UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE INGENIERÍA

TESIS

DISEÑO E IMPLEMENTACIÓN DEL ALGORITMO FLOOD-FILL PARA UN ROBOT MICROMOUSE

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO EN COMPUTACIÓN

PRESENTA:

LUIS SERGIO DURÁN ARENAS

DIRECTOR DE TESIS:

M.I. RUBÉN ANAYA GARCÍA

Septiembre 2013

CIUDAD UNIVERSITARIA, MÉXICO, D.F.

Agradecimientos

A mis padres: Que me han apoyado para lograr lo que he logrado y que siempre han estado a mi lado.

A mis hermanos: Con quienes crecí, viví y aprendí a ser mejor cada día.

A mis amigos: Que han estado conmigo en las buenas y en las malas, y con quienes pasé momentos inolvidables.

A mis profesores: De quienes obtuve el conocimiento para este logro, pero sobre todo, de quienes he aprendido que la vida coloca obstáculos que uno es capaz de afrontar y superar si se lo propone.

A mis sinodales: con quienes tuve el placer de compartir el aula de clases.

A la UNAM: Mi segunda casa a lo largo de los últimos años y a la que debo mucho.

A Dios.

Mención especial a Tomás: Quien me apoyó y aconsejó en la redacción de este trabajo.

En fin, gracias a todas y cada una de las personas que se han cruzado en mi vida, haya sido para bien o para mal.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 Introducción

1.1 Problemática

La problemática abordada en este trabajo es el diseño de un algoritmo capaz de integrarse a un robot autónomo, para hacer posible la resolución de un laberinto de paredes. Dicho algoritmo estará basado en uno ya existente llamado *Flood-Fill*.

El algoritmo *Flood-Fill*, o algoritmo de inundación, es un algoritmo que determina el área conectada a un nodo haciendo uso de matrices[1]. Esto es, divide el entorno en celdas multidimensionales (cada una de ellas es un nodo) y calcula las que se encuentran alrededor de una celda dada.

Se decidió realizar esta investigación referente a la Robótica debido a que en México este tema no ha tenido una difusión adecuada. En países como Japón, Singapur o Estados Unidos, los robots que resuelven laberintos de paredes comenzaron a desarrollarse desde la década de los 70's y la primera competencia fue realizada en los 80's.

1.2 Objetivo

El objetivo principal es "rediseñar" el algoritmo de inundación e implementarlo sobre un robot MicroMouse, el que resuelve laberintos de muros. Una vez teniendo el algoritmo, se diseñará un sistema autónomo que sea capaz de resolver el laberinto y que tendrá que cumplir con ciertas especificaciones del estándar IEEE para competiciones MicroMouse, e incluso que pueda participar en alguna competencia de tal índole[2].

Para cumplir con este objetivo, partiremos de la siguiente hipótesis:

Hipótesis: El algoritmo *Flood-Fill* es el algoritmo más eficiente para la resolución de un laberinto de paredes, independientemente de sus dimensiones, cuya solución o meta se encuentra en el centro del mismo.

1.3 Metodología

Existen especificaciones tanto para el desarrollo de sistemas de software como para sistemas de hardware. En nuestro caso, el sistema a desarrollar tendrá ambas partes, las

[1] Idea tomada a partir de lo mencionado en: *http://en.wikipedia.org/wiki/Flood_fill*
[2] Una explicación breve de las competencias para robots MicroMouse puede encontrarse en la *Sección 2.2*

cuales se podrán manejar por separado, aunque es necesario mencionar que una parte no funcionará sin la intervención de la otra.

Básicamente se tienen dos partes que componen el sistema a desarrollar: Software y Hardware. Cada parte de este proyecto deberá analizarse por separado y en una fase determinada deberán reunirse para poder realizar las pruebas convenientes y las correcciones que correspondan.

Considerando que el sistema a desarrollar se trata de un agente inteligente[3], existen ciertas metodologías a implementar para el desarrollo de sistemas de software orientados a agentes, las cuales se tratarán más adelante.

La metodología para cada pieza del sistema será la misma, siendo la siguiente para cada uno:

1.3.1 Análisis

La fase de análisis es quizá la más importante del proceso. Trata de precisar los límites del proyecto a construir y definir su originalidad; esto es, establecer las necesidades que deben ser cubiertas por el sistema y determinar si éstas se encuentran ya resueltas de manera adecuada a nuestro problema, con herramientas existentes en el mercado. [1]

Regularmente, el análisis consiste en realizar reuniones entre usuarios del producto y técnicos para concretar y definir el producto a partir de necesidades y requerimientos. Esta tarea suele ser complicada debido a los dos diferentes puntos de vista donde se encuentran los participantes. Generalmente, los usuarios no tienen los conocimientos técnicos de informática y los analistas informáticos suelen desconocer el área de trabajo hacia donde está destinada la aplicación. De manera paralela a este estudio de análisis de requerimientos, puede requerirse un estudio de mercado para conocer el entorno al que está dirigido el producto final con el fin de analizar los alcances de éste o la existencia de posibles competidores. [1]

1.3.2 Diseño

En esta etapa, la arquitectura software/hardware del sistema es delimitada. En ésta se toman decisiones técnicas acerca de las herramientas a emplear para la implementación del sistema y se utiliza una metodología concreta. Esta fase puede entenderse de distintas maneras de acuerdo al modelo del ciclo de vida del sistema que se escoja. [1]

[3] El concepto de agente inteligente se explica en la *Sección 2.3*.

1.3.3 Implementación

Esta fase consistirá en la programación del diseño de datos, bases de datos, etc., así como de los procesos definidos dentro del sistema. [1]

1.3.4 Pruebas

En esta etapase someterá al sistema a una serie de pruebas para garantizar el buen funcionamiento del mismo y obtener conclusiones al respecto. En muchas ocasiones, el objetivo de las pruebas es verificar el sistema en su totalidad, comenzando por obtener resultados independientes de cada uno de sus módulos (pruebas unitarias) y, una vez que éstas son satisfactorias, proseguir con las pruebas de integración del sistema (pruebas integrales). [1] [2]

Una de las maneras de realizar las pruebas es someter al sistema a una serie de situaciones comunes y no tan comunes en las que puede estar involucrado. Lo ideal sería realizar pruebas en todas las posibles situaciones en las que puede presentarse el sistema, sin embargo, las posibilidades son infinitas. En algunos campos existen conjuntos de pruebas estándar para sistemas que garantizan el buen funcionamiento del mismo, en caso de que las pruebas sean exitosas. [1]

1.3.5 Resultados

Por último, como etapa final de la metodología utilizada, en la parte de resultados se obtendrá el producto en su versión terminada para su posterior fabricación y las conclusiones correspondientes al comportamiento final del sistema.

2 Panorama general

En la actualidad, en algunas áreas de la robótica, la meta es crear sistemas autónomos que puedan igualar o superar el intelecto humano. Se parte de la premisa de que, algún día, una máquina podrá llegar a tener sentido común como lo tenemos los humanos.

El desarrollo de la robótica ocurre principalmente en los países desarrollados, que cuentan con programas de investigación gubernamentales y con inversión privada que fomentan y estimulan este campo de investigación.

En México existen relativamente pocas empresas que realizan investigación en el área de la robótica, por lo que son las instituciones educativas las que asumen esta responsabilidad. En las universidades se tienen recursos, sin embargo, no siempre son suficientes. Al no tener una industria importante en el área, los mecanismos y herramientas requeridos deben adquirirse, muchas veces, en otros países.

Una manera en la que se ha ido introduciendo el área de Robótica en México es a través de competencias como las que se tienen en países desarrollados. La Federación Mexicana de Robótica y la Asociación Mexicana de Robótica son instituciones nacionales dedicadas a promover el avance de esta disciplina. En consecuencia, ha sido posible obtener lugares para competencias en la *RoboCup* (torneo de robótica a nivel mundial) que se lleva a cabo anualmente. En México se tuvo el placer de ser sede en el año 2012.

Figura 2.1. México, sede de *RoboCup* 2012 [3]

RoboCup es una competencia de robótica a nivel internacional en la que participan más de 50 países y se realiza para incentivar el avance, investigación e innovación en el área de la robótica.

El objetivo principal de *RoboCup* es que en el año 2050 se pueda crear un equipo de futbol soccer, de robots completamente autónomos, capaz de ganar ante el equipo campeón de la copa mundial de futbol de ese año. Sin embargo, se ha ido expandiendo hacia otras áreas,

por ejemplo, existen competencias en el área de robots de rescate o de robots de servicio en el hogar. [4]

Figura 2.2. Competencia de *RoboCup Soccer Standard Platform* [5]

La principal competencia de Robótica que existe a nivel nacional, en México, es el Torneo Mexicano de Robótica (TMR), que es anual y de éste se obtienen los lugares para participar a nivel internacional en la RoboCup. Las competencias principales que hay en el TMR son básicamente algunas de las que hay en RoboCup.

Todas estas competencias tienen algo en común: Los robots deben ser autónomos y deben ser móviles para poder realizar sus tareas.

2.1 Robots Móviles

En principio,es necesario definir el significado de robot. Su origen viene del vocablo eslavo *robota*, que se refiere al trabajo forzado y se utilizó por primera vez en el año de 1920 por el escritor checo Karel Capek, en su obra de teatro titulada *Rossum's Universal Robots*. En ella se plasma a los robots no necesariamente como máquinas electromecánicas hechas por el ser humano, pero sí como entes biológicos que los sirven.

El concepto de lo que hoy se denomina "*robot"* ya existía, aunque no así el término como tal. Hasta ese entonces, esos "entes" capaces de realizar tareas para el ser humano se consideraban autómatas, definidos así porque tenían movimiento propio.

La mayoría de los autómatas que se fabricaban en la antigüedad eran mecánicos. Se creaban con la intención de realizar y facilitar labores cotidianas o tareas repetitivas. Algunos otros autómatas se elaboraban con la intención de entretener, principalmente a sus dueños, y que representaban figuras simbólicas de diferentes culturas, así fue hasta más o menos el siglo XVII. Los primeros autómatas se remontan aproximadamente al año 1500 a.C., cuando en la antigua Etiopía se construyó una estatua de su rey, Memon, que emitía sonidos cuando los rayos del sol pasaban a través de ella. Un autómata un poco más

conocido es aquel que fue elaborado por Leonardo Da Vinci, denominado "El León Mecánico", hecho para el rey Luis XII de Francia y cuya acción era mostrar el escudo de armas del rey a través de su pecho.

