas
      Se selecciona la Dirección de la ruta (esta puede ser norte, sur, este u oeste)
      Se crea el nuevo nodo
      Se ingresa el nodo a la matriz ortogonal
      Se ingresa el nodo al  árbol
      Si el nuevo nodo no esta en los limites del cuadrante del laberinto entonces
         Ingresar el nodo a la cola de nodos a ser expandidos
      Fin de si
    Fin de ciclo para
Fin de mientras
```

Luego viene el proceso de colocación de meta, para ello se examinan todos los nodos que están en los límites de cuadrante, se hace un conteo de ellos y luego se selecciona un número aleatorio para identificar el nodo que será es estado meta. Luego de Correr estos tres procesos el laberinto quedará completado como se muestra en la figura 37. Con esto se ha satisfecho el requisito de la generación de laberintos de forma totalmente aleatoria.

Figura 37. Ejemplo de un laberinto completo generado por software.



4.3.3. Módulo generador de soluciones

Para poder llevar a cabo el proceso de generación de laberintos, se hará uso de los conceptos de Agente Inteligente, Ambientes, Solución de Problemas y Algoritmos de Búsqueda expuestos en el capítulo 3.

4.3.3.1. Tipo de Agente y Ambiente

Para el caso del laberinto se utilizará el concepto de Agente basado en Metas, ya que al final lo que se persigue es alcanzar el estado meta del laberinto. El ambiente por sus características se considera que es Accesible, Determinista, Estático y Discreto; estas características del ambiente de un laberinto son favorables para que el Agente pueda interactuar en él y aprender en la medida que explora las diferentes rutas que el laberinto le ofrece.

4.3.3.2. Planteamiento del problema

- **Estado Inicial:**
  El estado inicial es la entrada del laberinto, ese será el primer parámetro que recibirá el agente para comenzar su tarea. Prácticamente se podrá suponer que el agente es abandonado o liberado en ese punto del laberinto, luego será el agente el que decida que hacer utilizando los operadores de los que dispone.

- **Conjunto de Operadores**

  Las funciones que el Agente podrá utilizar son:

  - Avanzar por una ruta.
  - Analizar cruce de camino.
  - Tomar decisión cuando se encuentra en un cruce de camino.
  - Regresar al cruce anterior cuando se encuentre en un camino sin salida.

- **Conjunto de Estados**

  Los estados en los cuales el Agente podrá estar son:

  - Entrada del laberinto.
  - Cruce a ser explorado.
  - Camino sin salida.
  - Salida del laberinto.

- **Prueba de meta**

  Para el robot virtual la prueba de meta la hará comprobando el estado del nodo en el que se encuentra, si el nodo esta marcado como meta entonces la tarea habrá concluido. En el caso del robot real la prueba de meta la hará comprobando si ya no hay paredes que le impidan moverse en la dirección que desee.

- **Estrategia de búsqueda**

  En el caso del ambiente de un laberinto la mejor estrategia a emplear es la Búsqueda Preferente por Profundidad, se ha seleccionado esta estrategia debido a que el problema tiene solución y ésta es única. La estrategia Preferente por Profundidad garantiza que localizara la meta cumpliendo con los requerimientos de Completez, Complejidad Temporal, Complejidad Espacial y Optimidad.

Tal como se explico en el capítulo 3 la estrategia de Búsqueda preferente por profundidad toma una ruta y la sigue hasta que encuentre la meta o un camino sin salida, si fuera el último caso entonces regresa al cruce anterior y continúa la búsqueda y así sucesivamente hasta que localiza la meta. Esta estrategia es la que le dará personalidad al Agente basado en meta que se va a utilizar. El proceso que define esta búsqueda se muestra a continuación:

```
Ingresar el estado inicial a una pila
Mientras no exista una solución
  Sacar un nodo de la pila
  Si el nodo no es el nodo meta entonces
    Si tiene un camino hacia su derecha entonces
      Localizar el nodo de la derecha
      Ingresar el nodo a la pila
    Si tiene un camino hacia adelante entonces
      Localizar el nodo de adelante
      Ingresar el nodo a la pila
    Fin de si
    Si tiene un camino hacia su izquierda entonces
      Localizar el nodo a su izquierda
      Ingresar el nodo a la pila
    Fin de si
  Fin de si
Fin de mientras
```

4.3.4. Interfaz del programa

La interfaz del programa será una interfaz tipo Windows, es decir tendrá todas las propiedades básicas de una ventana en el sistema operativo Windows. Contará con una barra de título, una barra de menús y un espacio para mostrar mensajes, los laberintos y la solución de los laberintos.

Las operaciones que se pueden realizar en el programa están agrupadas en tres menús, se han agrupado dependiendo del tipo de operación para que resulte amigable e intuitivo para el usuario. La distribución de las operaciones y los menús es como sigue:

4.3.4.1. Opciones del Menú Inicio

- Configuración del laberinto
- Imprimir el laberinto
- Salir del programa

4.3.4.2. Opciones del Menú Laberinto

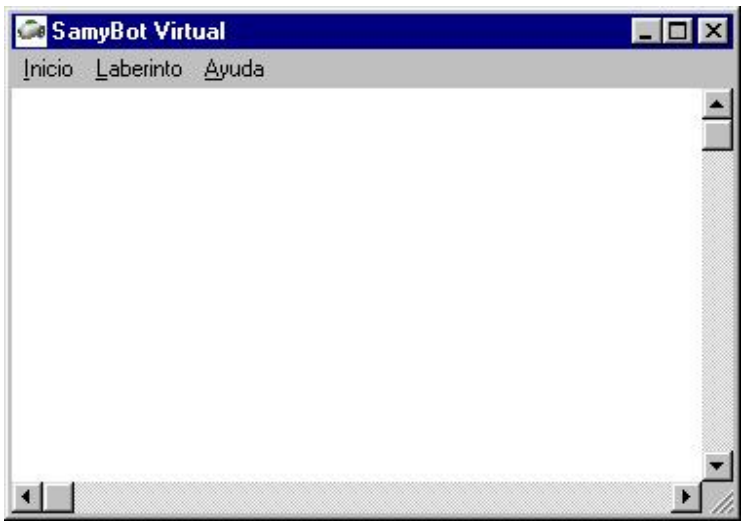
- Generar laberinto
- Resolver laberinto

4.3.4.3. Ayuda

- Acerca de

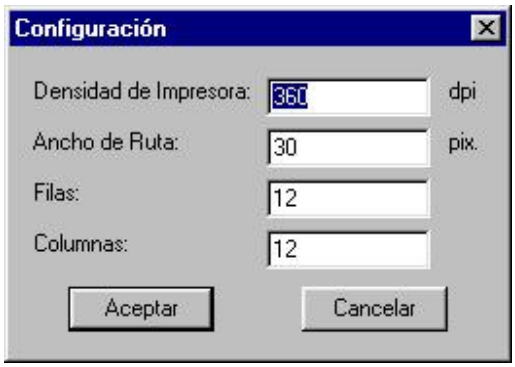
En la figura 38 se muestra como quedará distribuida la interfaz.

Figura 38. Diseño del entorno del *Samy-Bot* virtual.



Se utilizará un cuadro de dialogo para que el usuario ingrese las configuraciones de los laberintos, la distribución de este cuadro de dialogo se muestra en la figura 39.

Figura 39. Diseño del cuadro de dialogo de configuración.



4.4. Implementación

En esta sección se explicarán las clases que se utilizaron tomando como referencia el análisis y diseño realizado previamente. Antes de comenzar a explicar las clases es bien importante aclarar que el programa esta hecho en el lenguaje C++.

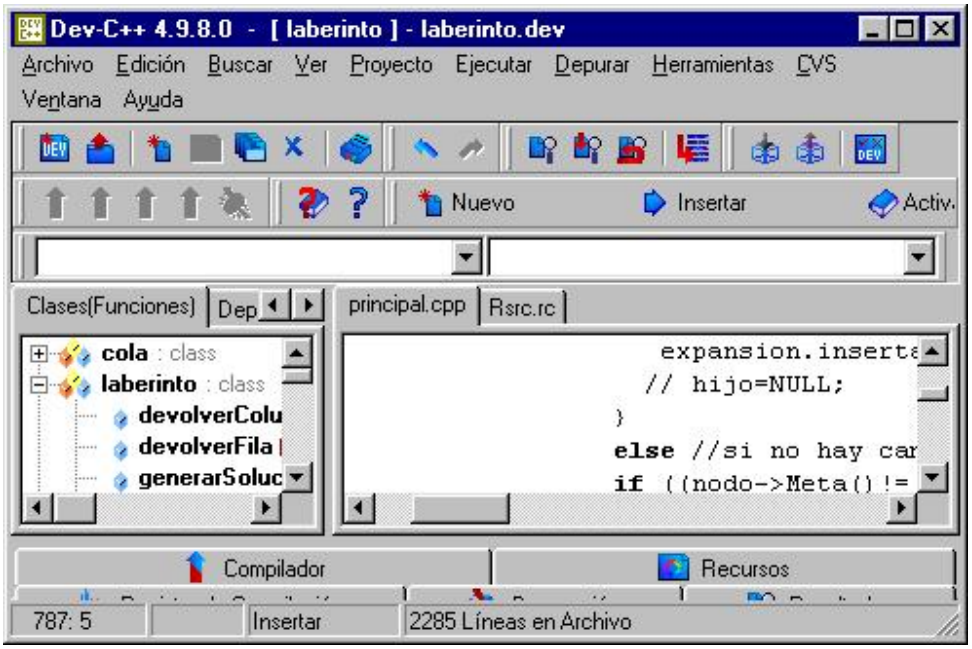
4.4.1. Herramienta de software

Para desarrollar este programa se va a utilizar un compilador de C llamado Dev-C++ en su versión beta 4.9.8.0 este software es distribuido por la empresa *Bloodshed* bajo la licencia GPL de GNU, las razones por las cuales se va a utilizar son las siguientes:

- Es absolutamente compatible con el estándar del lenguaje C
- Produce aplicaciones para Windows/Linux
- Es gratis

En la figura 40 se muestra como es la interfaz de Dev-C++.

Figura 40. Entorno de Dev-C++.



4.4.2. Clase Cola

Esta clase se utilizo para crear colas que auxilian a los procesos de generación de laberinto y solución del laberinto. Su diseño se muestra en la figura 41.

Figura 41. Código fuente de la clase cola.

```
class cola
{ private:
nodoC inicio;
nodoC fin;
int   cantidad; //cuenta el numero de nodos que componen la cola
public:
cola();
~cola();
void truncar();
void insertar(ApNodoH nodo);
ApNodoH sacar();
int contador();};
```

4.4.2.1. Atributos

- Inicio: este atributo indica el nodo inicial de la cola.
- Fin: este atributo indica el nodo final de la cola.
- Cantidad: este atributo indica la cantidad de nodos que hay en la cola.

4.4.2.2. Métodos

- void truncar(): este método se utiliza para vaciar completamente la cola.
- void insertar(ApNodoH nodo): este método ingresa un nodo a la cola, el cual recibe a través del parámetro nodo.
- ApNodoH sacar(): este método se utiliza para sacar un nodo de la cola.
- int contador(): este método devuelve la cantidad de nodos que hay en la cola.

4.4.3. Clase nodoHibrido

Esta clase creará las instancias que serán nodos dentro de la matriz ortogonal y el árbol de rutas. En la figura 42 se muestra su diseño.

Figura 42. Código fuente de la clase nodoHibrido.

```
class nodoHibrido/*esta clase se utiliza para construir el árbol de búsqueda
 también se utiliza en los nodos de la matiz esparcida
 dicha matriz se utiliza para representar el laberinto de forma que
 se puedan hacer comparaciones para que las diferentes rutas no se
 crucen*/
{  private:
  int          cont;      //numero de ramas que ha recibido el nodo
  ApNodoH   *    ramas;     //ramas del nodo
  ApNodoH   *    direccion;  //punteros a los otros nodos de la matriz que están en  sus cuatro
direcciones
             /*los punteros están codificados así:
              0: sur
              1: norte
              2: oeste
              3: este
              DESDE EL PUNTO DE VISTA DEL OBSERVADOR        */
 int           segx,segy;  //representa la posición en X y Y, esta posición se mide en segmentos
 int           distancia;  //distancia del entre el nodo hijo y el nodo padre
 ApNodoH       padre;     //padre del nodo
 char          meta;      //indica si es el nodo meta, en caso de serlo almacena el valor "v" de lo contrario
 int           sentido;    //indica el sentido de la ruta que describe el nodo
 public:
 nodoHibrido (int orden,int dist,char esmeta,ApNodoH nodoPadre, int x, int y,int sentid);
 ApNodoH    &rama(int num);
 ApNodoH    &direcciones(int num);
 ApNodoH    _padre();
 int       costo();
 int       &conteo();
 void       desencadenar(int orden);
 int       _sentido();
 int       &x();
 int       &y();
 void       Meta(char esmeta);
 char       Meta();
 int       &_distancia();
```

4.4.3.1. Atributos

- cont: representa el numero de ramas que salen del nodo
- ramas: este es un puntero que almacenará la dirección de un vector de punteros que llevan hacia las ramas que salen del nodo en cuestión.
- direccion: este es un puntero que almacenará la dirección de un vector de punteros que llevan hacia los nodos vecinos en la matriz.
- segy, segY: estos atributos almacenaran la posición del nodo dentro de la cuadrícula del laberinto.
- distancia: almacenará la distancia que hay entre el nodo padre y el nodo hijo.
- padre: este es un puntero que lleva al nodo padre del nodo en el que se encuentre.
- meta: indica si es un nodo meta, en caso de que lo sea almacenara la letra "v" si no lo es entonces almacenará la letra "f".
- sentido: indica en que sentido va la ruta que describe el nodo, es decir si va hacia el norte, sur, este u oeste.

4.4.3.2. Métodos

- ApNodoH &rama(int num): devuelve la rama en el nodo, que corresponde al numero que recibe en el parámetro num.
- ApNodoH &direcciones(int num): devuelve el vecino del nodo que corresponde a la dirección num. num puede representar norte, sur, este y oeste.
- ApNodoH _padre(): devuelve la dirección del padre del nodo.
- int costo(): devuelve el costo de llegar hasta el nodo partiendo desde el inicio del laberinto.
- int &conteo(): devuelve la cantidad de ramas que posee el nodo.
- void desencadenar(int orden): quita el puntero de la rama que corresponde al número orden.

- int _sentido(): devuelve el sentido de la ruta que describe el nodo.
- int &x(): devuelve la posición del nodo en el eje horizontal.
- int &y(): devuelve la posición del nodo en el eje vertical.
- void Meta(char esmeta): se utiliza para indicarle al nodo que él es la meta
- char Meta(): se utiliza para saber si el nodo es o no la meta.
- int &_distancia(): devuelve la distancia que hay entre el nodo padre y el nodo actual.

4.4.4. Clase Laberinto

Esta clase se utilizará para hacer instancias que representen el ambiente de los laberintos de prueba para el Agente, tanto las rutinas del simulador del ambiente como las rutinas del Agente Inteligente están encapsuladas en esta misma clase, se ha hecho de tal forma que el Agente no recibe información o pistas de por donde debe ir. En el cuadro 4-3 se muestra su diseño.

Figura 43. Código fuente de la clase laberinto.