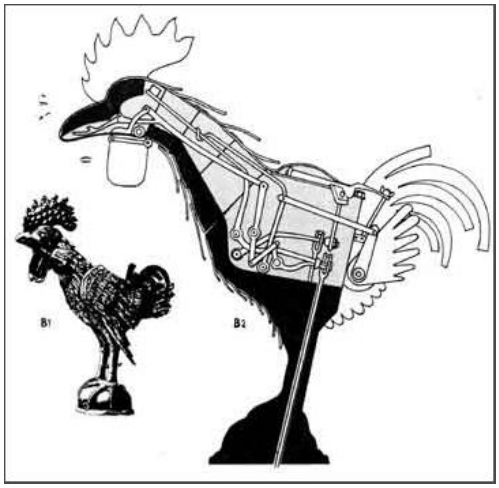


Figura 2.3. Autómata: Gallo de Estrasburgo [6]

A partir de finales del siglo XVIII y principios del XIX, los autómatas comenzaban a tener un enfoque más industrial. Fue así que se comenzaron a fabricar autómatas que pudieran sustituir actividades pesadas para el ser humano, sobre todo en la industria textil, ejemplos de ello fueron hiladoras o telares para confeccionar prendas. Tiempo después, iniciaría su incorporación a las industrias mineras y metalúrgicas en procesos de automatización.

Un gran acontecimiento para la evolución de los autómatas mecánicos y, a la postre, para el desarrollo de la robótica (una vez que se adoptó el concepto) fue el descubrimiento de la electricidad y su relación con el magnetismo, esto último por el uso de embobinados para motores (herramientas muy útiles para generar fuerzas mecánicas a partir de fuerzas electromagnéticas). [6]

La evolución de los autómatas dio lugar a la robótica. En nuestros días, los robots tienen diferentes, y muy variadas, aplicaciones; desde la industria para robots manipuladores, hasta robots de exploración o para la investigación. Asimismo, los robots móviles se consideran dentro de la clasificación de robots.

Los robots manipuladores son robots "estáticos", por decirlo de cierta manera, que se utilizan regularmente en la industria y que la mayoría simulan ser extremidades del ser humano para realizar tareas regulares y de precisión.

Ahora bien, un robot móvil se trata de un dispositivo que tiene la capacidad de cambiar de ubicación y consta de componentes, tanto físicos como computacionales, los cuales se pueden dividir en 4 subsistemas: Locomoción, Percepción, Razonamiento y Comunicación.

El tipo más común de robots móviles es el llamado AGV (*Automated Guided Vehicle*, o Vehículo Guiado Automatizado).

Un AGV es un sistema robótico que está guiado por líneas de cables de inducción (regularmente) y se utiliza para transportar objetos de un sitio a otro a través de rutas fijas. Pero un AGV es inflexible en cuanto al movimiento se refiere ya que si existe algún cambio en la ruta perjudicará el sistema y si existe algún objeto que se interponga en el camino del AGV producirá fallos en el mecanismo. Es por esta razón que se prefiere el manejo de robots autónomos. [7]

a) b)

Figura 2.4. Ejemplos de Vehículos Guiados Automatizados. a) Guiado por rieles [8] b) Seguidor de línea [9]

La autonomía de un robot se refiere específicamente a que no puede ser intervenido por algún control exterior a él y que posee "autogobierno". Típicamente, se le llama autónomo cuando no existe interacción con el ser humano para poder realizar su objetivo. Sin embargo, es evidente que requiere intervención humana para construirse y programarse.

En el caso de un robot móvil autónomo, éste tendrá la capacidad de moverse a través de su entorno pudiendo aprender de él con la ayuda de sensores. Tendrá la habilidad de moverse para desempeñar sus tareas y debe ser capaz de tomar decisiones de navegación a partir del reconocimiento de su entorno, razonando los posibles caminos que puede seguir sin que algo o alguien intervengan en su objetivo.

2.1.1 Aplicaciones

Las aplicaciones más comunes para este tipo de robots, aprovechando sus capacidades autónomas y de aprendizaje del entorno, son de exploración, transporte, vigilancia, dirección (guías), inspección, entre otras. En particular, los robots móviles son usados para aplicaciones en las que el entorno es inaccesible u hostil para los humanos, tales como robots submarinos, entornos contaminados, plantas nucleares o investigaciones espaciales (como el caso del *Curiosity* en la investigación en Marte).

Otras aplicaciones de los robots móviles se presentan en los campos de la Inteligencia Artificial, la Ciencia cognitiva y la Psicología; pues permiten la investigación en comportamientos inteligentes, percepción y cognición. El control programado de un robot móvil autónomo puede analizarse y, entonces, realizar experimentos controlados cuidadosamente para obtener resultados esperados y controlados. Por ejemplo, el sistema de navegación de las hormigas, que puede ser modelado en un robot móvil autónomo. [7]

2.2 Robot MicroMouse

La competencia de robots MicroMouse es una competencia de sistemas autónomos de un tamaño determinado que deben ser capaces de resolver un laberinto de muros de ciertas dimensiones y con ciertas reglas. Prácticamente, se trata de simular el comportamiento que tendría un ratón para llegar a un objetivo colocado en algún lugar del laberinto. Generalmente, el objetivo está colocado en el centro del laberinto.

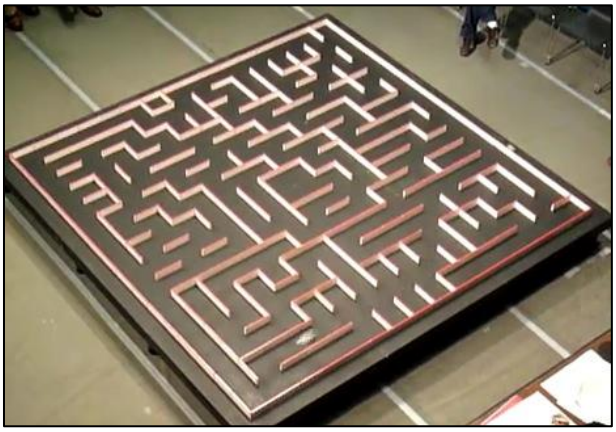


Figura 2.5. Laberinto de robot MicroMouse 16x16 [10]

Esta competencia nació a finales de la década de los 70's y fue creada por el *IEEE (Institute of Electrical and Electronics Engineers)* [11]

Las reglas principales de esta competencia son las siguientes:

- **Del robot**
  - Debe ser un sistema autónomo (sin control remoto) y su fuente de energía no debe de ser de combustión.
  - No debe dejar ninguna parte de sus componentes dentro del laberinto mientras está avanzando por el laberinto mismo.
  - No puede volar, saltar, cortar, escalar, quemar, marcar, dañar o destruir los muros del laberinto.
  - No debe ser mayor a 25 centímetros tanto de largo, como de ancho. No hay restricción en la altura.

  - En algunas zonas en las que se realiza la competencia, la construcción del robot está limitada a un presupuesto (comúnmente, no más de USD $500).
- **Del laberinto**
  - El laberinto está compuesto por unidades cuadradas múltiplos de 18 cm. x 18 cm.
  - El laberinto comprende 16 x 16 unidades cuadradas. Los muros del laberinto tienen 5 cm. de alto y 1.2 cm. de grosor.
  - Las caras de las paredes son blancas, la parte superior de color rojo, y el piso color negro. Aunque en la competencia puede haber variantes.
  - El inicio del laberinto se localiza en una de cualquiera de las 4 esquinas. El cuadro inicial estará constituido de 3 lados cubiertos, de tal manera que la orientación del primer cuadro tendrá la cara del "norte" descubierta, mientras que la del "sur" y la del "oeste" serán parte del muro exterior (el que cubre en su totalidad el laberinto). Habrá una "línea de inicio" ubicada entre el primer cuadro y el segundo, una vez que el robot ha cruzado la línea es cuando se comienza a contar el tiempo de la corrida.
  - El objetivo del laberinto está compuesto de 4 cuadros ubicados en el centro. Aunque son 4 cuadros, sólo habrá una entrada a la zona del cuarteto de celdas.
  - Puede haber diferentes caminos para llegar al objetivo del laberinto, sin embargo, los laberintos estarán diseñados para que un seguidor de muros (*wall-follower*) no sea capaz de encontrar el objetivo.

Cabe mencionar que dentro de cada corrida, el robot no debe tocar los muros, pues puede obtener alguna penalización. [12]

2.3 Agentes Inteligentes

Un concepto importante en este trabajo es el de "agente inteligente". Según Russell y Norvig en [13], un agente es un ente que puede percibir su entorno a través de sensores y funciona sobre ese mismo entorno por medio de actuadores de manera autónoma. Los agentes autónomos desempeñan tareas en ambientes dinámicos en tiempo real [14]. Un agente se considera *inteligente* cuando éste puede aprender de su entorno y tomar decisiones "prudentes".

Análogamente, y con la definición anterior, los seres humanos somos agentes, pues poseemos una serie de sensores de nuestro entorno (vista, oído, olfato, tacto y gusto); y dependiendo de lo que percibimos podemos realizar ciertas acciones con actuadores (manos, pies, cabeza, boca, ojos, etc.).

Un agente inteligente no solamente se refiere a un ente físico (hardware), sino también puede ser a un ente lógico (software).

En un robot móvil, los sensores pueden resultar de diferentes tipos, como los infrarrojos, de proximidad, de presión, de temperatura, de movimiento, etc. Los actuadores son aquellos que permiten realizar la acción al robot, los cuales en su mayoría suelen ser motores, tanto de diferentes tipos como de diferentes tamaños y diseños, que hacen posible la movilidad del robot a través de su entorno.

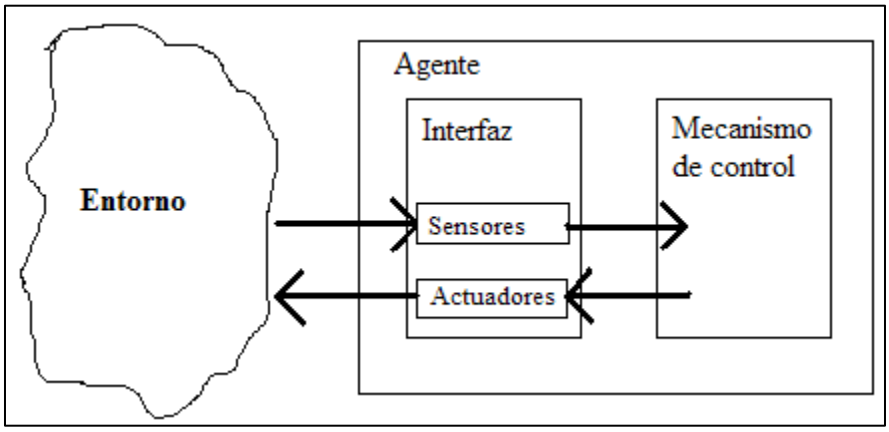


Figura 2.6. Estructura genérica de un agente [15]

Algunas características de un agente, listadas por Robert Schalkoff en [15], son las siguientes:

- **Tiene interfaces y límites a problemas bien definidos:** Lo que quiere decir que un agente podrá resolver un problema siempre que tenga solución finita.
- **Es embebido en un entorno particular:** Un agente estará contenido en un entorno, en el cual se dedicará a resolver un problema específico.
- **Está diseñado para lograr objetivos específicos:** Los cuales deben estar al alcance del agente y dichos objetivos acorde al entorno.
- **Es autónomo:** Actúa por sí mismo.
- **Es flexible, reactivo:** Un agente es reactivo cuando es flexible a cambios dentro de su entorno, dicho de otra manera, el agente es capaz de actuar ante cambios a su alrededor sin que esto afecte en su comportamiento y en su objetivo.
- **Despliega su comportamiento a la solución del problema:** El agente realizará acciones determinadas, de acuerdo a su comportamiento, para solucionar el problema relacionado a su objetivo.
- **Es proactivo:** Se encarga de cumplir sus objetivos, tiene iniciativa.

De acuerdo a Russell y Norvig, percepción se refiere a "*las entradas percibidas por el agente en cualquier momento dado*". Asimismo, "*una secuencia de percepciones es un historial completo de todo lo que el agente ha percibido y la elección de acciones de un agente en un momento dado puede depender de la secuencia de percepciones observadas hasta dicho momento*". Es así que, como lo exponen Russell y Norvig, se puede entender el

comportamiento de un agente, siempre y cuando, se pueda determinar la secuencia de acciones que realice con respecto a una secuencia de percepciones. "*Matemáticamente hablando, se dice que el comportamiento del agente está descrito en la función del agente que traza cualquier secuencia de percepciones dada a una acción*". [13]

Un agente de aprendizaje es el que puede adquirir y mantener conocimiento de su entorno por sí mismo [16]. Éste consiste de un módulo de aprendizaje en el que se procesan los datos adquiridos y crea o actualiza la información que se tiene en el también llamado módulo de conocimiento, que es el que almacena la información percibida del entorno [16]. El agente aprende del entorno por sí mismo o con ayuda de otros agentes, esto último en un Sistema Multiagente (SMA) del cual se hablará más adelante.

Un agente racional es un agente más sofisticado, por así decirlo. Es aquel capaz de realizar lo correcto. Pero cabe definir qué es "hacer lo correcto". Para un agente racional, la acción correcta es aquella que resulta de que el agente se aproxime cada vez más a su objetivo. Para medir el éxito de un agente, conocer la descripción de su entorno y, además, sus sensores y actuadores, proporcionarán una especificación completa de la tarea que debe realizar. Por lo tanto, se puede definir con mayor precisión la racionalidad de un agente. En resumen, un agente racional será el que realice una acción determinada por la función que lo defina y que le permita acercarse cada vez más a su objetivo, de acuerdo a su entorno, a sus sensores y actuadores. [13]

2.4 Programación Orientada a Agentes (POA)

El término de Programación Orientada a Agentes (POA) es un paradigma de programación relativamente nuevo que se introdujo por primera vez en el año 1993 por Yoav Shoham y que él mismo lo define como una variante del paradigma de Programación Orientada a Objetos (POO).

La POA está basada en el uso de agentes y los cuales tienen gran interacción con su entorno. El objetivo de este paradigma es la implementación de la programación sobre un agente que pueda percibir su entorno y que sea capaz de aprender del mismo por medio de acciones posibles a realizar. [17]

A continuación, se muestra una tabla comparativa entre los paradigmas de programación orientados a objetos y a agentes.

La información que percibe un agente siempre deberá ser consistente con el entorno en el que se encuentra.

POO vs POA

	POO	**POA**
Unidad Básica	Objeto	Agente
Parámetros que definen el estado de la unidad básica	Sin restricciones	Creencias, compromisos, capacidades, elecciones, etc.
Proceso de computación	Paso de mensajes y métodos de respuesta	Paso de mensajes y métodos de respuesta
Tipos de mensajes	Sin restricciones	Informe, solicitud, oferta, promesa, declinación, etc.
Restricciones en métodos	Ninguna	Honestidad, consistencia, etc.

Tabla 2-1. Comparación entre Programación Orientada a Objetos y Programación Orientada a Agentes [18]

Una diferencia principal entre la programación orientada a objetos y la orientada a agentes es que un objeto es un componente pasivo, que contendrá una serie de atributos y métodos o comportamientos que se caracterizan por ser pasivos (no cambian) y dependerán de una estimulación externa. En el caso de los agentes, éstos dependerán de sí mismos para realizar cualquier acción, además de ser capaces de manejar entornos continuamente cambiantes (componentes reactivos).

Las primordiales aplicaciones de este paradigma de programación orientado a agentes se encuentran en negocios electrónicos, ERPs (Planeación de Recursos de Empresas), sistemas de control de flujo áereo, PDAs (Asistentes Personales Digitales), ruteo de redes de computadoras, bases de datos, entre otras más.

Una arquitectura para agentes autónomos que interactúan entre ellos debe soportar programación dirigida a metas u objetivos, puesto que son la base para el comportamiento de un agente. [14]

La idea de la programación orientada a agentes propuesta por Shoham admite que un agente debe estar compuesto de 3 elementos básicos: creencias, capacidades y decisiones.

Las creencias de un agente serán aquellos conocimientos que tenga acerca del entorno, su manera de actuar ante él y todo lo relacionado a sus experiencias pasadas y presentes.

Las capacidades del agente son las acciones que le son posibles efectuar.

Las decisiones de un agente están ligadas a las creencias y a las capacidades del mismo, pues la decisión de un agente estará restringida a lo que el agente conozca del entorno y a lo que sea capaz de realizar.

2.5 Sistemas Multiagente (SMA)

Un sistema multiagentes es un sistema que contiene un conjunto de agentes interactuando entre ellos con el fin de lograr un objetivo (o varios) en común. Los sistemas multiagentes están basados en la programación concurrente, donde varios procesos interactúan para poder alcanzar objetivos de manera más rápida. Un sistema multiagentes podría observarse como un sistema de programación concurrente en el que los agentes interactúan entre ellos a través de comunicación y sincronización. Cada agente es autónomo, pero existirán situaciones en las que dependerán de las acciones de otro agente. [17]

Al igual que ocurren problemas de bloqueo entre procesos concurrentes, es posible que en sistemas multiagentes ocurran estos problemas, en los que un agente requiera manejar algún recurso del entorno que esté ocupado por otro agente. Para corregirlo, se utilizan técnicas muy parecidas a las de programación concurrente como el monitoreo de los recursos o el uso de semáforos. [17]

Los elementos que componen un SMA son:

- **Entorno:** Es el espacio en el que se encuentran los objetos y/o agentes. Se percibe y se actúa sobre él.
- **Objetos:** Es el conjunto de entidades pasivas que se encuentran en el entorno, y que pueden ser afectados o modificados por un agente.
- **Agentes:** Es el conjunto de entidades activas en el entorno, pertenecen al conjunto de objetos.
- **Relaciones entre objetos y/o agentes:** Son las relaciones que ligan a los objetos con los agentes.
- **Operaciones:** Son las acciones realizadas por los agentes para percibir y manipular su entorno.

Un SMA es una sociedad de agentes, ya que los agentes tienen la capacidad de socializar con otros. La habilidad de socializar entre agentes permite la existencia de interacción y cooperación. Estas relaciones se pueden considerar comportamientos resultantes del agrupamiento de agentes que deben actuar juntos para lograr sus objetivos. Una característica importante en un SMA es la cooperación entre agentes, la cual se puede hacer presente cuando se ingresa un nuevo agente al entorno. Dicha cooperación está compuesta de 3 elementos fundamentales, estos elementos son la coordinación, la colaboración y la resolución de conflictos. [17]

La coordinación está referida a la sincronización entre agentes para lograr la realización de sus tareas, con el objetivo de que funcionen correctamente. La colaboración consiste en hacer la asignación de tareas pequeñas a cada uno de los agentes, con la intención de cumplir un objetivo en común en el sistema. Para esto, se consideran técnicas de asignación

de tareas que distribuyan las acciones de manera adecuada. Los conflictos se producen como consecuencia de dos situaciones, cuando un agente pretende cumplir sus objetivos a costa de los demás agentes y cuando existe una lucha por los recursos entre los mismos agentes. [17]

2.6 Metodologías de Ingeniería de Software Orientado a Agentes (AOSE)

Como existen metodologías para el desarrollo de software orientado a objetos, también existen metodologías aplicadas para el desarrollo de sistemas orientados a agentes. Algunas de las metodologías existentes son:

- *Tropos*
- *Prometheus*
- AUML (*Agent Unified Modeling Language*)
- MESSAGE (*Methodology for Engineering Systems of Software AGEnts*)
- *Gaia*
- SODA (*Societies in Open and Distributed Agent spaces*)
- ROADMAP: Variante de Gaia que implementa el análisis de requerimientos más detallado utilizando casos de uso e implementa elementos para sostener ambientes de sistemas abiertos.
- MaSE (*Multi-agent System Engineering*)
- *Adelfe*
- *MAS-CommonKADS*
- PASSI (*Process of Agents Societies Specification and Implementation*)
- AOR
- RAP
- INGENIAS
- *Nemo*
- MASSIVE
- *Cassiopeia*
- CAMLE
- *Zeus*

Ciertas metodologías orientadas a agentes surgen de otras orientadas a objetos. Estas variantes utilizan conceptos similares tanto para objetos como para agentes.

Las metodologías AOSE se basan en un punto de vista externo, ya que para que un agente pudiera realizar una acción, deberá de reconocer un acontecimiento externo, es decir, que no está dentro del sistema para ejecutar dicha acción.