```
class laberinto{  private:
  ApNodoH matriz;
  int    Xfinal,Yfinal;
  int    anchoSeg;        //ancho de las rutas.
  int    segY,segX;       //segmentos en X y segmentos en Y
  int    longmin,longmax;  //longitud mínima y máxima que puede tener un segmento
  int    orden;
  int    solucionado;   //variable que indica si un laberinto ya fue solucionado
  ApNodoH raiz;
 public:
  laberinto(int Xfin,int Yfin,int anchoSeg,int longmin,int longmax);
  int insertarMatriz(ApNodoH nodo);
  int generarLaberinto();
  void generarExpansion(); //saca los nodos de la cola expansión y genera las rutas
  int colocarMeta();
  ApNodoH devolverFila(int n);
  ApNodoH devolverColumna(int n);
  void solucion();
  char generarSolucion(ApNodoH nodo);
~laberinto();
  int existe(int x, int y);
```

4.4.4.1. Atributos

- matriz: es un puntero que indica donde comienza la matriz ortogonal.
- Xfinal, Yfinal: almacenarán el ancho y largo de los laberintos medido en píxeles.
- anchoSeg: almacenará el ancho de los caminos medido en píxeles, este atributo servirá cuando el laberinto se vaya a dibujar en la pantalla de la computadora o cuando se va a imprimir.
- segX, segY: estos atributos representan la cantidad de columnas y filas, respectivamente que puede tener el laberinto, estos son parámetros que el usuario debe ingresar. La multiplicación de segX por anchoSeg y de segY por anchoSeg dan como resultado los valores de los atributos Xfinal e Yfinal respectivamente.
- longmin, longmax: representan la longitud mínima y máxima que puede tener un segmento del camino. Se entiende por segmento aquel segmento de ruta que esta ubicado entre dos cruces de ruta.
- orden: este atributo representa el orden del árbol, para el caso de los laberintos el orden del árbol siempre será tres, debido a que como máximo cada nodo podrá tener tres ramas.
- solucionado: este atributo indica si el laberinto ya fue solucionado.
- raíz: este atributo es un puntero que indica el comienzo del árbol de rutas.

4.4.4.2. Métodos

- int insertarMatriz(ApNodoH nodo): Este método se utiliza para insertar nodos en la matriz ortogonal, el nodo a insertar se le pasa a través del parámetro nodo.
- int generarLaberinto(): Este método es el que inicia el proceso de generación del laberinto, construye el nodo inicial y lo inserta en la cola de nodos a ser expandidos, luego llama al método generarExpansion y finalmente coloca la meta.

- void generarExpansion(): Este método es el que genera el resto del laberinto, lo hace utilizando la cola de nodos a ser expandidos hasta que ya no queda ningún nodo para ser expandido.
- int colocarMeta(): Este es el método que selecciona el nodo que será la meta del laberinto.
- ApNodoH devolverFila(int n): Este método devuelve el encabezado de la fila que indica el parámetro n.
- ApNodoH devolverColumna(int n): Este método devuelve el encabezado de la columna que indica el parámetro n.
- void solucion(): Este método es el que inicia el proceso de solución del laberinto.
- char generarSolucion(ApNodoH nodo): Este es un método recursivo, es el método que representa el comportamiento del Agente Inteligente, se desarrollo basado en el método de Búsqueda Preferente por Profundidad.
- int existe(int x, int y): Este es un método que se utiliza para verificar si existe un nodo en la posición x e y de la matriz ortogonal.
- void mostrarLaberinto(HDC hDC,int inicioY,int inicioX,int printer): Este es el método que muestra en pantalla el diseño del laberinto.
- void mostrarSolucion(HDC hDC,int inicioY,int inicioX, int printer): Este es el método que muestra en pantalla la solución encontrada por el Agente Inteligente utilizando el método de Búsqueda Preferente por profundidad.

4.5. Manual de usuario

En esta sección se mostrará paso a paso como se pueden realizar operaciones en el programa. Antes de comenzar a describir en detalle cada operación y la forma de realizarla, se va a plantear un esquema general de cómo se deben coordinar las diferentes funciones que ofrece el programa *Samy-Bot* Virtual. La secuencia de las operaciones es como sigue:

- Definir los parámetros que se utilizaran en la generación de laberintos, este paso es opcional ya que el programa tiene valores predeterminados para los diferentes parámetros, los cuales se deberán cambiar en caso de que se desee generar laberintos con parámetros distintos.
- Generar el laberinto.
- Resolver el laberinto.
- Imprimirlo, si desea tener una copia impresa.

4.5.1. Cómo definir los parámetros para la generación de laberintos

Para que un laberinto pueda ser generado el programa toma en cuenta diferentes parámetros, los cuales se explican a continuación:

- Densidad de Impresora: éste parámetro le indica al programa la densidad con que imprime la impresora, medida en puntos por pulgada, el valor predeterminado es 360, éste parámetro se utiliza en el momento en el que se va a imprimir el laberinto.
- Ancho de ruta: éste parámetro le indicará al programa la cantidad de píxeles de ancho que deben ser los caminos dentro del laberinto, se utiliza cuando se va a dibujar el laberinto.

- Filas: éste parámetro le indica al programa la cantidad de filas que tendrá la cuadricula del laberinto.
- Columnas: éste parámetro le indica al programa la cantidad de columnas que tendrá la cuadricula del laberinto.

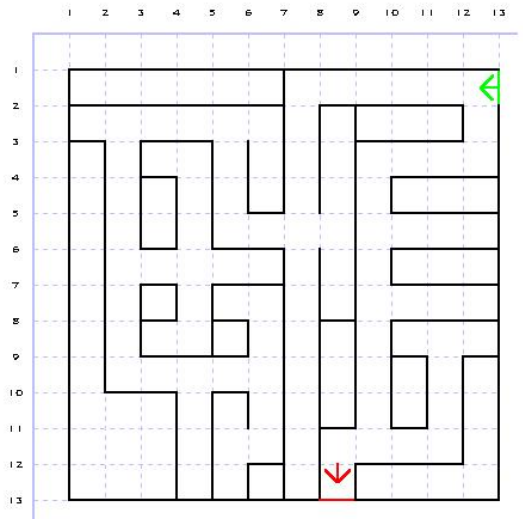
Los pasos de cómo definir los parámetros se muestran a continuación.

<table>
<tr>
<td>1) Ir al menú inicio, y luego seleccionar la opción Configurar, tal como se muestra en la figura 44.</td>
<td>Figura 44. Opción Configurar del menú Inicio.
 
</td>
</tr>
<tr>
<td>2) Ahora se debe ingresar valores a los parámetros de configuración, como se muestra en la figura 45.

3) Finalmente se presiona Aceptar para grabar los valores o Cancelar para ignorar los cambios.</td>
<td>Figura 45. Cuadro de diálogo de Configuración.
 
</td>
</tr>
</table>

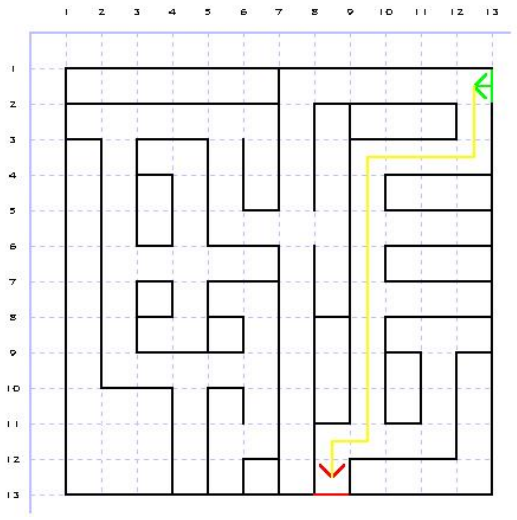
4.5.2. Cómo generar los laberintos

Una de las principales funciones del programa *Samy-Bot* Virtual es, generar laberintos totalmente aleatorios, estos laberintos toman como base los parámetros ingresados en el cuadro de dialogo de configuración de parámetros. Los pasos para obtener un nuevo laberinto se muestran a continuación:

1) Ir al menú Laberinto y seleccionar la opción Generar Laberinto, como se muestra en la figura 46.	**Figura 46.** Opción Generar Laberinto del menú Laberinto. 
2) El laberinto generado se muestra en la ventana del programa, el inicio lo indica la flecha hacia adentro del laberinto, la meta la indica la flecha que esta hacia fuera del laberinto, como se muestra en la figura 47.	**Figura 47.** Laberinto generado por Samy-Bot Virtual. 

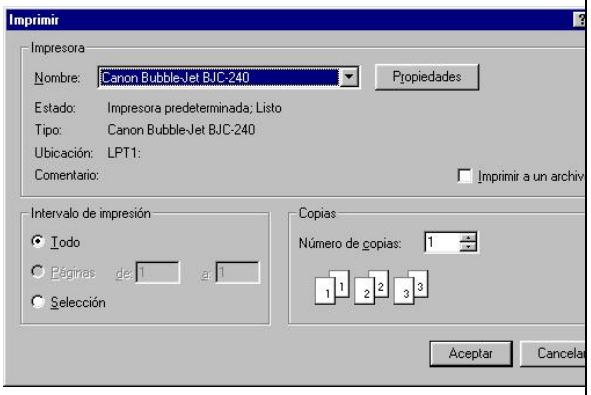
4.5.3. Cómo resolver los laberintos

La otra función principal del programa es la de resolver los laberintos, para resolverlos se ha utilizado el algoritmo de Búsqueda preferente por profundidad presentado en el capítulo 3. A continuación se muestra la secuencia de pasos a seguir para solicitarle al programa que resuelva el laberinto:

1) Ir al menú Laberinto y seleccionar la opción Resolver Laberinto, tal como se muestra en la figura 48.	**Figura 48.** Opción Resolver Laberinto del menú Laberinto. 
2) El programa muestra la solución trazando la ruta solución, tal como se muestra en la figura 49.	**Figura 49.** Laberinto resuelto por software. 

4.5.4. Cómo imprimir los laberintos

Cuando se desea tener una copia impresa del laberinto generado se utiliza la opción de imprimir laberinto, los pasos de esta operación se muestran a continuación.

1) Ir al menú Inicio y seleccionar la opción Imprimir Laberinto, tal como se muestra en la figura 50.	**Figura 50.** Opción Imprimir Laberinto del menú Inicio. 
2) Seleccionar la impresora y presionar el botón Aceptar para enviar la impresión o Cancelar para ignorara la impresión. La ventana de impresión se muestra en la figura 51.	**Figura 51.** Ventana de Impresión. 
3) Luego de que se ha enviado a imprimir, se muestra un mensaje que indica que el laberinto ha sido impreso, tal como se muestra en la figura 52, luego debe presionar el botón Aceptar.	**Figura 52.** Mensaje de status de impresión. 

4.6. Resumen

En este capítulo se ha hecho un recorrido completo a través del análisis, diseño e implementación de la aplicación *Samy-Bot* Virtual, el principal objetivo de ésta aplicación es probar la estrategia de búsqueda en un ambiente virtual, antes de pasar al desarrollo del robot real que resolverá laberintos. Para el análisis se ha hecho uso de los conceptos de Agente Inteligente, Ambientes y Algoritmos de búsqueda tratados en el capítulo 3.

En el caso del ambiente de un laberinto la mejor estrategia a emplear es la Búsqueda Preferente por Profundidad, se ha seleccionado esta estrategia debido a que el problema tiene solución y ésta es única. La estrategia Preferente por Profundidad garantiza que localizará la meta cumpliendo con los requerimientos de Completez, Complejidad Temporal, Complejidad Espacial y Optimidad.

Se utilizó el concepto de Agente basado en Metas, ya que al final lo que se persigue es alcanzar el estado meta del laberinto. El ambiente por sus características se considera que es Accesible, Determinista, Estático y Discreto; estas características del ambiente de un laberinto son favorables para que el Agente pueda interactuar en él y aprender en la medida que explora las diferentes rutas que el laberinto le ofrece.

La aplicación *Samy-Bot* Virtual tiene dos funciones esenciales, una de ellas es el generador de ambientes, el cual genera los laberintos virtuales totalmente aleatorios. Para almacenar la información de los laberintos utiliza una estructura de datos híbrida, compuesta por un árbol de búsqueda inmerso en una matriz ortogonal. La otra función esencial es el generador de soluciones, esta es la parte encargada de resolver los laberintos y mostrar la solución, para esta parte se utilizo la estrategia de búsqueda Preferente por Profundidad.

5. CONSTRUCCIÓN DE SAMY-BOT

5.1. Introducción

Los robots son usados en la industria automotriz, médica, plantas fabriles y por supuesto, en las películas de ciencia-ficción. Construir y programar un robot es una resolución combinada de problemas, de electrónica, mecánica y programación. Para el caso de esta investigación se construirá un robot autónomo que pueda maniobrar dentro del ambiente de los laberintos, de forma que pueda resolverlos y aprender la solución.

En éste capítulo se hablará acerca de cómo se construyo *Samy-Bot* y los componentes que se utilizaron. Se mostrará paso a paso como se fue ensamblando cada componente. Se introducirá el concepto de microcontrolador y servomotores. Para poder tener un orden durante el desarrollo de este capítulo se ha utilizado la metodología de sistemas para poder desglosar a *Samy-Bot* en sus sistemas principales los cuales son:

- Sistema de locomoción.
- Sistema de visión.
- Sistema de control.
- Sistema de plataforma mecánica
- Sistema de alimentación de energía

El sistema de locomoción estará constituido por dos servomotores truncados, el sistema de visión estará constituido por dos sensores infrarrojos uno de medición de distancia y un sensor detector de líneas negras. El sistema de control estará constituido por un microcontrolador *Basic Stamp* 2, y una Plaqueta de Educación para microcontroladores *Basic Stamp* 2.

El sistema de plataforma mecánica esta constituido por el chasis, tornillos y tuercas necesarios para sujetar todos los demás sistemas. El sistema de alimentación de energía está constituido por un porta pilas que va conectado a la Plaqueta de Educación.

Durante el desarrollo de cada sistema se irá explicando la teoría concerniente a los componentes principales de cada sistema.

El robot que se pretende construir tendrá Inteligencia Artificial con el propósito de que pueda maniobrar dentro de los laberintos de forma autónoma, de tal forma que el resultado de sus acciones dependa exclusivamente de las decisiones que el robot tome cuando este en la fase de resolución de un laberinto. Para que esto sea posible se tomará como base el concepto de Agente Inteligente específicamente Agentes basados en metas, también se utilizará el algoritmo de Búsqueda preferente por profundidad para programar el comportamiento del Agente.

5.2. Descripción de *Samy-Bot*

Se busca construir un robot autónomo que pueda maniobrar dentro del ambiente de los laberintos, de forma que pueda resolverlos y aprender la solución. En esta parte del ejercicio se va a hacer uso de los conocimientos adquiridos en los cursos relacionados con la electrónica, un poco de física y un poco de matemática.

El robot que se construirá será un robot tipo vehículo, tendrá una rueda atrás y dos ruedas adelante, se ha seleccionado esta arquitectura ya que presenta la ventaja de que tener ruedas soluciona de forma sencilla el requerimiento de locomoción. Para identificar las rutas en el laberinto se utilizará una línea negra que pasará por en medio de todas las rutas, parecido a las líneas blancas o amarillas que tienen las carreteras. Para delimitar los caminos se utilizaran paredes de color blanco.

Para el módulo de visión se utilizarán sensores infrarrojos colocados en el frente en forma de pequeños ojos, estos serán los encargados de identificar si hay o no paredes delante del robot. Para identificar las líneas en el suelo se utilizaran sensores infrarrojos colocados abajo del robot.

Para que el robot sea absolutamente autónomo, se utilizara por cerebro un microcontrolador *Basic Stamp* II de la casa *Parallax Inc*. Todo esto se explicará en detalle más adelante.

5.3. Construcción y prueba de *Samy-Bot*

Cuando se está construyendo un robot, es mejor imaginarlo como un conjunto de sistemas, subsistemas y elementos básicos. Un buen ejemplo de un sistema que puede ser descompuesto en subsistemas y elementos básicos podrían ser los servomotores de Samy-Bot. Como sistema, un par de servomotores modificados que funcionan como motores, trabajando juntos, hacen que el robot se desplace. Cada servomotor puede ser visto como un subsistema. Cada servomotor tiene una pequeña plaqueta de circuito impreso en su interior, con componentes electrónicos. Este es un ejemplo de un subsistema dentro de otro subsistema. Cada componente electrónico, si no puede seguir siendo descompuesto en componentes más pequeños, podría considerarse un elemento básico. Cada servomotor también tiene un subsistema de engranajes. Un engranaje en particular no puede ser separado en más partes, así que será un elemento básico. Cada servomotor recibe señales eléctricas, que le dicen lo que debe hacer, desde el *BASIC Stamp*, el cerebro del robot. El *Basic Stamp* es otro sistema.

Una de las actividades más importantes, cuando se hace un robot, es el desarrollo y prueba de cada subsistema individual, de un sistema dado. Luego, también deben realizarse pruebas a nivel del sistema, para asegurarse que todos los subsistemas trabajan conjuntamente en la forma que se esperaba.

Por último, pero no menos importante, se realiza la integración de sistemas, asegurándose que los mismos funcionen coordinadamente. La prueba y la solución de problemas en cada fase del desarrollo, en niveles de sistema y subsistema es, hasta cierto punto, una habilidad que se perfecciona con la práctica.

El desarrollo de robots es un proceso iterativo en varios sentidos. Desarrollo "iterativo" significa probar repetidamente y ajustar algo hasta que trabaje como se planeaba. La clave del desarrollo iterativo está en que los resultados de las pruebas son usados para realizar ajustes finos. Luego se realizan más pruebas y ajustes en la siguiente "iteración" de pruebas. En este capítulo, el proceso iterativo será desarrollado, probado, ajustado si es necesario, luego desarrollado un poco más, probado nuevamente, etc. El objetivo principal es armar el robot y hacerlo funcionar, sin tener que desarmarlo para realizar más pruebas y reparaciones.

5.3.1. Componentes de Hardware de *Samy-Bot*

Para todas las actividades que se llevarán a cabo en este capítulo se necesitará una computadora personal con sistema operativo Windows 95,98,2000 o NT.

Para construir el robot se utilizaron los siguientes componentes:

a) 1 Chasis
b) 1 Porta-Pilas
c) 2 Servomotores
d) 2 Ruedas plásticas
e) 1 Bolilla de polietileno
f) 2 Bujes de goma 9/32"
g) 1 Buje de goma 13/32"
h) 1 Plaqueta de educación para el microcontrolador *Basic Stamp* 2
i) 2 Cubiertas de goma para las ruedas

j) 1 Chaveta o 1 alambre de aluminio para sujetar la bolilla de polietileno
k) 10 Tuercas 4-40
l) 2 Tornillos de cabeza plana 4-40
m) 8 Tornillos 3/8" 4-40
n) 8 Tornillos 1/4" 4-40
o) 4 Separadores 1/2"
p) 1 Cable serial *Null Módem*
q) 4 Pilas AA
r) 1 Módulo de sensores medidores de distancia por infrarrojos de *Gazbot* GIDS-01
s) 1 Módulo de sensores infrarrojos seguidores de líneas de *Parallax* número de parte 29115.

5.3.2. Sistema de control

5.3.2.1. Componentes del sistema de control

En esta actividad se va a utilizar los siguientes componentes:

4 Tornillos 1/4" 4-40
4 Separadores
1 Módulo *Basic Stamp* 2
1 Plaqueta de educación para *Basic Stamp* 2
1 Porta-pilas
4 Pilas AA
1 Cable Serial

Estos componentes se muestran en la figura 53.

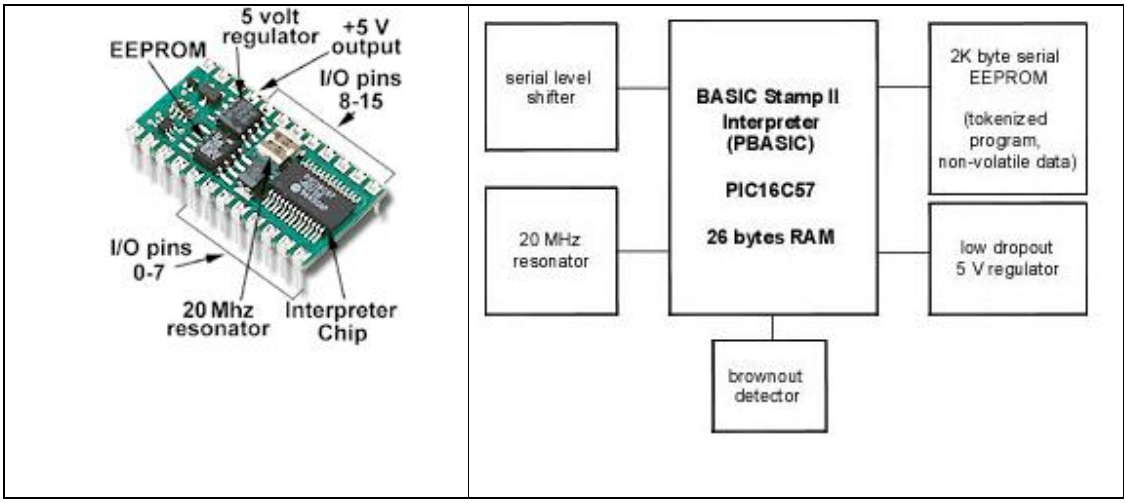
Figura 53. Componentes del sistema de control.

5.3.2.2. Basic Stamp 2

Es un pequeño microcontrolador del tamaño de una estampilla postal que ejecuta programas en lenguaje PBASIC. El *Basic Stamp* 2 tiene 16 pines I/O (entrada / salida) que pueden ser conectados directamente a dispositivos digitales o de niveles lógicos, tales como botones, LEDs, parlantes, potenciómetros, y registros de desplazamiento. Además, con unos pocos componentes extra, estos pines de I/O pueden ser conectados a dispositivos que no manejen niveles TTL, tales como solenoides, relés, redes RS-232, y otros dispositivos de alta corriente o tensión.

El diseño físico consiste de un regulador de 5 voltios, resonador, EEPROM serial, e intérprete PBASIC, en la figura 54 se muestra el diseño físico y el diagrama del *Basic Stamp* 2. Un programa simbolizado (o tokenizado) en PBASIC es almacenado en la EEPROM serial no volátil, desde donde el chip intérprete lee y escribe valores. Este chip intérprete ejecuta una instrucción por vez, realizando la operación apropiada en los pines de I/O o en la estructura interna del intérprete. Debido a que el programa PBASIC es almacenado en una EEPROM, puede ser reprogramado una cantidad casi infinita de veces, sin la necesidad de borrar primero la memoria.

Figura 54. Diseño físico y diagrama del Basic Stamp 2.



Para programar el *Basic Stamp* 2, simplemente se coloca en la Plaqueta de Educación, luego se conecta a una computadora, y se ejecuta el software editor para crear y descargar los programas, a través del cable serial. El *Basic Stamp* 2 almacena hasta 500 a 600 instrucciones y ejecuta un promedio de 4,000 instrucciones / seg.

5.3.2.3. Plaqueta de Educación

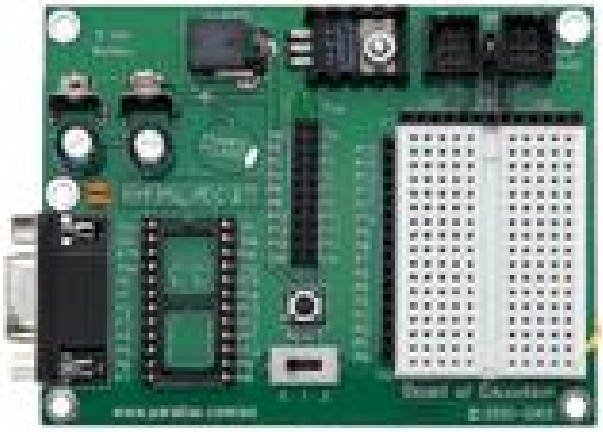
La Plaqueta de Educación es una herramienta para realizar proyectos con el microcontrolador *Basic Stamp* 2. La plaqueta fue diseñada de acuerdo a los requerimientos de facilidad en la conexión y programación de microcontroladores. Las características de la Plaqueta de Educación son las siguientes:

- Conectores de alimentación dispuestos para evitar la conexión simultánea de batería de 9 V y fuente de alimentación externa.
- Cuatro puertos para conectar servos para los proyectos de robótica.
- Zócalo para el *Basic Stamp* 2 que permite sacarlo con facilidad.

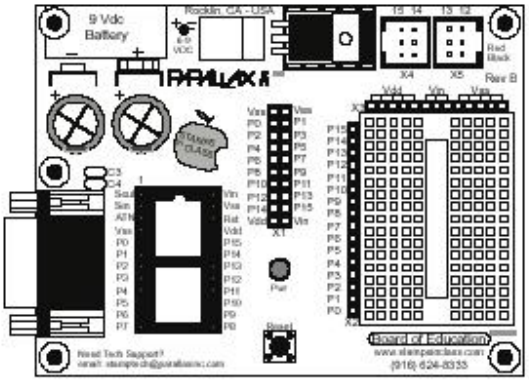
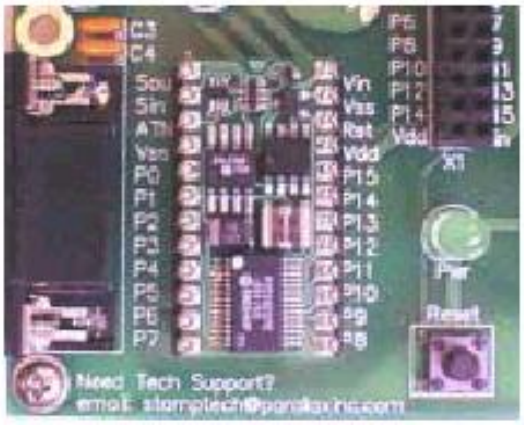
- Conector DB9 para programar el *Basic Stamp* 2 y mantener comunicación serial durante la ejecución del programa.
- Conexiones a los pines de I/O P0 - P15, Vdd y Vss, a los costados del área de prototipo de 2" x 1 3/8".
- Conector hembra de 10 pines para adicionar *AppModules* (más espacio de prototipo).
- Los trazos sobre la plaqueta muestran las conexiones entre el *Basic Stamp* 2 y los conectores laterales de la *protoboard*.

La plaqueta de educación se muestra en la figura 55.

Figura 55. Diseño físico de la Plaqueta de Educación.



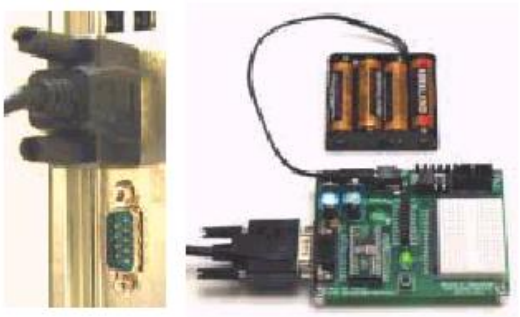
A continuación se muestran los pasos de cómo se ensambla el *Basic Stamp* a la Plaqueta de Educación.

1) La figura 56 muestra el porta-pilas antes y después de ponerle las 4 pilas AA, es importante colocar las pilas con la polaridad como lo indica el porta pilas.	**Figura 56.** Porta-Pilas de *Samy-Bot.* 
2) La figura 57 muestra el esquema de la plaqueta de educación que se utiliza con el microcontrolador *Basic Stamp* 2 de *Parallax*.	**Figura 57.** Esquema de la plaqueta de educación. 
3) El microcontrolador debe montarse en el zócalo de la plaqueta de educación, tal como se muestra en la figura 58. El *Basic Stamp* tiene medio círculo impreso en el centro de su extremo, el semicírculo debe estar cercano a los rótulos Sout y Vin de la placa de educación.	**Figura 58.** Basic Stamp 2 ensamblado en la plaqueta de educación. 

4) El cable serial debe conectarse a un puerto com de la computadora como se muestra en la figura 59(a).

5) El otro extremo del cable serial y el cable del porta-pilas se deben conectar a la plaqueta de educación, como se muestra en la figura 59(b).

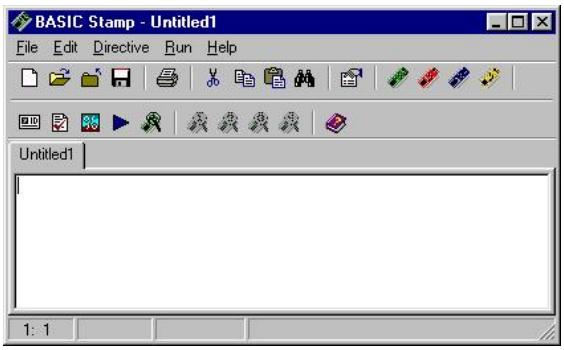
Figura 59. (a) Cable serial conectado a un puerto com de una computadora. (b) Cable serial y porta-pilas conectados a la plaqueta de educación.



a b

6) Para el siguiente paso se debe tener instalado el software *Basic Stamp Editor* de *Parallax*, este se utiliza para programar los microcontroladores *Basic Stamp* de *Parallax*. El editor se muestra en la figura 60.

Figura 60. Entorno de *Basic Stamp Editor*.

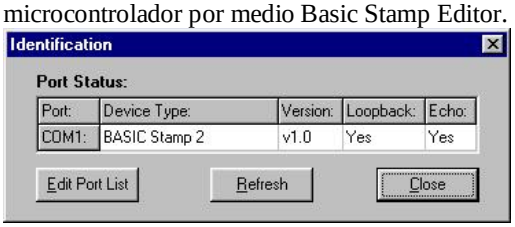


7) Ahora se procede a comprobar la comunicación del PC con el *Basic Stamp* 2, para ello se debe ir al menú *Run* del Basic *Stamp Editor* y seleccionar la opción *Identify*, como se muestra en la figura 61.

Figura 61. Detectando el microcontrolador que está conectado a la computadora.

8) Si la comunicación entre el PC y el *Basic Stamp* esta bien, entonces se muestra un mensaje como el de la figura 62. Ahora ya se puede comenzar a escribir programas.

Figura 62. Mensaje de identificación de microcontrolador por medio Basic Stamp Editor.



5.3.3. Sistema de locomoción

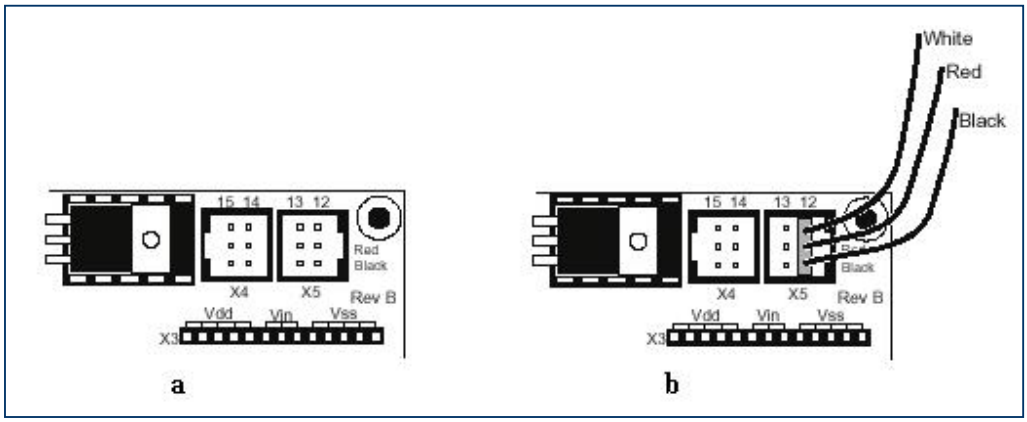
5.3.3.1. Servomotores

De ahora en adelante se llamará a los servomotores simplemente, servos. Los servos para hobby son motores especiales con realimentación de posición interna. Su rango de giro es típicamente de 90 ó 180 grados y son especiales para aplicaciones donde se requiera un movimiento de mucha fuerza con precisión y a bajo costo. Son muy populares en los sistemas de control de autos, botes y aviones radio controlados. Los servos están diseñados para controlar la posición de un alerón en un avión o el timón en un bote radio controlado.

5.3.3.2. Instalación de los servos

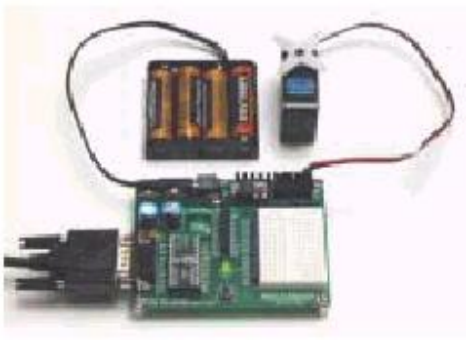
Los servos a utilizar son servos que vienen modificados y calibrados de fábrica, estos tienen un conector formado por tres cables de diferente color. La placa de educación para *Basic Stamp*, de ahora en adelante se le llamará PDE, tiene cuatro puertos para servos, la figura 63(a) muestra un acercamiento a los puertos de la PDE. Los números en la parte superior indican el número de puerto. Si se conecta un servo en el Puerto 12, significa que la línea de control del servo está conectada al pin de E/S P12. La línea que conecta a P12 es un trazo metálico en la PDE que une el pin superior del puerto del servo con el pin de entrada/salida P12 del *Basic Stamp*. Los rótulos del costado derecho de los puertos para servos están para asegurarse que se conecte en el sentido correcto. La figura 63(b) muestra un servo conectado al puerto 12 de forma que el cable negro (*black*) coincida con el rótulo *black* y que el cable rojo (*red*) coincida con el rótulo *red*.

Figura 63. (a) Esquema de los puertos para servos en la placa de educación. (b) Ejemplo de conexión de un servo a un puerto.



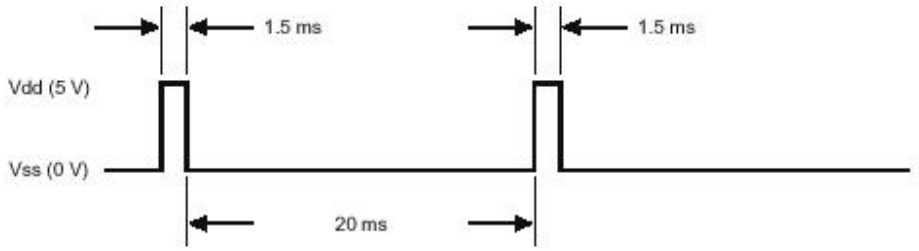
Luego de conectar el servo, a la PDE, entonces se conecta el porta-pilas, la figura 64 muestra como quedan los elementos después de esta operación.

Figura 64. Servo y porta-pilas conectados a la plaqueta de educación.



La señal de control que recibe el servo es llamada “tren de pulsos”, un ejemplo de tren de pulsos se muestra en la figura 65. El *Basic Stamp* puede ser programado para producir esta forma de onda en cualquiera de sus pines de E/S. En la siguiente actividad, se usara el pin de E/S P12, que ya se encuentra conectado al puerto de servos 12 por un trazo metálico de la Plaqueta de Educación. Primero, el *Basic Stamp* fija la tensión de P12 a 0 V (estado bajo) por 20 ms. Luego, fija la tensión de P12 a 5 V (estado alto) durante 1.5 ms. luego, repite el ciclo con un estado bajo por 20 ms y una salida en estado alto por 1.5 ms y así sucesivamente.

Figura 65. Ejemplo de tren de pulsos enviado a un servo.



Este tren de pulsos tiene un tiempo de encendido de 1.5 ms y un tiempo de apagado de 20 ms. El tiempo de encendido o de estado alto se suele llamar ancho del pulso. Cuando se habla de pulsos se sobreentiende que son pulsos positivos. Los pulsos negativos se toman como tiempo de descanso o separación entre pulsos positivos.

Luego de que un servo es modificado, se le pueden enviar pulsos para hacerlo girar constantemente. Los anchos de pulsos para los servos modificados típicamente oscilan entre 1.3 y 1.7 ms. para velocidad máxima en sentido horario y antihorario respectivamente, es decir si se desea que el servo gire continuamente en sentido horario se debe enviar un tren de pulsos con ancho de pulso de 1.3 ms., y si se desea que el servo se gire en sentido antihorario entonces se debe enviar un tren de pulsos con ancho de pulso de 1.7 ms. El ancho del pulso central típico es 1.5 ms y un servo modificado y calibrado correctamente debería permanecer estacionario cuando recibe pulsos de 1.5 ms. Si gira muy lentamente en respuesta a estos pulsos puede realizarse un ajuste por software. Si gira rápidamente con los mismos pulsos, el servo deberá ser desarmado y recalibrado.

En el caso de los servos utilizados para la locomoción de *Samy-Bot*, ya vienen modificados y calibrados de fábrica. Para el motor izquierdo, se debe usar un ancho de pulso de 1,504 ms para que se quede estático, para el motor derecho, se debe usar un ancho de pulso de 1,470 ms.

La importancia de localizar los anchos de pulsos en los cuales los motores se quedan estáticos radica en que estos anchos de pulsos se toman como base para producir movimientos en sentido horario o antihorario, lo único que se debe hacer es decrementar o incrementar los anchos de pulsos a partir del pulso en el cual el motor se queda estático. Para enviar un pulso al servo, se utiliza el comando *pulsout*, la sintaxis de este comando es como sigue:

pulsout <pin>,<ancho de pulso/2>

Para enviar un tren de pulsos a un servo que este conectado en el pin 12 del microcontrolador se utiliza la siguiente el siguiente código:

```
low 12
bucle:
  pulsout 12,750
  pause 20
goto bucle
```

La explicación de este código es como sigue:

low 12= Ajusta el pin 12 del microcontrolador como salida baja.
bucle: = Etiqueta hacia donde saltar.
pulsout 12,750
pause 20 = Envía pulsos de 1.5 ms por el pin 12 del microcontrolador cada 20 ms.
goto bucle = Salta hacia la etiqueta "*bucle*:".

5.3.4. Sistema de plataforma mecánica

5.3.4.1. Componentes del sistema de plataforma mecánica

Los componentes que se utilizarán para esta tarea son los siguientes:

1 Chasis
4 Separadores
4 Tornillos 1/4” 4-40
2 Pasa cables de goma 9/32”
1 Pasa cables de goma 13/32”

Estos componentes se muestran en la figura 66.

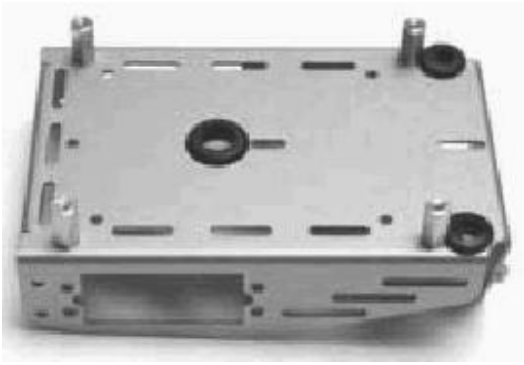
Figura 66. Componentes de la plataforma mecánica.

5.3.4.2. Montaje

El dibujo de abajo muestra al hardware superior montado en el chasis de *Samy-Bot*. Cada pasa cables de goma tiene una ranura en su cara exterior que la mantiene sujeta al agujero del chasis.

1. Se inserta la goma pasa cables de 13/32” en el agujero central del chasis.
2. Se inserta las otras dos gomas de 9/32” en los agujeros de las esquinas como se muestra en la figura 67.
3. Se usa los cuatro tornillos 1/4” 4-40 para colocar los cuatro separadores.

Figura 67. Plataforma mecánica ensamblada.



5.3.5. Montaje del sistema de locomoción

La lista de componentes que se utilizarán en esta parte son los siguientes:

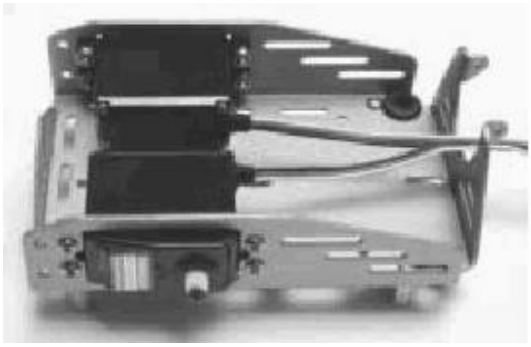
El chasis armado previamente
2 Servos
8 Tornillos 3/8" 4-40
8 Tuercas 4-40

Estos componentes se muestran en la figura 68.

Figura 68. Componentes necesarios para ensamblar los servos.

Los ocho tornillos 3/8” 4-40 y tuercas se utilizaron para fijar los servos al chasis. Tal como se muestra en figura 69.

Figura 69. Servos ensamblados.



5.3.6. Sistema de alimentación de energía

5.3.6.1. Componentes

Los componentes que se utilizarán en esta parte son los siguientes:

Chasis parcialmente armado.

1 Porta pilas vacío

2 Tornillos cabeza plana 4-40

2 Tuercas 4-40

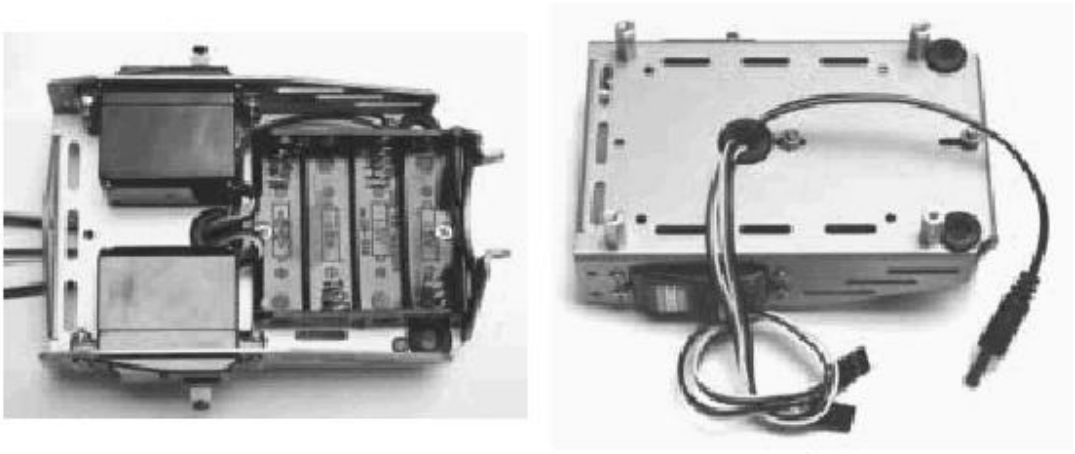
Estos componentes se muestran en la figura 70.

Figura 70. Componentes necesarios para ensamblar el porta-pilas.

5.3.6.2. Montaje

- Se utilizan los tornillos cabeza plana y las tuercas para colocar el porta pilas en la parte inferior del chasis como se muestra la figura 71(a).
- Se pasa el cable del porta pilas a través de la goma pasa cables más grande, ubicada en el centro del chasis.
- Se pasa los cables de los servos por el mismo agujero.
- Se acomodan los cables de los servos y de alimentación como se muestra en la figura 71(b). El cable de alimentación debe salir hacia el frente de *Samy-Bot* entre los separadores y los cables de los servos.

Figura 71. (a) Porta-pilas ensamblado. (b) Cables de los servos y del porta-pilas.



a b

5.3.7. Montaje del sistema de control al chasis de *Samy-Bot*

Los componentes que se utilizarán en esta parte son los siguientes:

1 Plaqueta de Educación con *BASIC Stamp* 2.
4 tornillos 1/4” 4-40.

Estos componentes se muestran en la figura 72.

Figura 72. Componentes necesarios para colocar el sistema de control a *Samy-Bot*.

5.3.7.1. Montaje

- Se debe asegurar que la *protoboard* blanca de la Plaqueta de Educación quede hacia el frente de *Samy-Bot* como se muestra en la figura 73.
- Se utilizan los cuatro tornillos 1/4” para sujetar la Plaqueta de Educación a los separadores.
- Se conecta el servo derecho de *Samy-Bot* en el puerto 12 y el izquierdo en el puerto 13. Para saber cuál es el servo derecho y cual el izquierdo, se toma como referencia la figura 73. El servo derecho de *Samy-Bot* es el que se muestra y el izquierdo está del otro lado del chasis (no se muestra).

Figura 73. Sistema de Control montado al chasis de *Samy-Bot.*

5.3.8. Ensamblando las ruedas al sistema de locomoción

Los componentes que se utilizarán en esta parte son los siguientes:

1 *Samy-Bot* parcialmente armado.
1 Pasador (chaveta) 1/16”.
2 Cubiertas de goma.
1 Bolilla de polietileno de 1”.
2 Llantas de plástico.
2 Tornillos que originalmente sujetaban la palanca de mando del servo.

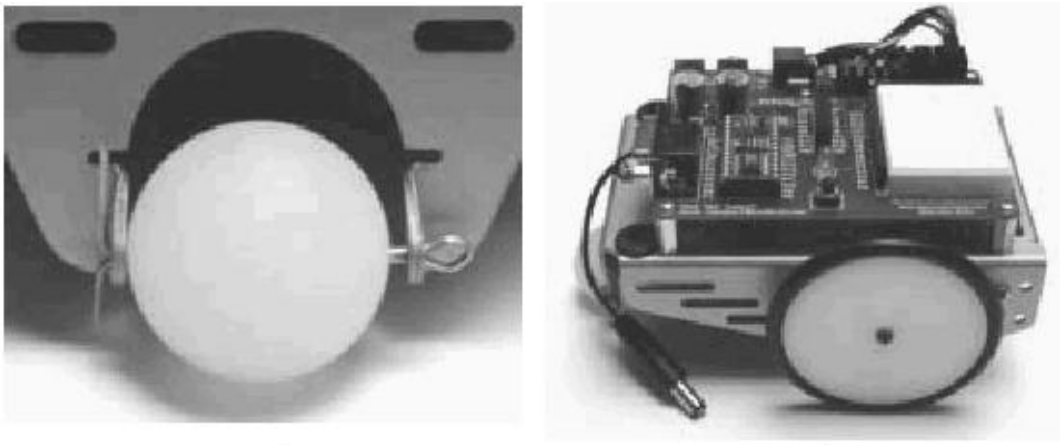
Estos componentes excepto el primero, se muestran en la figura 74.

Figura 74. Ruedas de *Samy-Bot.*

5.3.8.1. Montaje

- La bolilla plástica es usada como rueda trasera o de arrastre y la chaveta es su eje. Para evitar que la chaveta que sujeta la bolilla al chasis se salga, se separan sus puntas una vez colocada, como se muestra en la figura 75(a).
- Se colocan las cubiertas de goma en las llantas de plástico.
- Cada rueda debe tener un rebaje que coincide con el engranaje de salida de los servos. Una vez que la rueda y el engranaje estén alineados, se presiona firmemente hasta que el engranaje se introduzca en la rueda.
- Se usan los tornillos, para sujetar las ruedas en su lugar. El resultado de esta operación se muestra en la figura 75(b).

Figura 75. (a) Rueda trasera de *Samy-Bot*. (b) Ruedas ensambladas en *Samy-Bot*.



5.3.9. Ensamblado el sistema de visión

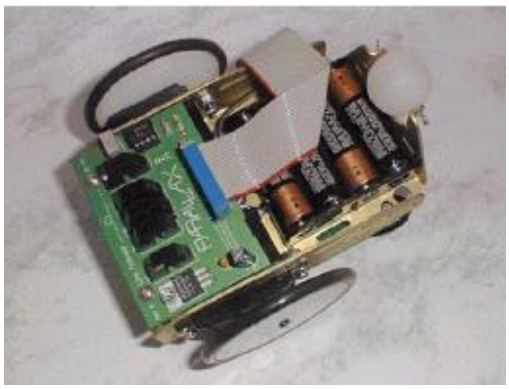
Los componentes que se utilizarán en esta parte son los siguientes:

1 Módulo de sensores infrarrojos seguidores de líneas de *Parallax* número de parte 29115.
1 Módulo de sensores medidores de distancia por infrarrojos de *Gazbot* GIDS01.
1 Cable plano de 10 pares con terminadores macho-macho.
Samy-Bot parcialmente armado.

5.3.9.1. Montaje

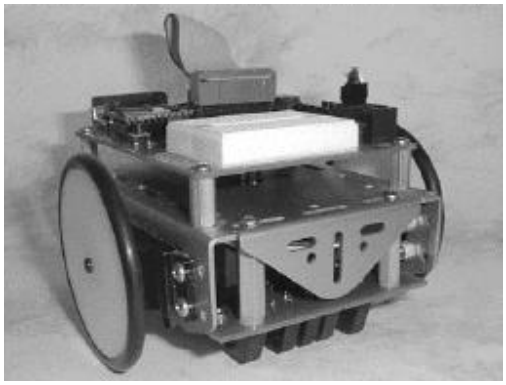
- El módulo seguidor de líneas se debe fijar con tornillos a la parte de abajo del chasis del robot, justo en la parte donde están las ruedas, como se muestra en la figura 76. Este módulo es un circuito diseñado para detectar líneas negras sobre fondo blanco o viceversa.
- El cable plano de 10 pares se conecta al módulo, tal como se muestra en la figura 76.

Figura 76. Módulo seguidor de líneas ensamblado al chasis de *Samy-Bot.*



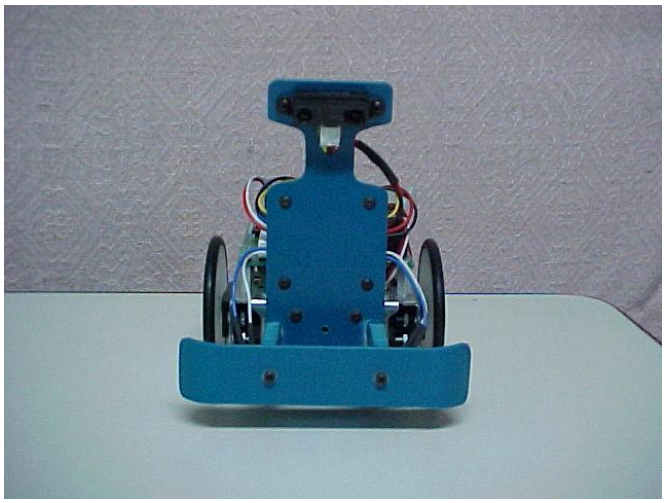
- El otro extremo del cable plano se conecta a la plaqueta de educación, tal como se muestra en la figura 77.

Figura 77. Módulo seguidor de líneas conectado a la plaqueta de educación.



- El módulo medidor de distancia viene adherido a una especie de carátula diseñada por *Gazbot*, viene diseñado para que se acople perfectamente a robots de tipo vehículo como *Samy-Bot*. El módulo se conecta en la parte de enfrente del robot, tal como se muestra en la figura 78.

Figura 78. Módulo detector de paredes ensamblado al chasis de *Samy-Bot*.



- El módulo medidor de distancia tiene 7 cables que deben ser insertados en la plaqueta de educación de *Samy-Bot*, dos de ellos se utilizan para llevar corriente y tierra para alimentar los sensores, y los otros 5 son para enviar y recibir información del sensor. La figura 79 muestra una vista lateral de *Samy-Bot* con el módulo conectado. La plaqueta de educación tiene una *protoboard* para armar circuitos, a un lado del *protoboard* tiene un *socket* de 16 agujeros, cada agujero etiquetado desde P0 a P16, estas etiquetas representan los pines, del *Basic Stamp* 2, a los que va conectado cada uno. Este *socket* se va a utilizar para conectar sensores o circuitos armados en el *protoboard*, a los pines del microcontrolador, en este *socket* se va a conectar los cables de control del módulo medidor de distancia, tal como se muestra en el dibujo de abajo. En la parte de arriba del *protoboard* hay otro *socket* que provee corriente y tierra para los circuitos que se armen en el *protoboard*, en este *socket* se va a conectar los cables de corriente y tierra del módulo, tal como se muestra en la figura 80.

Figura 79. Vista lateral de *Samy-Bot* con todos sus componentes ensamblados.

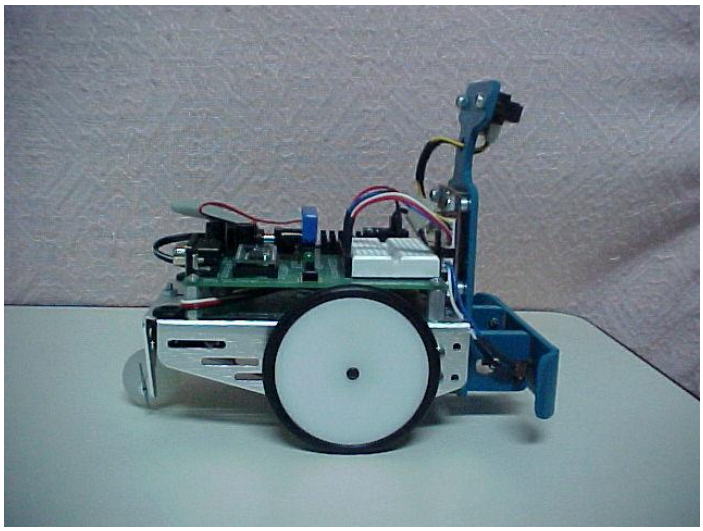
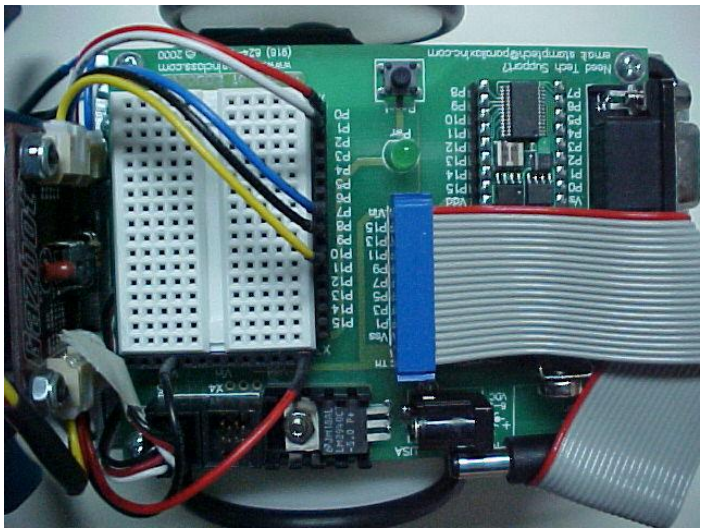


Figura 80. Módulo detector de paredes conectado a la plaqueta de educación.



En este punto *Samy-Bot* ya cuenta con un sistema de locomoción, un sistema de visión, un sistema de alimentación de energía, una plataforma mecánica, una plataforma electrónica y un sistema de control. En éste capítulo no se profundiza mucho acerca de las bondades y propiedades de cada módulo que se utilizó para el sistema de visión de *Samy-Bot*. Los componentes más importantes son: el microcontrolador, los sensores para el sistema de visión y los motores. En el siguiente capitulo se programará el microcontrolador para que pueda enviar señales a los servos para que se muevan de forma coordinada para que los movimientos de *Samy-Bot* sean coherentes, también se programará el microcontrolador para que sea capaz de interpretar los datos obtenidos con sus dos sensores de visión.

5.4. Resumen

Cuando se está construyendo un robot, es mejor imaginarlo como un conjunto de sistemas, subsistemas y elementos básicos, los sistemas más elementales de un robot son los siguientes:

- Un sistema de sensores para obtener información de su entorno.
- Un sistema de locomoción, para poder interactuar con su entorno.
- Un sistema de control que procese la secuencia "Entrada/Proceso/Salida".