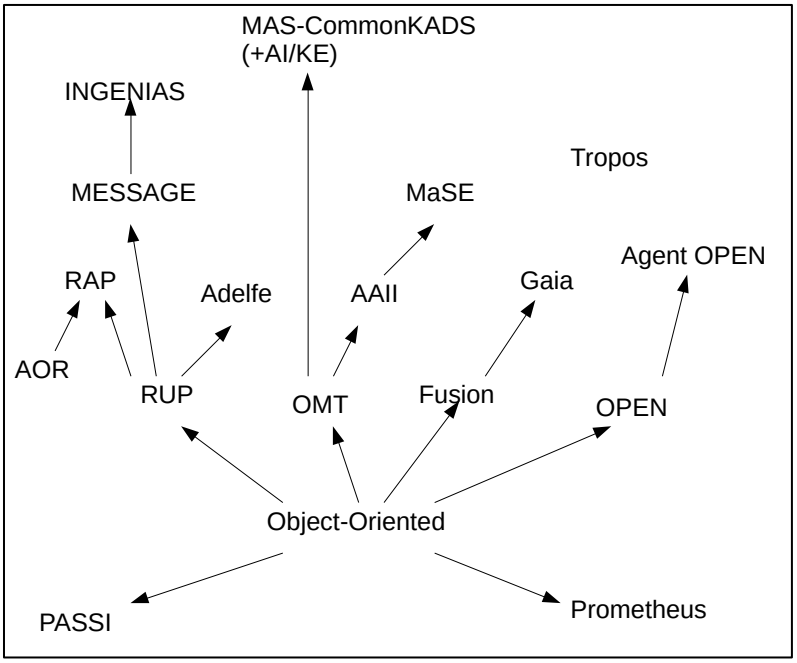


Figura 2.7. Influencia directa e indirecta de Metodologías Orientadas a Objetos sobre Metodologías Orientadas a Agentes [19]

Un agente se puede considerar un objeto complejo, que realiza funciones de manera externa. [20]

El uso de una metodología AOSE que fue adaptada de alguna metodología OOSE (Ingeniería de Software Orientado a Objetos) permite, en caso de que se esté familiarizado con estas últimas, el uso y el mejor entendimiento del modelado de la metodología AOSE, además, cualquier concepto utilizado en OOSE puede ser bien adaptado a la AOSE.

A continuación, se explicarán algunas metodologías orientadas a agentes.

2.6.1 Tropos

Tropos es una metodología que se basa en el *framework i**, que explica los conceptos actores, metas, tareas y dependencia. Esta metodología se caracteriza por su etapa de análisis de requerimientos en la que se tratan de detallar de manera explícita las necesidades del sistema, además de la forma en la que pueden interactuar los diferentes actores dentro del mismo.

Tropos está dividida en 4 etapas [19]:

- **Requerimientos tempranos:** Es el entendimiento del problema analizando su contexto y organización dentro del sistema.
- **Requerimientos tardíos:** Se definen los requerimientos funcionales y no funcionales del sistema.

- **Diseño arquitectónico:** Se define la arquitectura global del sistema en términos de subsistemas, con sus debidas dependencias y relaciones de datos y control.
- **Diseño detallado:** Explica el funcionamiento y las relaciones entre cada componente de la arquitectura propuesta en el diseño arquitectónico.
- **Implementación:** Esta etapa no está bien definida dentro de la metodología, aunque es necesaria para el desarrollo.

Las últimas 4 fases ya están bien establecidas en documentación de Ingeniería de Software y se utilizan en la mayoría de las metodologías, mientras que la primera fase es aceptada por la comunidad de investigación de este rubro, pero no se practica a menudo. [21]

La metodología *Tropos* está basada en agentes BDI (*Beliefs-Desires-Intentions*, o creencias, deseos e intenciones). Las creencias son las cosas que el agente sabe del sistema en general, principalmente del entorno. Los deseos se basan en el objetivo, u objetivos, del agente y permiten que tome ciertas decisiones siempre que éstas ayuden a satisfacer o a cumplir sus metas. Las intenciones representan las decisiones tomadas por el agente de acuerdo a su objetivo, involucra también la consecución de planes.

Existen varias herramientas de software para la creación de modelos de *Tropos*, una de ellas es llamada TAOM4E (*Tool for Agent-Oriented Modeling For Eclipse*)[4]. TAOM4E es un *plug-in* diseñado para el entorno de desarrollo (IDE) *Eclipse* y es de mucha ayuda para diseñar modelos de actores con sus respectivos planes y metas.

2.6.2 Prometheus

Prometheus es una metodología utilizada para el diseño de agentes y sistemas multiagentes. Es una de las metodologías que están diseñadas para su aplicación en software industrial, puesto que la mayoría están solamente diseñadas para su implementación en el campo de la investigación. [19]

Prometheus se basa, al igual que *Tropos*, en agentes BDI.

Consiste de 3 fases:

- **Especificación del sistema:** En esta etapa se define el sistema y sus funcionalidades, haciendo uso de las metas y los escenarios en los que se puede presentar el agente. La interfaz del sistema se describe a través de las acciones, percepciones y datos externos al agente.
- **Diseño arquitectónico:** Aquí se identifican todos los agentes participantes en el sistema general. La estructura completa del sistema se define en un diagrama

[4] *http://selab.fbk.eu/taom/*

previo. También se definen los protocolos de interacción para los agentes del sistema.

- **Diseño detallado:** Se definen las acciones de cada agente, en específico en función de sus capacidades, datos, eventos y planes. Se utilizan diagramas de procesos para detallar la relación entre los protocolos de interacción y los planes.

Prometheus cuenta con algunas herramientas de soporte para el diseño y modelado de la metodología. *Prometheus Design Tool* (PDT)[5] y *JADE Development Environment* (JDE).

Un detalle importante de esta metodología es que no está basada en notaciones de UML, lo cual puede ser una ventaja y una desventaja al mismo tiempo. Como ventaja, al ser una metodología que no depende de otra centrada a objetos, la orientación hacia los agentes puede ser mucho más fácil comenzando desde los conceptos básicos de agentes. Como desventaja, UML es un lenguaje estándar y muchos desarrolladores están familiarizados con él, así que para un desarrollador podría ser complicado comprender una nueva metodología desde cero. [19]

2.6.3 MAS-CommonKADS

Es otra metodología AOSE y se especializa en el proceso de análisis y diseño de sistemas multiagentes. Está basada en *CommonKADS*[6], que se considera como metodología para la ingeniería del conocimiento. *CommonKADS* implementa notaciones de UML. [22]

Consta de varias fases [19]:

- **Conceptualización:** Es en donde se delimita el sistema multiagentes y se identifican a todos aquellos agentes que estarán dentro de él, así como sus propiedades.
- **Análisis:** En esta fase se desarrollan diferentes modelos con el fin de identificar y analizar el sistema desde diferentes puntos de vista.
- **Diseño:** Es en la que se enfocan los modelos de la fase de análisis a un entorno operacional.
- **Desarrollo y pruebas:** Se hace la codificación y las pruebas pertinentes de los agentes definidos en las fases previas.

Esta metodología permite realizar una combinación con otras metodologías, incluyendo metodologías orientadas a objetos.

[5] *http://www.cs.rmit.edu.au/agents/pdt/*
[6] *http://www.commonkads.uva.nl/*

2.6.4 PASSI (*Process of Agents Societies Specification and Implementation*)

PASSI es una metodología para el diseño y desarrollo de SMA, que integra el diseño de modelos y conceptos de metodologías orientadas a objetos e inteligencia artificial, todo ello por medio de la notación de UML como referencia. Esta metodología trata de modelar un sistema multiagente como elementos autónomos con funcionalidades inteligentes. [19]

PASSI ha surgido de una serie de estudios y experimentos para el desarrollo de aplicaciones de robótica embebida [19]. Incluso, existe una herramienta de soporte para su diseño y modelado, que es un complemento (*add-in*) de *Rational Rose*™ [7], la cual lleva por nombre *PASSI ToolKit* (PTK)[8]. Genera el esqueleto de los agentes, cómo estarán estructurados; además de que es posible revisar las relaciones de los diagramas siguiendo el modelado paso por paso.

Las etapas de la metodología PASSI son llamadas modelos, basadas en los conceptos de SPEM (*Software Process Engineering Metamodel*). Así, cada modelo de PASSI estará dividido en fases. Los modelos y las fases de PASSI son las siguientes [19]:

a) **Modelo de Requerimientos del Sistema:** Modela los propósitos del sistema de agentes. Se compone de 4 fases:
    a. **Descripción de Requerimientos de Dominio:** Descripción funcional del sistema usando diagramas de casos de uso.
    b. **Identificación del Agente:** Atribuye las responsabilidades al agente. Se representa con UML.
    c. **Identificación de Role:** Se despliegan diagramas, analizando las responsabilidades de cada agente, dándole un role a cada uno.
    d. **Especificación de tareas:** Se especifican las capacidades de cada agente con diagramas de actividades.
b) **Modelo de Sociedad del Agente:** Modela las dependencias y las formas de interactuar de un agente para con los demás del sistema. Se compone de 3 fases:
    a. **Descripción de Ontología:** Describe los conocimientos que conciernen a cada agente y la forma en que se comunican con los demás. Utiliza diagramas de clases.
    b. **Descripción del Role:** Los diagramas anteriores permiten mostrar los roles de cada agente, de manera que permite obtener una mejor descripción de ellos, además de sus tareas, comunicaciones con los demás agentes y sus dependencias.

[7] Marca Registrada de IBM. Es una herramienta diseñada por dicha empresa para el manejo de aplicaciones desarrolladas en base a UML.
[8] PTK es una herramienta *open-source*

   c. **Descripción del Protocolo:** Detalla los protocolos utilizados para la comunicación de agentes.

c) **Modelo de Implementación del Agente:** Realiza la implementación de los modelos anteriores, en términos de clases y métodos. La abstracción se realiza hacia agentes y en dos niveles distintos, a nivel social (multiagente) y a nivel singular (un agente). Tiene dos fases:
   a. **Definición de la Estructura del Agente:** Describe la estructura de clases de un agente.
   b. **Descripción del Comportamiento del Agente:** Describe el comportamiento del agente, individualmente, con diagramas de actividad.
d) **Modelo de código:** En este modelo se genera el código de los modelos anteriores y se retoca (si se utilizó PTK).
e) **Modelo de Despliegue:** Muestra la distribución de las partes del sistema. Está compuesto de la Configuración de Despliegue, que describe con diagramas la localidad de los agentes.
f) **Pruebas:** Se divide en dos fases:
   a. **Prueba del Agente Singular:** Verifica el comportamiento de cada agente con respecto a los requerimientos originales del sistema.
   b. **Prueba de Sociedad:** Integra el funcionamiento de los sistemas individuales para su validación.

PASSI se adapta al proceso de desarrollo de software iterativo, esto quiere decir que se realiza varias veces una misma etapa con el fin de identificar detalles que no concuerden con los requerimientos del sistema, o cambios dentro de los mismos. PASSI cuenta con dos tipos de iteraciones dentro de la metodología, la primera se realiza en presencia de nuevos requerimientos y esto afecta a todos los modelos de PASSI; la segunda iteración corresponde al Modelo de Implementación del Agente, debido a que existen dependencias en cuanto al comportamiento y estructura entre el sistema de agente múltiple y el de agente singular. [19]

2.6.5 Gaia

Esta metodología aprovecha las estructuras organizacionales de un sistema multiagente para definir un proceso de diseño que sea coherente para el sistema. Esas estructuras organizacionales están definidas por reglas organizacionales, que a su vez están definidas por los roles y los protocolos de interacción entre agentes. Es decir, cada agente tendrá uno o varios roles establecidos dentro del sistema, pero para que las metas de los agentes sean cumplidas tendrá que haber interacción entre ellos. [19]

No obstante, algunas características de *Gaia* son las siguientes:

- No trata directamente con técnicas de modelado en particular.

- No trata directamente con cuestiones de implementación.
- No trata explícitamente con las actividades de captura y modelado de requerimientos. Ni siquiera con la etapa de requerimientos temprana.

Gaia se compone de 3 fases en particular:

- **Análisis:** En esta fase se obtienen las especificaciones para el sistema dentro de un modelado del entorno, modelados de roles e interacción, y el conjunto de reglas organizacionales. El objetivo primordial de esta fase es entender a la perfección el sistema multiagente en cuestión.
- **Diseño arquitectónico:** Una vez obtenidos los modelos básicos, el paso siguiente es realizar una estructura organizacional para identificar el sistema multiagente de manera eficiente.
- **Diseño detallado:** En esta fase del diseño se identifica el modelo del agente y el modelo de servicios. Por agente se entiende, en la metodología *Gaia*, a una entidad activa conformada por un conjunto de roles de agente; por lo tanto, el modelo de agente se encarga de identificar qué clases de agente tienen que ser definidas para ciertos roles y cuántas instancias de cada clase tienen que ser ejecutadas en el sistema. Por otro lado, el modelo de servicios identifica los servicios que estarán asociados con cada clase de agente, o puede ser también con cada uno de los roles desempeñados por las clases de agente.

Por último, la salida de la fase de diseño detallado se deberá considerar en la fase de Implementación, pero como esta fase no se encuentra definida dentro de la metodología *Gaia*, ésta se puede realizar utilizando un método tradicional. [19]

2.7 Comparación entre metodologías AOSE

A continuación, se hará una comparativa entre las metodologías AOSE analizadas en la sección anterior (*Sección 2.6*) que hará posible tomar una decisión de la metodología más conveniente para el desarrollo del trabajo.

La comparación se basa en las hechas en [19] y [23], en donde se analizan ciertas características de cada metodología.

Ciclo de vida de la metodología

Cada metodología tiene un ciclo de vida que permite dar seguimiento a la metodología misma. Además, sirve como referencia para el desarrollo del sistema, donde se especifican las diferentes acciones que se realizan por cada etapa del ciclo y su secuencia.

Metodología / Etapa	Tropos	Prometheus	MAS-CommonKADS	PASSI	Gaia
Análisis	Requerimientos tempranos	Especificación del Sistema	Conceptualización	Modelo de Requerimientos del Sistema	Sí, sin análisis de requerimientos
	Requerimientos tardíos		Análisis		
Diseño	Diseño arquitectónico	Diseño arquitectónico	Diseño	Modelo de Sociedad	Diseño arquitectónico
	Diseño detallado	Diseño detallado			Diseño detallado
Implementación	Sí	No	Desarrollo y pruebas	Modelo de Implementación de Agente	No

Tabla 2-2. Comparación de ciclos de vida.

Aunque en algunas metodologías no esté definida una etapa de implementación no significa que no se deba realizar. En las metodologías donde haya etapa de implementación se puede ejecutar adaptándola de otras metodologías, pues esta etapa es genérica en casi todos los sistemas.

Características relacionadas al proceso

	Tropos	Prometheus	MAS-CommonKADS	PASSI	Gaia
Desarrollo del ciclo de vida	Iterativo e incremental	Iterativo	Proceso cíclico conducido por riesgos	Iterativo (excepto en los modelos de código y despliegue)	Iterativo dentro de cada fase pero secuencial entre fases.
Perspectiva de desarrollo	*Top-Down*	*Bottom-up*	Híbrido	*Bottom-up*	*Top-Down*
Naturaleza del agente	Agente BDI	Agente BDI	Heterogéneo	Heterogéneo	Heterogéneo
Enfoque hacia el desarrollo de sistemas multiagente	-Basado en modelado i* -No orientado a roles	-Orientado a Objetos -No orientado a roles	-Ingeniería de Conocimiento -No orientado a roles	-Orientado a Objetos -Orientado a roles	-Orientado a Objetos -Orientado a roles (a organización)

Tabla 2-3. Características relacionadas a los procesos de cada metodología. [19]

En la *Tabla 2-3*, se hace un análisis de la manera en la que se desarrolla el ciclo de vida de cada una de las metodologías revisadas, además de su perspectiva, naturaleza y enfoque de desarrollo para multiagentes. El ciclo de vida en la mayoría de las metodologías es iterativo, pues se ha mostrado que este enfoque suele corregir detalles en cada etapa. La perspectiva de desarrollo es el punto de vista en el que se maneja la secuencia de cada metodología. La naturaleza del agente es la base para el desarrollo del sistema. Por último,

el enfoque hacia sistemas multiagentes está orientado a cómo se percibe una interacción de varios agentes entre ellos.

Modelado y herramientas CASE (*Computer-Aided Software Engineering*)

	Tropos	**Prometheus**	**MAS-CommonKADS**	**PASSI**	**Gaia**
CASE	TAOM4E	PDT y JDE	Sí	PTK	No
Lenguaje de modelado	UML/AUML	UML/AUML	UML	UML	Neutral
Modelado de Entorno	No	Sí	No	No	Sí
Modelado de Inteligencia	Sí	Sí	Sí	Sí	Sí
Modelado de Interacción	Sí	Sí	Sí	Sí	Sí

Tabla 2-4. Características de modelado y herramienta CASE.

La **herramienta CASE** se refiere a una herramienta utilizada para el diseño de Ingeniería de Software Asistido por Computadora, de ahí las siglas. En la *Tabla 2-4* se muestra la herramienta CASE que utiliza cada metodología comparada. **Lenguaje de modelado** es el lenguaje que utiliza como base para el diseño de modelos del sistema. Los **modelados de entorno, inteligencia e interacción** son parte de los modelos realizados dentro de algunas metodologías, ya que parten de dichos modelos para generar un SMA. En la tabla se especifica si el modelado es parte o no de la metodología.

Obtener información acerca de las metodologías de desarrollo de software orientado a agentes es complicado, pues en algunos casos las páginas web en donde se encontraban las definiciones de las metodologías ya no existen y, en otras, el acceso a la documentación está restringido a miembros de las comunidades de investigación. La documentación impresa existente de estas metodologías es limitada y/o no se ha difundido de manera extensa. Las principales fuentes se encontraron en documentos difundidos a través de la web y, especialmente, en sitios de instituciones académicas de varias partes del mundo.

3 Algoritmos

3.1 Algoritmia

El término *Algoritmia* es considerado como la ciencia que se encarga del estudio y análisis de los algoritmos, además de su caracterización e implementación en todos sus aspectos.

Cuando se habla de algoritmos, no necesariamente se está hablando de "computación". En pocas palabras, un algoritmo es una serie de pasos a seguir, bien definidos, para la resolución de alguna actividad. El ejemplo más común de ello es una receta de cocina, la receta es un algoritmo, es decir una serie de pasos bien definidos, para llegar a un objetivo, en este caso, un platillo.

Los algoritmos tienen algunas características o elementos que pueden enumerarse para su comprensión, especificados en [24]:

- **Debe ser finito**: Esto quiere decir que un algoritmo debe terminar después de algún número finito de pasos, que tenga un fin.
- **Bien definido:** Cada paso del algoritmo debe estar precisamente definido; esto es, no debe ser ambiguo y tendrá que estar rigurosamente especificado en todos los casos.
- **Entrada**: Un algoritmo puede tener cero o más entradas, es decir, algunas cantidades que estén inicializadas antes de que el algoritmo pueda comenzar a realizarse.
- **Salida**: Un algoritmo tendrá una o más salidas, las cuales estarán relacionadas a las entradas del mismo.
- **Efectividad**: Generalmente, un algoritmo se espera que pueda ser efectivo. Lo que implica que las operaciones que puedan ser realizadas en el algoritmo sean lo suficientemente básicas y se hagan en un tiempo finito.

3.2 Eficiencia y complejidad de un algoritmo

En cuestiones computacionales no solamente se debe buscar la solución de un problema, sino también de la mejor manera posible, optimizando recursos tanto de software y hardware; es decir, realizar un algoritmo que además de eficaz, sea eficiente.

La eficiencia y la complejidad son criterios que pueden ayudarnos a medir el desempeño de un algoritmo. Principalmente, están basados en la simplicidad y en usar los recursos de la computadora de una manera eficiente.

El uso eficiente de recursos se refiere a dos parámetros dentro la computadora: el tiempo, que será lo que tarde en ejecutarse el algoritmo; y el espacio, la memoria que necesite el algoritmo para lograr su objetivo.

El costo temporal se refiere al tiempo que emplea un algoritmo en ejecutarse, dados los datos de entrada del algoritmo. Depende de diversos factores como los datos de entrada suministrados, la calidad del código generado por el compilador, la naturaleza y la rapidez de las instrucciones máquina del procesador que ejecuta el programa y la complejidad intrínseca del algoritmo.