Una de las actividades más importantes, cuando se hace un robot, es el desarrollo y prueba de cada subsistema individual, de un sistema dado. Luego, también deben realizarse pruebas a nivel del sistema, para asegurarse que todos los subsistemas trabajan conjuntamente en la forma que se esperaba. Por último, pero no menos importante, se realiza la integración de sistemas, asegurándose que los mismos funcionen coordinadamente. La prueba y la solución de problemas en cada fase del desarrollo, en niveles de sistema y subsistema es, hasta cierto punto, una habilidad que se perfecciona con la práctica.

En el caso de *Samy-Bot* el primero en ser instalado y probado fue su sistema de control el cual consta de una Plaqueta de Educación y un microcontrolador *Basic Stamp* 2, la Plaqueta de Educación es una placa con un circuito impreso, viene equipada con *sockets* y puertos para conectar los sensores y servos al microcontrolador, el microcontrolador es un dispositivo que puede ser capaz de ejecutar programas grabados en él y comunicarse con el exterior a través de sus pines, el microcontrolador va incrustado en la Placa de Educación la cual trae un *socket* especialmente diseñado para el *Basic Stamp*.

Luego se instaló el sistema de locomoción, el cual consta de dos servomotores conectados a los puertos 12 y 13 de la Plaqueta de Educación. Estos servos le darán la libertad de movimiento a *Samy-Bot*.

Finalmente se instaló el sistema de visión que esta compuesto por dos módulos: un módulo seguidor de líneas fabricado por *Parallax* Inc. y un módulo medidor de distancias fabricado por *Gazbot*. Estos sistemas guiaran a *Samy-Bot* a través de los laberintos.

6. IMPLEMENTACIÓN DE INTELIGENCIA ARTIFICIAL EN SAMY-BOT

6.1. Introducción

En este capítulo se mostrará paso a paso como se fue desarrollando el software que controla los sistemas de *Samy-Bot*. Como primer punto se hará una breve descripción del comportamiento deseado en *Samy-Bot*, así como de las características físicas de los ambientes en los cuales se va a desplazar.

Se definirá una arquitectura a seguir en la construcción del software, tomando como base la estrategia de Búsqueda Preferente por Profundidad estudiada en el capítulo 3 de este trabajo de investigación. En el transcurso de la explicación de cada módulo que compone el software se va ha realizar una explicación en detalle de cómo funciona el hardware que esta relacionado con cada rutina.

Los módulos principales en los cuales se va a dividir el software son los siguientes:

- Módulo de navegación
- Módulo de visión
- Sistema de Control

Conforme se vaya avanzando en el desarrollo de cada módulo se irá haciendo pruebas para determinar su funcionalidad; tomando en cuenta que es un sistema en tiempo real es necesario que el proceso de programación incluya muchas pruebas y calibración de cada elemento que componen a *Samy-Bot*.

Finalmente, cuando el sistema de control este completamente diseñado y programado, se mostrará el código fuente, así como una serie de imágenes que muestran a Samy-Bot ejecutando el programa.

6.2. Descripción del comportamiento de *Samy-Bot*

Las características que identificarán a *Samy-Bot* es que se podrá desplazar dentro de un laberinto de forma inteligente, para ello se necesita desarrollar rutinas que se grabarán en el microcontrolador, estas rutinas deben permitir al robot moverse en los laberintos en busca de la salida. El robot debe ser capaz de identificar cuando ha quedado atrapado en un camino sin salida y regresar a explorar el laberinto hasta encontrar la meta.

Se deberá proveer al robot con la capacidad de aprender o memorizar la ruta que lo llevo a la meta del laberinto, para comprobar esto, cuando *Samy-Bot* llegue a la meta del laberinto, entonces deberá regresar a su posición inicial dentro del laberinto. *Samy-Bot* regresará a su posición inicial utilizando la ruta exacta que lo llevo a la meta, de tal forma que ya no tendrá que explorar todo el laberinto.

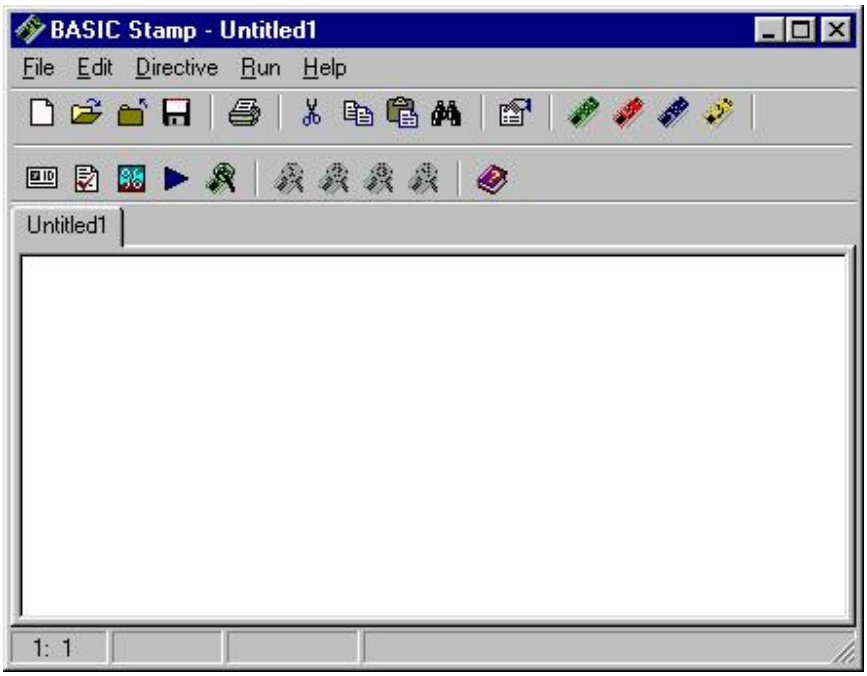
Las características de los laberintos en los que se desplazará *Samy-Bot* son las siguientes:

- Las paredes debe ser de color claro, de preferencia color blanco.
- Los ángulos de las esquinas de las paredes son de 90 grados.
- Las rutas dentro del laberinto están marcadas por líneas negras en el centro de las rutas, estas líneas se cruzan en cada cruce de camino.

6.3. Herramienta de software

Para desarrollar todas las rutinas se va a utilizar un software llamado *BASIC Stamp Editor* versión 1.32, en figura 81 se muestra el entorno del programa.

Figura 81. Entorno de desarrollo del *Basic Stamp Editor*.



El *Basic Stamp Editor* es un programa que viene con los microcontroladores *Basic Stamp* de *Parallax Inc*. Es un programa que permite escribir, compilar y grabar programas para los microcontroladores *Basic Stamp*. Los programas son escritos en un lenguaje llamado *Pbasic*, el cual es una variante del lenguaje *BASIC* desarrollado para los microcontroladores de *Parallax*, de allí que la P de *Pbasic* significa *Parallax*, para un conocimiento más profundo de este lenguaje consulte el libro ***BASIC Stamp Programming Manual Version 2.0*** escrito por *Parallax Inc*.

Cuando se ha escrito un programa es necesario verificar su sintaxis antes de grabarlo en el microcontrolador, esta operación se realiza seleccionando la opción ***Check Sintax*** del menú ***Run*** del Editor de *Basic Stamp*.

Finalmente para grabar el programa en la memoria del microcontrolador se debe conectar el microcontrolador al puerto com de la computadora tal como se indica en el capítulo anterior, luego se debe seleccionar la opción ***Run*** del menú ***Run*** del *Basic Stamp Editor*.

6.4. Arquitectura del software de *Samy-Bot*

El software de *Samy-Bot* puede analizarse y diseñarse utilizando tres módulos principales, estos módulos son los siguientes:

- Módulo de navegación del ambiente: aquí están agrupadas las rutinas que permitirán el desplazamiento de *Samy-Bot* dentro de los laberintos.
- Módulo de Visión: aquí están agrupadas las rutinas que permitirán a *Samy-Bot*, obtener información de su entorno.
- Módulo de toma de decisiones: aquí están agrupadas las rutinas que le permitirán a *Samy-Bot*, tomar sus propias decisiones basándose en la información recabada por su sistema de visión.

6.4.1. Módulo de navegación del ambiente

En esta parte se define como se programarán las rutinas que utilizará *Samy-Bot* para indicarle a los servos los movimientos que deben hacer para avanzar en la dirección deseada. Para las rutinas de navegación, se utilizaran los comandos *pulsout* y pause para generar un tren de pulsos, como se explicó en el capítulo anterior para que los servos se muevan es necesario enviar un tren de pulsos hasta lograr el movimiento deseado, entre cada pulso se deben hacer pausas de 20 ms según recomendaciones del fabricante de los servos. El ancho de los pulsos depende de la rapidez que se desea en el movimiento. El comando *pulsout* se utiliza para generar el pulso. La sintaxis del comando es:

> *pulsout* <pin del microcontrolador>,<mitad del ancho de pulso deseado medida en microsegundos>
>
> Por ejemplo: *pulsout* 1,500

Este comando hace exactamente lo que su nombre indica. Crea un pulso único de salida en el pin de E/S P1. El número "500" es un valor que determina la duración del pulso. Como se explico antes, ésta duración es medida en microsegundos. *Pulsout* tiene una duración de 2 microsegundos, por lo tanto un valor de 500, dará un pulso con una longitud de 500 veces 2 microsegundos, o 1000 microsegundos. Un valor de 1,000 creará un pulso con una longitud de 1,000 x 2 microsegundos = 2 ms.

Antes de continuar se debe aclarar que existe un ancho de pulso, el cual al ser enviado al servo no produce ningún movimiento, a este ancho de pulso se le llamará ancho de pulso central, este pulso central es muy importante debido a que a partir de él se puede ir incrementando y se logrará que el servo gire en sentido contrario al movimiento de las agujas del reloj, si por el contrario se decrementa a partir del pulso central entonces el servo va a girar en el sentido del movimiento de las agujas del reloj.

Los pulsos centrales para los servos que se utilizaron en *Samy-Bot* se muestran en la tabla IV, estos pulsos fueron calculados en base a prueba y error, es decir se probaron diferentes anchos de pulsos hasta que los servos se quedaron estáticos. Existen varios anchos de pulsos que pueden producir éste efecto en los servomotores, por esa razón se anoto el primer ancho de pulso en donde los servos se quedaron parados y se continuo variando hasta que los servos se movieron de nuevo, se anoto el ancho de pulso anterior al pulso que produjo el movimiento, se calculo el promedio entre el primer y ultimo ancho de pulso que no produjo movimiento y este fue el ancho de pulso central.

Tabla IV. Ancho de pulsos centrales de los servos.

Servomotor	Ancho del pulso central
Derecho	1.504 ms
Izquierdo	1.470 ms

6.4.1.1. Movimiento hacia adelante

Para esta rutina se necesita que el servo derecho gire en el sentido del movimiento de las agujas del reloj a la vez que el servo izquierdo lo hace en sentido contrario al movimiento de las agujas del reloj. La rutina para mover a *Samy-Bot* hacia delante se muestra en la figura 82.

Figura 82. Código fuente del movimiento hacia adelante.