Se puede estudiar de dos maneras distintas:

- Medida teórica (*a priori*), que consiste en obtener una función que acote el tiempo de ejecución del algoritmo para valores de entrada dados.
- Medida real (*a posteriori*), que consiste en medir el tiempo de ejecución del algoritmo para valores de entrada dados y en una computadora en concreto.

Para obtener la medida *a posteriori* de un algoritmo en un código es necesario analizar todas las líneas e instrucciones del programa. Algunas de las líneas de código en todos los lenguajes de programación no tienen ningún costo en cuanto a tiempo, por ejemplo, la declaración de variables o los prototipos de funciones, la declaración de constantes simbólicas (en C con la directiva *#define*), comentarios. Sin embargo, cada línea que represente una expresión o una asignación tendrá como costo un paso (aunque el tiempo que utilice en ejecutarse dicho paso depende de las operaciones que se realizan).

Por lo regular, se utiliza medida *a priori*, ya que se busca generalizar la función del algoritmo. La manera de representar las funciones de los algoritmos utilizando esta medida se hace con ciertas notaciones denominadas asintóticas, las cuales se tratarán más adelante. [25]

El costo espacial hace referencia al uso de memoria que tiene un algoritmo. Este uso de memoria se da por las variables definidas dentro del algoritmo, tanto estáticas como dinámicas, y usadas por el mismo; al igual que como se hace al realizar llamadas a funciones. Hay dos tipos de estos gastos en memoria por parte de un algoritmo:

- El uso estático, que son las variables globales declaradas en el algoritmo y que permanecen desde que el programa es ejecutado hasta que éste termina su ejecución. Cabe mencionar que este dato es constante y se conoce desde el principio.
- El uso dinámico, que se efectúa mientras se hacen llamadas a funciones, ya que es en ese momento cuando se reserva la memoria para las variables locales que se utilizan dentro de la función, además de una localidad extra para el regreso de la

función principal. Esta sección de memoria crecerá cada vez más en caso de que se realicen funciones recursivas.

Generalmente, los costos se calculan en instrucciones críticas de un algoritmo. Estas instrucciones críticas suelen presentarse como ciclos iterativos. En el caso del lenguaje C (que es el que a nosotros nos concierne), los ciclos de iteración se presentan como ciclos *for*, *while* y *do-while*. [1]

En Computación, se suele confundir el término Desempeño hablando del uso de los recursos del sistema computacional con el término de Complejidad, cuando se habla de la eficiencia de un algoritmo. Que un algoritmo sea complejo no se refiere únicamente a que utiliza la mínima cantidad de memoria o requiere cada vez menos del CPU para poder completarse. La complejidad de los algoritmos radica también en la manera de procesar cierta cantidad de datos. Por ejemplo, se tiene un algoritmo que se encarga de ordenar una serie de 100 datos numéricos y lo hace en 10 milisegundos y, además, se tiene otro que procesa la misma cantidad de datos, aunque éste en 30 milisegundos. En un principio, la mejor solución debería ser el primer algoritmo. Pero ahora se procesan 1000 datos, el primer algoritmo tarda 100 milisegundos, mientras que el segundo lo hace en 50 milisegundos. Éste último tiene mayor eficiencia respecto al primero conforme aumenta la cantidad de datos a tratar. [26]

3.3 Notaciones asintóticas

Las notaciones asintóticas son utilizadas para analizar algoritmos que tienen cantidades de datos grandes, es decir, cuando $n \to \infty$, siendo n el número de datos a analizar. Esto significa que se examina el comportamiento del algoritmo para problemas grandes y el crecimiento de su complejidad conforme vaya aumentando la cantidad de datos.

Existen funciones matemáticas que ejercen patrones de comparación con los que nosotros como programadores solemos comparar la complejidad de un algoritmo. Los patrones antes mencionados se utilizan en las notaciones asintóticas para decir que un costo asintótico pertenece a cierto patrón. Estas funciones se enlistan a continuación (en orden ascendente de su tasa de crecimiento) [1]:

- La función $f(n) = 1$ es la más típica en algoritmos secuenciales y la más deseable, ya que mide únicamente las instrucciones que se ejecutan una sola vez hasta terminar el algoritmo. La ejecución de la secuencia de instrucciones no depende de los datos y siempre se ejecutará en un tiempo constante.
- La función $f(n) = \log n$ es típica para aquellos algoritmos que van resolviendo el algoritmo de manera que va reduciendo la cantidad de datos a procesar mientras avanza el problema. Un ejemplo de ello es la búsqueda binaria, debido a que en ésta, en cada iteración, se divide el conjunto de datos a procesar por la mitad.

- La función $f(n) = n$ es una función lineal. Típica para aquellos algoritmos en los que el tiempo que se le dedica a cada iteración será el mismo para cada elemento e irá creciendo proporcionalmente al número de elementos.
- La función $f(n) = n \log n$ es típica para los algoritmos que utilizan estrategia de *divide y vencerás*. Éstos se encargan de dividir un problema en partes iguales y al obtener los resultados independientes, se construirá un resultado total a partir de ellos. En general, esta función crece un poco más rápido que la función lineal.
- La función $f(n) = n^2$ indica que un algoritmo posee un costo cuadrático. Es típica para algoritmos de ordenamiento que se basan en comparaciones dos a dos. En esta función es donde comienzan las restricciones para problemas de gran cantidad de elementos.
- La función $f(n) = n^3$ se utiliza para costo cúbico. Tiene un crecimiento más acelerado que la función cuadrática. El uso de este tipo de algoritmos es de uso limitado. Regularmente, en los algoritmos de este tipo de complejidad, se encuentran ciclos 3 veces anidados.
- La función $f(n) = 2^n$ contiene un costo exponencial. El uso de este tipo de algoritmos prácticamente es nulo, ya que para una cantidad de elementos a procesar relativamente corta, la complejidad del algoritmo se vuelve inmensa. En esta categoría entran los algoritmos de fuerza bruta.
- Las funciones $f(n) = n!$ y $f(n) = n^n$ corresponden a algoritmos con complejidades mayores de problemas que se consideran computacionalmente inaceptables.

Todas las funciones anteriores suelen utilizarse como cotas para definir la complejidad de un algoritmo. Estas funciones representarán la base para decidir a qué tipo de función o tasa de crecimiento pertenece un algoritmo. [1]

3.3.1 Notación O (O grande)

Esta notación establece una cota superior de complejidad de un algoritmo. Dando dicha notación, decimos que el costo real del algoritmo está por debajo de la función utilizada como cota. Es decir:

$$f(x) \in O\big(g(x)\big)\ si\ \exists x_0, n_0\ \forall x > x_0, f(x) < n_0 g(x)$$

Generalmente la cota superior es ajustada. No es lo mismo decir que el costo de un algoritmo es $T(n) \in O(2^n)$, si se ajusta a $T(n) \in O(nlog(n))$. [1]

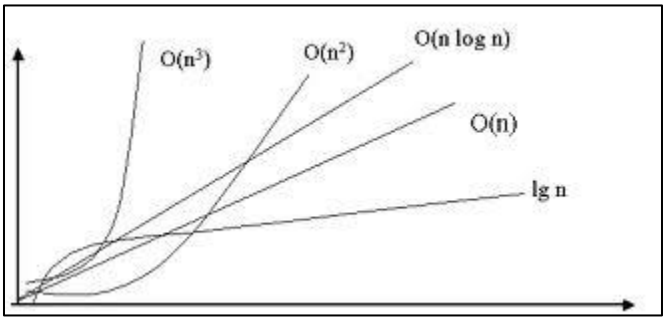


Figura 3.1. Tasas de crecimiento *O(f(x))* de algunas de las funciones patrón [27]

3.3.2 Notación Ω (Omega grande)

Se utiliza para expresar una cota inferior del costo real del algoritmo. Su expresión es la siguiente:

$$f(x) \in \Omega\big(g(x)\big)\ si\ \exists x_0, n_0\ \forall x > x_0, f(x) > n_0 g(x)$$

Expresa el costo mínimo que, al menos, empleará el algoritmo. Se utiliza para representar el mejor caso. [1]

3.3.3 Notación Θ (Theta)

Se utiliza para referir un costo de una cota ajustada. Una cota del mismo orden que el costo real del algoritmo, expresada de la siguiente manera:

$$f(x) \in \Theta\big(g(x)\big) \Leftrightarrow g(x) es\ cota\ inferior\ y\ superior\ de\ f(x)$$

Quiere decir que:

$$f(x) \in \Theta\big(g(x)\big) \Longrightarrow f(x) \in O\big(g(x)\big)\ y\ f(x) \in \Omega\big(g(x)\big)$$

$$\Longrightarrow f(x) \in O\big(g(x)\big) \cap \Omega\big(g(x)\big)$$

Expresada de otra manera:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R} - \{0\} \Longrightarrow f(x) \in O(g(x))$$

Existen varios algoritmos que resuelven problemas de búsqueda, como en nuestro caso es un laberinto de muros. Estos algoritmos de búsqueda, en su mayoría, se encargan de encontrar el camino más corto para llegar a la solución, pero cada uno lo hace de una manera distinta.

Algunos de estos algoritmos se explicarán más adelante.

3.4 Mejor caso, peor caso y caso promedio

Una manera de analizar un algoritmo es por medio del análisis de casos. El mejor caso es aquel que tiene la menor complejidad computacional cuando se ejecuta el algoritmo. El peor caso es, al contrario del anterior, el que tiene la mayor complejidad en cuanto a computación se refiere. Y el caso promedio es el caso que se encuentre entre los anteriores, es decir, que no sea el mejor, pero tampoco el peor. Un ejemplo de ello está en los algoritmos de ordenamiento, cuando se busca ordenar un conjunto finito de datos, el mejor caso para este ordenamiento será en el que los datos ingresados ya se encuentran ordenados (al ingresarse), el peor caso será cuando los datos, al ser ingresados, están en orden inverso al buscado, y el caso promedio será cualquier otro orden que tengan los datos.

En la práctica, es común que se deseche el mejor caso, ya que las probabilidades de que ocurra son muy pocas, más aún cuando la cantidad de datos a operar se va incrementando. Regularmente, se toma el peor caso como referencia para el análisis del algoritmo. [28]

3.5 Principio KISS

Para algunos autores, el principio KISS es una regla de diseño cuyo acrónimo del inglés significa *Keep It Stupid Simple* (mantenlo estúpidamente simple), o *Keep It Simple, Stupid!* (mantenlo simple, ¡estúpido!). Es un principio que ha sido clave para la elaboración de sistemas, sobre todo en software. Está basado en mantener una idea de desarrollo de software de manera simple y entendible para los usuarios y desarrolladores. De este modo, puede obtenerse un producto que sea fácil de entender y de modificar en caso de que se requiera por cualquier persona que deba hacerlo, sin necesidad de que tenga profundos conocimientos del área. [29]

Algunas ventajas de aplicar este principio son:

- Capacidad de solucionar más problemas de manera rápida.
- Producir código para problemas difíciles en menor cantidad de líneas.
- Producir código de calidad.
- Construir sistemas amplios con facilidad de mantenimiento.
- Construir código base más flexible y fácil de comprender para modificarlo, si es necesario.

3.6 Problemas bien definidos y soluciones

Un problema puede ser definido por 4 elementos:

- Un estado inicial en el que el agente comienza.
- Una descripción de las acciones posibles disponibles para el agente.

- La prueba del objetivo, la cual determina si un estado dado es un estado objetivo.
- Una función de costo de trayectoria, que asigna un costo numérico a cada trayectoria.

Una solución a un problema es una trayectoria del estado inicial al estado objetivo. La calidad de la solución está medida por la función del costo de la trayectoria y una solución óptima tendrá el menor costo de trayectoria entre todas las soluciones. [13]

3.7 Algoritmos de Búsqueda

Las estrategias de búsqueda para la solución de problemas se pueden dividir en dos grupos, las búsquedas desinformadas y las búsquedas informadas.

3.7.1 Estrategias de búsqueda desinformada

Una búsqueda desinformada, o también conocida como búsqueda ciega, es aquella que no tiene información adicional acerca de los estados además de la que es proporcionada en la definición del problema. Todo lo que puede hacer es generar estados sucesores y distinguir entre los estados objetivo de los que no lo son. [13]

3.7.1.1 BFS (Breadth-First Search)

Breadth-First Search (o Búsqueda por el Primero en Anchura) es una estrategia simple en la cual el nodo raíz es expandido primero en todos sus sucesores, entonces los sucesores del nodo raíz son expandidos en los siguientes sucesores, después sus sucesores, y así sucesivamente. En general, todos los nodos son expandidos a una profundidad dada en el árbol de búsqueda antes de que cualquier nodo del siguiente nivel sea expandido.

Del ejemplo de la *Figura 3.2*, el orden de revisión de los nodos es: A-B-C-D-E-F-G-H-I-J-K-L. Aunque M resultó de una expansión, es el único que no se revisa, ya que antes de él está L que es la solución.

BFS puede implementarse con un llamado árbol de búsqueda y una estructura de datos de tipo FIFO (*First-In-First-Out*), llamada cola. Esto para asegurar que los primeros nodos visitados son expandidos primero. Es decir, el primer dato en entrar a la cola será el nodo raíz, cuando se hace la expansión, se retira el nodo de la cola y se expande, cada nodo obtenido de la expansión es ingresado al final de la cola. Si hay elementos en la cola, se procede a retirar el primero que haya entrado y se realiza su expansión. De igual manera, los nodos expandidos serán ingresados al final de la cola, lo cual significa que los nodos poco profundos (los que entraron una expansión antes) serán expandidos antes que los nodos más profundos (los que están siendo expandidos en ese momento).

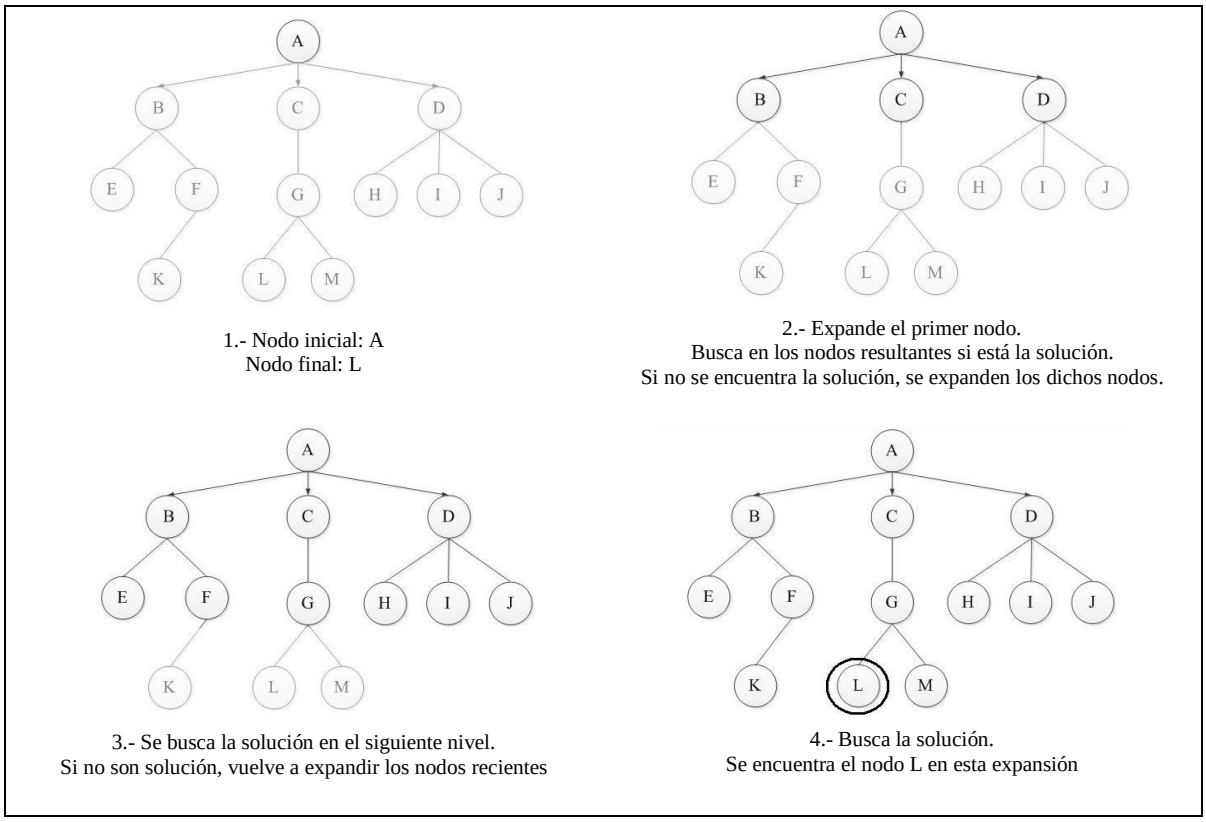


Figura 3.2. Ejemplo de BFS

3.7.1.2 DFS (Depth-First Search)

DFS (o Búsqueda por el Primero en Profundidad) siempre expande el nodo más profundo en el conjunto de nodos generados del árbol de búsqueda. Esta búsqueda procede de inmediato al nivel más profundo del árbol, donde los nodos no tienen sucesores. Como esos nodos están expandidos son tomados del conjunto de nodos generados, por lo que la búsqueda regresa al siguiente nodo menos profundo que aún tiene sucesores inexplorados.

En la *Figura 3.3* se puede observar un ejemplo de DFS, el orden en el que se revisan los nodos es el siguiente: A-B-C-D-E-F-K-G-L. De igual modo que en el ejemplo de la *Figura 3.2*, el nodo M resultó de la última expansión pero no se revisa.

Esta estrategia de búsqueda puede ser implementada por un árbol de búsqueda con una estructura de datos LIFO (*Last-In-First-Out*), conocida como una pila. Como alternativa al árbol de búsqueda, es común implementar DFS con una función recursiva que se llama a sí misma en cada expansión en turno.

Un caso especial de esta estrategia de búsqueda es la llamada Búsqueda por marcha atrás (*Backtracking*). En la búsqueda por marcha atrás sólo un sucesor es generado a la vez antes de cualquier otro sucesor, es decir, no se expande el nodo por completo en ese instante; cada nodo expandido parcialmente reconoce qué sucesor generar después.