```
avanzar: 'rutina para ir hacia adelante
pulsout motorIzquierdo,centralIzquierdo+avanzaIzquierdo
pulsout motorDerecho,centralDerecho-avanzaDerecho
goto rutinaPrincipal
```

Este código envía un pulso a cada servo de *Samy-Bot*, esto produce un pequeño avance hacia adelante, no es necesario hacer una pausa debido a que el proceso de analizar el estado del robot es suficiente pausa.

6.4.1.2. Giros hacia la derecha

Existen dos tipos de *Samy-Bot* puede hacer hacia la derecha.

- **Giro fuerte**

Este giro debe hacerse cuando *Samy-Bot* se ha desviado mucho hacia la izquierda, y es necesario que gire hacia la derecha lo más pronto posible para volver a su ruta correcta. Durante este giro el servo de la izquierda trata de mover a *Samy-Bot* hacia adelante mientras el derecho mueve a *Samy-Bot* hacia atrás, de tal forma que gira hacia la derecha sin avanzar. En la figura 83 se muestra el código fuente de esta rutina.

Figura 83. Código fuente del giro fuerte hacia la derecha.

```
giroDerecho: 'rutina para girar con fuerza hacia la derecha
pulsout motorIzquierdo,centralIzquierdo+100
pulsout motorDerecho,centralDerecho+100
goto rutinaPrincipal
```

- **Giro suave**

Este giro se utiliza cuando *Samy-Bot* se ha desviado una pequeña fracción hacia la izquierda y es necesario hacerlo volver a su ruta correcta. Durante este giro el motor izquierdo trata de mover a *Samy-Bot* hacia delante mientras el derecho se queda estático, de tal forma que gira hacia la derecha. El código fuente de esta rutina se muestra en la figura 84.

Figura 84. Código fuente del giro suave hacia la derecha.

```
aLaDerecha:    'rutina para un giro suave hacia la derecha
  pulsout motorIzquierdo,centralIzquierdo+100
  pulsout motorDerecho,centralDerecho
  goto rutinaPrincipal
```

6.4.1.3. Giros hacia la izquierda

- **Giro fuerte**

Este giro debe hacerse cuando *Samy-Bot* se ha desviado mucho hacia la derecha, y es necesario que gire hacia la izquierda lo más pronto posible para volver a su ruta correcta. Durante este giro el servo de la izquierda trata de mover a *Samy-Bot* hacia atrás mientras el derecho mueve a *Samy-Bot* hacia adelante, de tal forma que gira hacia la izquierda. En la figura 85 se muestra el código fuente de esta rutina.

Figura 85. Código fuente del giro fuerte hacia la izquierda.

```
giraraLaIzquierda: 'rutina para girar fuerte hacia la izquierda
  pulsout motorIzquierdo,centralIzquierdo-100
  pulsout motorDerecho,centralDerecho-100
  goto rutinaPrincipal
```

- **Giro suave**

Este giro se utiliza cuando *Samy-Bot* se ha desviado una pequeña fracción hacia la derecha y es necesario hacerlo volver a su ruta correcta. Durante este giro el motor derecho trata de mover a *Samy-Bot* hacia delante mientras el izquierdo se queda estático, de tal forma que gira hacia la izquierda. El código fuente de esta rutina se muestra en la figura 86.

Figura 86. Código fuente del giro suave hacia la izquierda.

```
aLaIzquierda:   'rutina para girar suave hacia la izquierda
  pulsout motorIzquierdo,centralIzquierdo
  pulsout motorDerecho,centralDerecho-100
  goto rutinaPrincipal
```

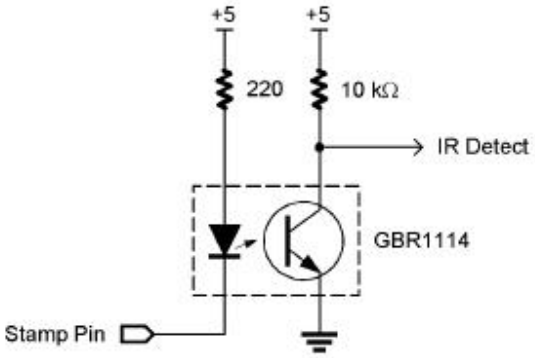
6.4.2. Módulo de Visión

El módulo de visión esta compuesto por dos módulos uno para seguir las líneas negras que marcan las rutas en los laberintos a este módulo se le llamará Seguidor de líneas, y otro módulo que servirá para detectar paredes. Ambos módulos ya vienen listos para ser controlados por el *Basic Stamp* 2.

6.4.2.1. Módulo seguidor de líneas

Este módulo está compuesto por un arreglo de cinco emisores y detectores infrarrojos alineados. La luz infrarroja tiene la propiedad de que no es reflejada por el color negro. Cada emisor y detector esta colocado en un ángulo de tal forma que cuando el emisor envía su luz, el detector registra la luz infrarroja; siempre y cuando la luz no haga contacto con una superficie de color negro. Esta propiedad se utilizará para detectar las líneas negras en el suelo que marcan las rutas dentro de los laberintos. Estos emisores y detectores están encapsulados en un solo dispositivo llamado Sensor infrarrojo por reflexión, el diagrama s estos sensores se muestra en la figura 87.

Figura 87. Diagrama de sensor infrarrojo por reflexión.



La línea etiquetada *Stamp Pin*, es la que va conectada a un pin del *Basic Stamp* para que éste envíe la orden de emitir luz infrarroja, la línea *IR Detect* también va conectada a un pin del *Basic Stamp* para registrar la luz infrarroja. Para utilizar éste sensor, el *Basic Stamp* envía la orden al sensor para que emita un rayo de luz, luego lee la línea *IR Detect* para saber si se ha detectado el reflejo de la luz, si se ha detectado quiere decir que está fuera de la línea negra, si no se registra ningún reflejo quiere decir que *Samy-Bot* va en el curso correcto.

El código para controlar este módulo se muestra en la figura 88.

Figura 88. Código fuente que controla el módulo seguidor de líneas.

```
LeerLineaDeCamino: 'rutina que lee líneas en el suelo
  lfbits = 0

  for ledpos = 2 to 6
    outl.lowbit(ledpos) = ledencendido
    pause 1
    lfbits.lowbit(ledpos) = in9^lfmode
    outl = outl | %01111100
  next
  lfbits=lfbits>>2
return
```

La parte principal de éste código es un ciclo que va de 2 a 6, debido a que se van a utilizar los pines del 2 al 6, del *Basic Stamp* 2, para conectar el módulo seguidor de líneas; en cada ciclo se activa un pin del *Basic Stamp* de tal forma que uno de los sensores infrarrojos se encienda, luego hace una pausa de 1 ms. y a continuación lee el módulo para ver si se ha detectado el reflejo de la luz, todas las líneas *IR Detect* del módulo seguidor de líneas están conectadas al pin 9 del *Basic Stamp*. Esta entrada pasa por un operador XOR antes de ser almacenado en el registro *lfbits*, esto se hace porque la entrada es cero cuando se detecta una línea negra, entonces es necesario hacer un XOR de esta entrada con el valor 1 lógico de tal forma que transforme las entradas uno a cero y las entradas cero a uno.

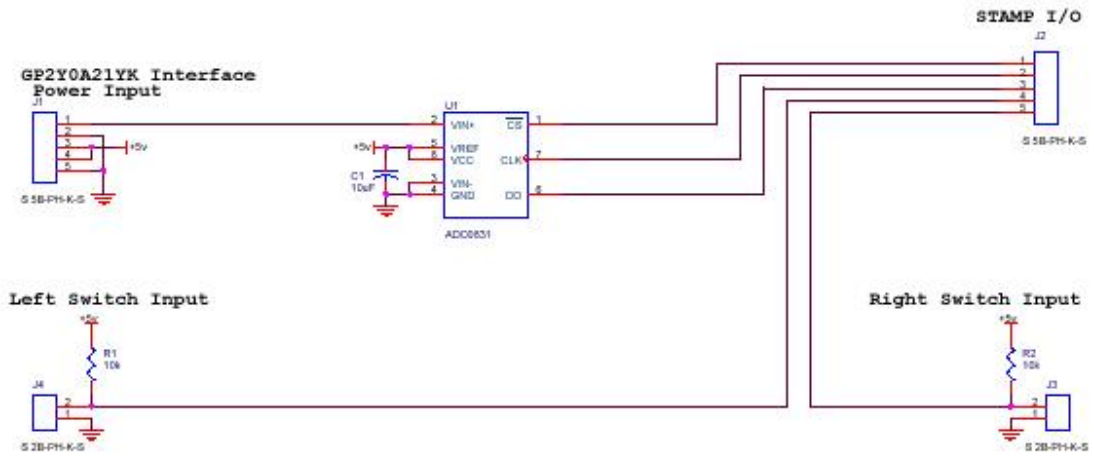
En la figura 89 se muestra a *Samy-Bot* ejecutando el módulo seguidor de líneas.

Figura 89. *Samy-Bot* ejecutando el módulo seguidor de líneas.

6.4.2.2. Módulo detector de paredes

Este módulo está compuesto por un Circuito Integrado código ADC0831 y un sensor emisor detector de luz infrarroja, el ADC0831 es el encargado de controlar el sensor infrarrojo, primero le ordena al emisor que dispare un rayo de luz y luego analiza la señal que percibe el detector para poder establecer la distancia a la que se encuentra el objeto, éste sensor funciona igual que los sensores del seguidor de líneas el emisor dispara el rayo de luz, y si choca con una superficie entonces la luz rebota y es recibida por el detector. Para utilizarlo en *Samy-Bot* no es necesario establecer la distancia del objeto, es suficiente con saber si hay un objeto o no adelante del robot, esto le indicará si hay pared o no hay. En la figura 90 se muestra el diagrama del módulo detector de paredes.

Figura 90. Diagrama del módulo detector de paredes.



El código para controlar éste módulo se muestra en la figura 91.

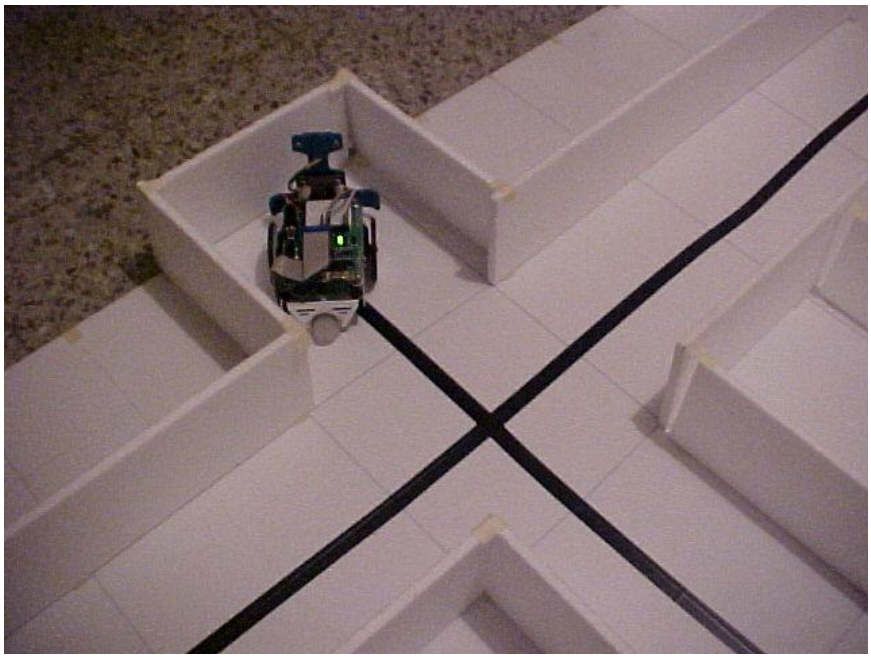
Figura 91. Código fuente que controla el módulo detector de paredes.

```
'---------------- constantes del sensor de paredes ------------
CS                                  CON         0' ADC0831 chip select
CLK                                 CON         1' ADC0831 línea del reloj
D0                                  CON         7' ADC0831 línea de datos
'---------------- variables del sensor de paredes -------------
adcbits                     VAR     byte' almacena lo que venga en la línea de datos


gosub verpared
if adcbits > 100 then nohaycaminoenfrente 'verifica si hay camino
    caminos.bit1=%1  'definiendo que caminos están disponibles
    rutas=1 'al haber camino en frente entonces hay al menos una ruta
.
.
.
goto rutinaPrincipal


verpared: 'rutina que se utiliza para detectar paredes
  High cs     '  desactiva el  ADC0831
  low cs      '  activa ADC0831.
  Low clk '   pone el reloj del ADC0831 en estado bajo.
  pulsout CLK,210         '   inicia el pulso.
  shiftin d0,clk,msbpost,[adcbits\8] 'lee el valor registrado por el módulo
Return
```

Esta rutina, primero desactiva el chip ADC0831, luego lo vuelve a activar, inicializa el reloj y le envía un pulso 0.42 ms., esta es la señal para que el módulo detector de paredes empiece a trabajar, finalmente utiliza el comando *shiftin* para leer el valor obtenido por el módulo y lo almacena en la variable adcbits, entre más grande sea el valor devuelto entonces más cerca está de una pared. Realizando pruebas se ha detectado que si el valor en adcbits es mayor que cien entonces con toda seguridad existe una pared enfrente de *Samy-Bot* y por lo tanto no hay camino.

En figura 92 se muestra a *Samy-Bot* ejecutando el módulo de detección de paredes.

Figura 92. *Samy-Bot* detectando un camino sin salida.



6.4.3. Módulo de toma de decisiones

Este es el módulo más importante, aquí están agrupadas todas las rutinas que se necesitan para mantener el control de todo el sistema de *Samy-Bot*, antes de mostrar el código se va a describir en palabras sencillas la forma en que funciona en *Samy-Bot* la toma de decisiones.

Antes de hacer cualquier cosa lo primero que *Samy-Bot* busca es alinearse con una línea negra que busca en el suelo, de allí en adelante se dedica a explorar todas las rutas hacia donde las líneas negras lo guíen, lo mas seguro es que va a encontrar un cruce de rutas, aquí es donde comienza a aplicar la estrategia de búsqueda que se ha seleccionado, la cual es Búsqueda Preferente por Profundidad.

Siguiendo la estrategia de Búsqueda Preferente por Profundidad, cuando *Samy-bot* llega a un cruce de rutas, lo analiza para saber cuantos posibles caminos hay a partir del cruce, luego almacena en su memoria la información recabada acerca del cruce y comienza a explorar siempre el camino que esta hacia la derecha en caso de no existir entonces hecha mano del camino hacia delante, si este no existiera entonces toma la ruta hacia la izquierda. Y así lo va haciendo en todos los cruces de ruta hasta que el camino se le acabe. Cuando la ruta termina entonces *Samy-Bot* realiza una comprobación para saber si ha llegado al final de un camino sin salida o ha llegado a la meta, en esta parte es donde utiliza el módulo detector de paredes. Cuando detecta que es un camino sin salida entonces da un giro de 180 grados y regresa al cruce anterior para seguir explorando las rutas restantes. En caso de que haya localizado la meta entonces regresa hacia su posición inicial para mostrar que ha aprendido el camino que lleva hacia la meta.

El código fuente de este módulo se muestra en la figura 93.

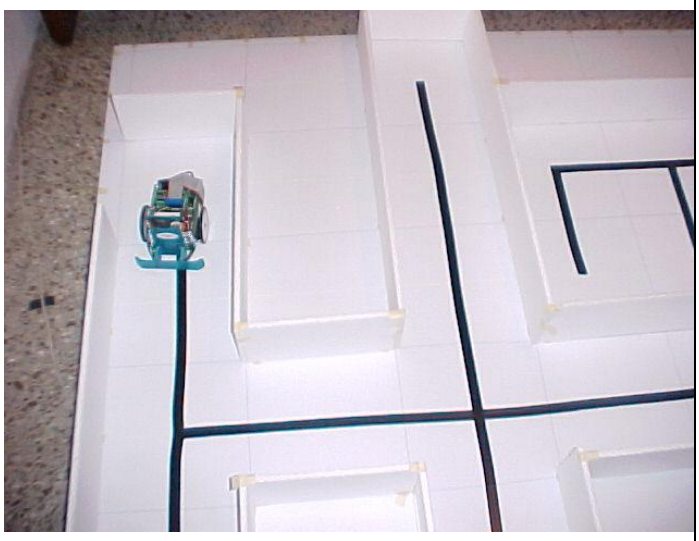
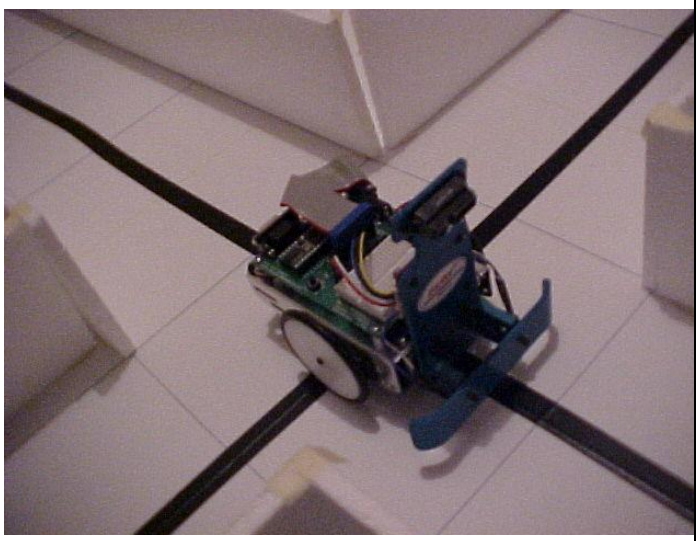
Figura 93. Código fuente del sistema de control.

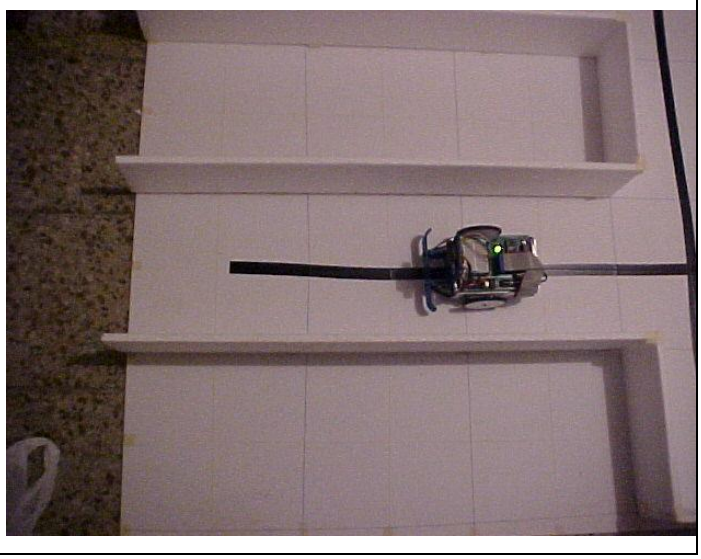
```
RutinaPrincipal:
  gosub leerLineaDeCamino
  if lfbits=%0 or lfbits=%11111 then tomaDecision
  oncourse = yes


tomaDecision:
  lookdown lfbits, =
[%00000,%00001,%00011,%00111,%00110,%01110,%01100,%11100,%11000,%10000,%11110,%011
,%11111],steer
  branch steer,
[fueraDeLinea,giroDerecho,giroDerecho,aLaDerecha,aLaDerecha,avanzar,aLaIzquierda,aLaIzquierda,girar
aLaIzquierda,giraraLaIzquierda,cruce,cruce,cruce]


cruce: 'aqui se analiza los cruces de los caminos
 if oncourse = no then rutinaPrincipal
  if explorando=yes then procesarcruce 'en caso de que sea un cruce nuevo vaya a procesarcruce
  'else  si viene de regreso ya no tiene que procesar el cruce
    if rutasexploradas=rutas then cruceanterior 'si ya exploro todas las rutas del cruce
    'else si todavia no ha explorado todas las rutas del cruce
       rutasexploradas=rutasexploradas+1
```

```
    explorando=yes
    lookdown rutatomada, = [%001,%010],steer
    branch steer, [regresadederecha,regresadefrente]

    regresadederecha: 'si regresa de la derecha
      if distribucion.bit1=%1 then doblarderecha
        gosub seguirdefrente
        rutatomada=%100
        goto goahead
      doblarderecha:
        gosub tomarsiguienterutaaladerecha
        rutatomada=%010
        goto goahead

    regresadefrente: 'si regresa del frente
     gosub tomarsiguienterutaaladerecha
     rutatomada=%100

    goahead:
      gosub sobreescribirnodo

    goto rutinaPrincipal

 cruceanterior:'rutina para volver al cruce anterior cuando se exploraron ya todas las rutas del cruce actual

   lookdown rutatomada, = [%001,%010,%100],steer
   branch steer, [iralaizquierda,irdefrente,iraladerecha]
   iralaizquierda:
     gosub tomarsiguienterutaalaizquierda
   goto findecruceanterior
   irdefrente:
     gosub seguirdefrente
   goto findecruceanterior
   iraladerecha:
     gosub tomarsiguienterutaaladerecha
   findecruceanterior:
   gosub recuperarNodo
   goto rutinaPrincipal
procesarcruce:
 caminos       =0
 rutas         =0
 distribucion   =0
 rutasexploradas=0
 rutatomada     =0
 for contadorPulsos = 0 to 2 'Se mueve hacia adelante para poder leer bien el cruce
  pulsout motorIzquierdo,centralIzquierdo+avanzaIzquierdo
  pulsout motorDerecho,centralDerecho-avanzaDerecho
  pause 15
 next
 gosub leerLineaDeCamino
 gosub verpared
 if adcbits > 100 then nohaycaminoenfrente 'verifica si hay camino enfrente
```

```
    caminos.bit1=%1             'definiendo que caminos están disponibles
    rutas=1                   'al haber camino en frente entonces hay al menos una ruta

   nohaycaminoenfrente:
   caminos.bit0=lfbits.bit0
   caminos.bit2=lfbits.bit4

   if lfbits.bit0=%0 then intermedio 'comprueba si hay ruta a la derecha
    rutas=rutas+1               'si hay camino a la derecha entonces hay otra ruta
   intermedio:
   if lfbits.bit4=%0 then continuar 'comprueba si hay ruta a la izquierda
    rutas=rutas+1               'al haber camino a la izquierda entonces se suma otra ruta
   continuar:
   distribucion=caminos 'se define la distribución de rutas en el cruce
   rutasexploradas=1    'se pone que rutas exploradas es igual a 1 por que ahora se va a explorar
   if caminos=%111 or caminos=%001 or caminos=%011 or caminos=%101 then tomarcaminoderecha
   if caminos=%110 then tomarcaminoenfrente
   if caminos=%100 then tomarcaminoizquierda

   tomarcaminoderecha:
    rutatomada=%0001
    gosub almacenarNodo
    gosub tomarsiguienterutaaladerecha
    goto rutinaPrincipal
   tomarcaminoenfrente:
    rutatomada=%0010
    gosub almacenarNodo
    gosub seguirdefrente
    goto rutinaPrincipal
   tomarcaminoizquierda:
    rutatomada=%0100
    gosub almacenarNodo
    gosub tomarsiguienterutaalaizquierda
    goto rutinaPrincipal


comprobarsolucion: 'comprueba si el fin del camino es solucion o es un tope
  gosub verpared
  if adcbits > 100 then volver
 ' for contadorPulsos = 0 to 100 'Se encontro la salida, entonces avanza hacia afuera
 '   pulsout motorIzquierdo,centralIzquierdo+avanzaIzquierdo
 '   pulsout motorDerecho,centralDerecho-avanzaDerecho
 '   pause 15
 ' next
 gosub mostrarsolucion
 end 'encontro la salida entonces para
  volver: 'rutina que hace volver al robot para que continúe su búsqueda
    gosub tomarsiguienterutaaladerecha
    explorando=no
    goto rutinaPrincipal
```

A continuación se muestra una serie de imágenes que muestra a *Samy-Bot* ejecutando todo el módulo de control.

1) *Samy-Bot* localiza la línea negra y comienza la búsqueda, ver figura 94.	**Figura 94.** *Samy-Bot* iniciando su recorrido por un laberinto. 
2) *Samy-Bot* enfrentándose a un cruce de caminos, ver figura 95.	**Figura 95.** *Samy-Bot* analizando un cruce de caminos. 

3) *Samy-Bot* enfrentándose a un camino sin salida, ver figura 96.	**Figura 96.** *Samy-Bot* enfrentándose a un camino sin salida. 
4) *Samy-Bot* a punto de localizar la salida del laberinto, ver figura 97.	**Figura 97.** *Samy-Bot* saliendo de un laberinto. 

Así se concluye la explicación de la arquitectura del software que corre en el microcontrolador *Basic Stamp* 2 de *Samy-Bot*.

6.5. Resumen

En este capítulo se profundizó acerca de la programación de los microcontroladores *Basic Stamp 2* de *Parallax Inc*., el objetivo principal de este capítulo es hacer un análisis de cada módulo que compone el software de *Samy-Bot*. Los módulos de software principales de *Samy-Bot* son tres:

- Módulo de navegación del ambiente: aquí están agrupadas las rutinas que permitirán el desplazamiento de *Samy-Bot* dentro de los laberintos.
- Módulo de Visión: aquí están agrupadas las rutinas que permitirán a *Samy-Bot*, obtener información de su entorno.
- Módulo de toma de decisiones: aquí están agrupadas las rutinas que le permitirán a *Samy-Bot*, tomar sus propias decisiones basándose en la información recabada por su sistema de visión.

El módulo de navegación comprende las funciones:

- Movimiento hacia delante
- Giros hacia la derecha
- Giros hacia la izquierda

El módulo de Visión administra dos módulos de hardware:

- Módulo seguidor de líneas
- Módulo detector de paredes

En éste capítulo se mostró y explicó el código fuente para realizar cada una de las funciones de *Samy-Bot*, también de incluyeron imágenes de *Samy-Bot* ejecutando las diferentes rutinas.

CONCLUSIONES

1. La Inteligencia Artificial es una ciencia donde hay mucho campo para investigar, el panorama que aquí se mostró es uno de muchos en los cuales la Inteligencia Artificial puede ser aplicada. Para el caso del robot, de nombre *Samy-Bot*, se utilizaron los conceptos de Agente Inteligente, Ambiente y Estrategia de Búsqueda. Estos conceptos facilitaron el planteamiento del problema y el desarrollo de la solución.

2. Los microcontroladores son dispositivos que se han utilizado desde hace varios años, hasta ahora se habían utilizado en sistemas de control de electrodomésticos o en automóviles. Recientemente se ha comenzado a difundir en las universidades, información de cómo utilizarlos para construir robots. Dando como resultado una revolución en la forma de aprender electrónica y programación; en el caso de *Samy-Bot* se utilizó un microcontrolador *BASIC Stamp* 2, el cual tuvo un excelente rendimiento.

3. El advenimiento de una gran gama de sensores así como de los servomotores ha permitido avances impresionantes en el área de la Robótica, cada día salen al mercado sensores muy versátiles que pueden ser utilizados con facilidad. En el caso de *Samy-Bot* se utilizaron dos sensores uno para detectar líneas negras en el suelo y otro para detectar paredes.

RECOMENDACIONES

1. Los laberintos que resuelve *Samy-Bot* no tienen ciclos en sus rutas, esta característica hace que los laberintos sean manejables de una forma no muy compleja. Cuando se trata de laberintos cuyas rutas tienen ciclos entonces el problema se vuelve más complejo, para poder resolver estos laberintos la programación del robot debe cambiar para incluir módulos de control que sean capaces de identificar estos ciclos, esto es difícil, pero no imposible. Si se desea continuar un paso adelante de esta investigación, el siguiente paso será que el robot resuelva laberintos cuyas rutas tienen ciclos.

2. Los microcontroladores están evolucionando rápidamente, y la tendencia es a utilizar lenguajes de alto nivel para programarlos, al momento de terminar este trabajo de investigación ya se encuentran en el mercado microcontroladores que utilizan un interprete del lenguaje Java, lo cual permite utilizar un enfoque de programación orientado a objetos y también permite programación concurrente. Estos dos elementos serán de mucha utilidad si se desea programar comportamientos más complejos e inteligentes en el robot, como por ejemplo que pueda resolver laberintos con caminos que tengan ciclos y/o comunicarse con otros robots, de tal forma que pueda ser posible comunicar a otros entes su conocimiento adquirido.

BIBLIOGRAFÍA

1. **BASIC Stamp Programming ManualVersion 2.0**(Ingles). s.l. s.e. 2000. 341pp.

2. Gilliland, Matt. **¿Qué es un microcontrolador? Versión en Castellano 1.1.** s.l. s.e. 1999. 111pp.

3. González, Juan. **Diseño y construcción de un robot articulado que emula modelos de animales: aplicación a un gusano.** s.l. s.e. 2001. 220pp.

4. Lindsay , Andrew. **Robótica, versión en Castellano 1.5**. s.l. s.e. 2000. 197pp.

5. Lindsay, Andrew. **Analógico y Digital Básicos Versión 1.1 en Castellano**. s.l. s.e. 2002. 153pp.

6. Russell Stuart, Norvig Peter. **Inteligencia Artificial, Un enfoque moderno.** México: Editorial Prentice Hall. 1995.

7. Torres, Andres. **Diseño, construcción y control de un robot mediante una red de microcontroladores.** s.l. s.e. 2001. 191pp.

8. Williams, John. **StampWorks Experimentos y código fuente para el Basic Stamp Versión en Castellano 1.1.** s.l. s.e. 1999. 177pp.

Sitios en Internet consultados

1. **http://www.cursoderobotica.com.ar**
   http://www.cursoderobotica.com.ar/robotica/Libros/libros.htm

2. **http://www.microbotica.es**
   http://www.microbotica.es/web/mastall.htm
   http://www.microbotica.es/web/flota.htm
   http://www.microbotica.es/web/taller.htm
   http://www.microbotica.es/web/downl-doc.htm

3. **http://news.nationalgeographic.com**
   http://news.nationalgeographic.com/news/2001/09/0914_TVdisasterrobot.html

4. **http://www.robocup.org**
http://www.robocup.org/Intro.htm

5. **http://www.sec.upm.es**
http://www.sec.upm.es/agustin/charrito/charrito.html
http://www.sec.upm.es/agustin/hardarm/hardarm.html