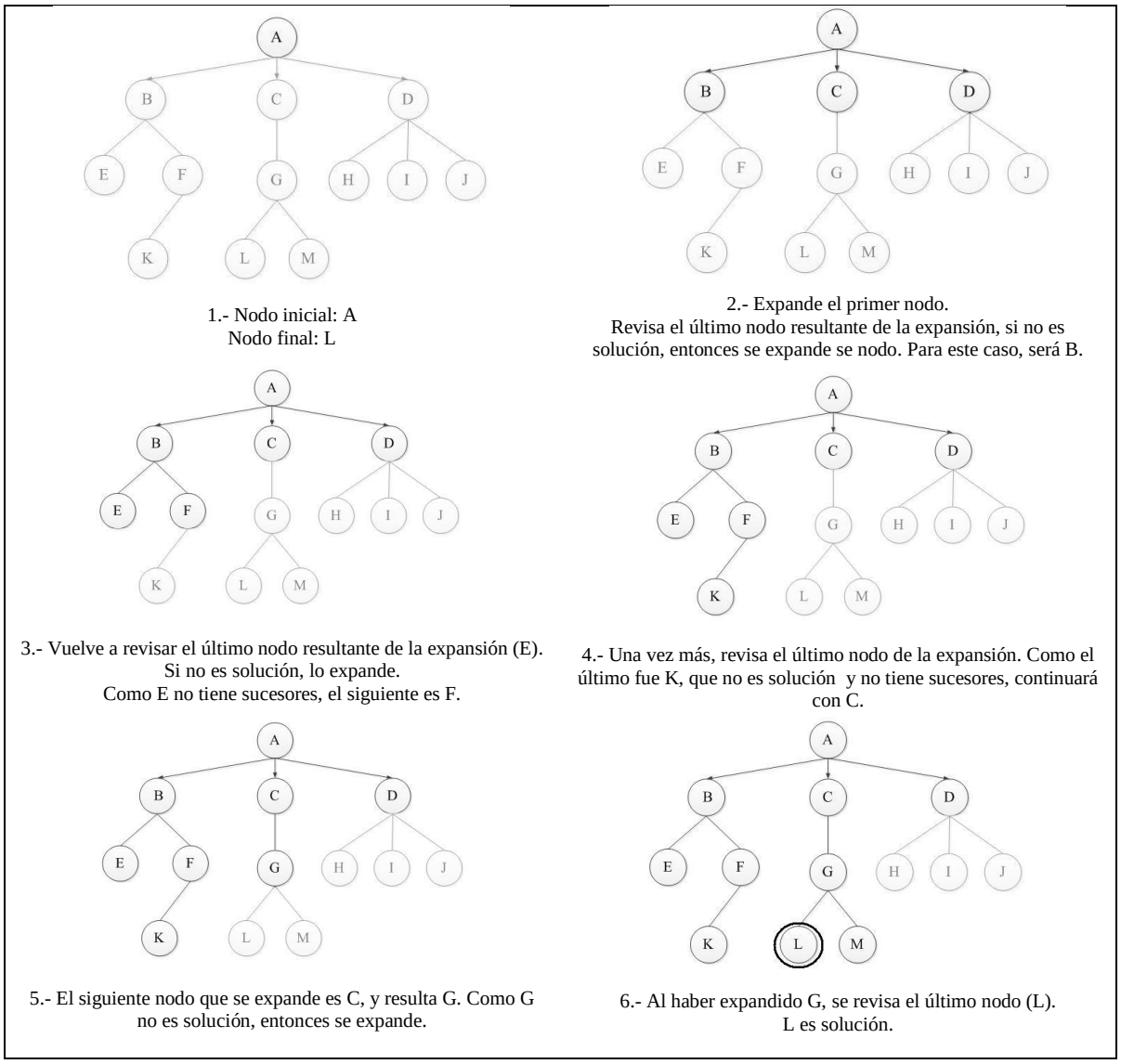


Figura 3.3. Ejemplo de DFS

3.7.1.3 Mano derecha

Conocida como la estrategia *Wall Follower* (seguidor de muros), es una estrategia sencilla utilizada generalmente para la solución de laberintos. Se puede considerar una implementación de *Backtracking* sobre problemas de laberinto. La búsqueda se hace realizando la expansión de un nodo en una sola dirección, en el caso de ser un laberinto tendrá como máximo 3 sucesores (derecha, enfrente e izquierda). Para mano derecha, el sucesor de la extrema derecha es expandido. En caso de que el nodo actual no tenga sucesor en su extrema derecha, se procede al más próximo que es el sucesor de enfrente. Cuando no se encuentra sucesor a la derecha o hacia el frente se realiza la expansión a la izquierda; y, por último, si no existe sucesor en ninguno de los 3 lados distintos, regresa al nodo anterior.

Del mismo ejemplo mostrado en las técnicas anteriores, el orden de revisión para esta estrategia es: A-D-J-I-H-C-G-M-L.

3.7.1.4 Mano izquierda

Es una variante de la estrategia *Wall Follower*, y en este caso la búsqueda se hace empezando por la extrema izquierda. La secuencia es similar, expande el nodo del lado izquierdo; si no tiene sucesor izquierdo, expandirá al sucesor de enfrente, y si no tiene los dos anteriores, procederá a expandir del lado derecho. Igualmente, si un nodo no se puede expandir, regresará al nodo anterior.

Del mismo ejemplo mostrado antes, para este caso, la revisión de los nodos sería la siguiente: A-B-E-F-K-C-G-L.

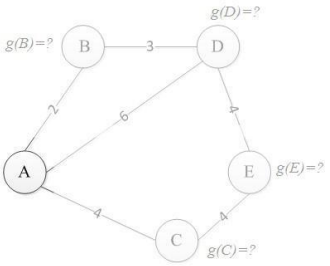
3.7.1.5 UCS (Uniform-Cost Search)

Para que BFS encuentre la solución óptima, con el menor costo, requiere que el costo de un nodo a alguno de sus adyacentes sea igual para todos los casos. Sin embargo, cuando varía el costo de nodo a nodo se utilizan otros algoritmos de búsqueda para localizar el objetivo [13]. Uno de estos algoritmos es el denominado UCS.

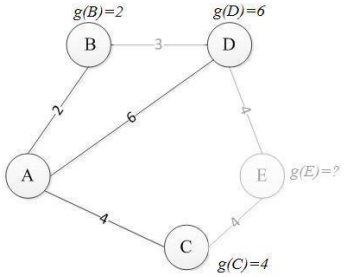
Uniform-Cost Search (o, Búsqueda por Costo Uniforme) es otra estrategia de búsqueda desinformada para la resolución de problemas. En ésta son calculados los costos del nodo origen a cada nodo hasta encontrar una solución. A este costo se le denomina *g(n)*. Con UCS es posible encontrar una solución óptima y completa.

La expansión de nodos se realiza con el costo *g(n)* menor hasta que el siguiente nodo a expandir es el nodo solución.

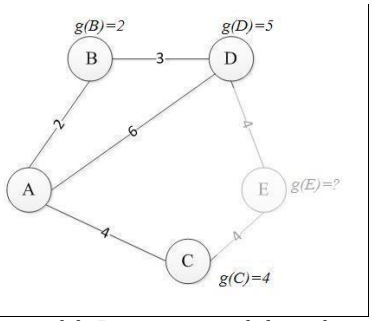
Del ejemplo de la *Figura 3.4*, el orden en que se expandieron los nodos fue: A-B-C-D. Es evidente que cuando el siguiente nodo a expandir es el nodo objetivo significa que la solución ha sido encontrada, debido a que para expandir el nodo, éste tiene que tener el menor costo *g*, así que se puede afirmar que cualquier otro camino que exista a la solución no es óptimo.



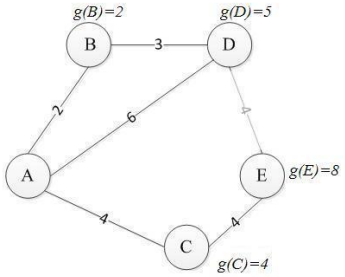
1.- Nodo inicial: A
Nodo objetivo: D
Se expande A.



2.- El que tenga el menor valor *g* será el siguiente nodo a expandir, en este caso B.



3.- Una vez expandido B, se compara *g* de los nodos que no han sido expandidos (nótese que *g(D)* se actualizó). El menor es C.



4.- Se repite la comparación *g(n)* en los nodos no expandidos. En este caso, se observa que el siguiente nodo sería D, pero como D es el nodo objetivo, la búsqueda ha terminado.

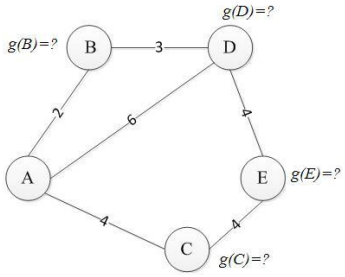
Figura 3.4. Ejemplo de UCS

3.7.1.6 Algoritmo de Dijkstra

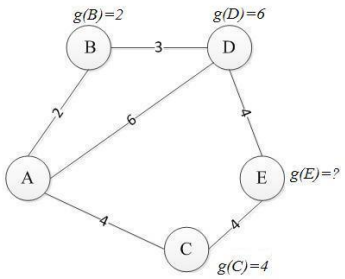
Otra de las estrategias de búsqueda desinformada utilizadas cuando el costo de nodo a nodo es variable es el *algoritmo de Dijkstra*. Fue creado por el científico en computación holandés Edsger W. Dijkstra y es una estrategia que se encarga de encontrar el menor costo a cada nodo partiendo del nodo origen.

El algoritmo de Dijkstra dio origen a UCS [13]. La expansión de nodos en ambas estrategias es equivalente. Se diferencian en que el algoritmo de Dijkstra no conoce el nodo objetivo, UCS sí. Otra diferencia es que en el algoritmo de Dijkstra todos los nodos se insertan a una cola de prioridad desde el principio, mientras que en UCS los nodos son introducidos a la cola conforme se van generando [30]. Esto nos permite concluir que, en un principio, el algoritmo de Dijkstra tiene la información de cuántos nodos hay en el problema y cuáles son.

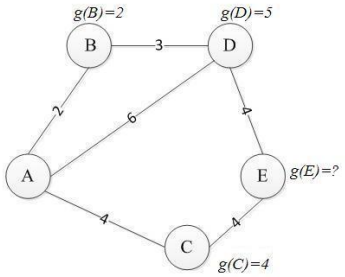
En la *Figura 3.5* se muestra el ejemplo de UCS (*Figura 3.4*) aplicado al algoritmo de Dijkstra. Los costos *g(n)* obtenidos con este algoritmo son óptimos. No obstante, cuando la cantidad de datos (o nodos) es muy grande, el tiempo que tarda en calcular el costo del origen a cada nodo es muy alto, aunado a que cada nodo abarca espacio en memoria. [31]



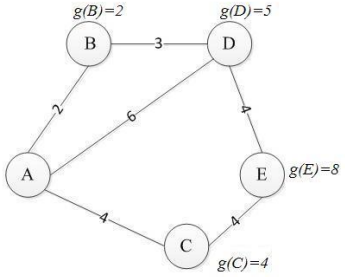
1.- Nodo inicial: A
Inicializa el costo *g* de todos los nodos con un valor muy grande. Comienza revisando los diferentes caminos de A.



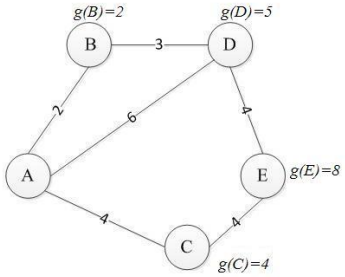
2.- El que tenga el menor valor *g(n)* será el siguiente nodo a expandir, en este caso B.



3.- Una vez expandido B, se compara *g(n)* de los nodos que no han sido expandidos (*g(D)* se actualizó). El menor es C.



4.- Se repite la comparación *g(n)* en los nodos no expandidos. El menor es D, con costo *g* igual a 5.



5.- Al expandir D el resultante fue E y su costo a través de D es mayor que el actual, así que se mantiene. El último nodo a expandir sería E pero como no tiene sucesores, entonces finaliza el algoritmo.

Figura 3.5. Ejemplo del algoritmo de *Dijkstra*

El orden de expansión de los nodos fue: A-B-C-D-E. Este algoritmo "generaliza" la solución, pues el costo de todos los nodos es calculado sin importar si son nodos solución o no, aunque no se sabe con precisión qué nodo es solución. Se puede decir entonces que el algoritmo de Dijkstra es completo y óptimo. Este algoritmo es muy socorrido en el área de redes de datos, para calcular el camino más corto por el cual pueden viajar los datos a través de una red de comunicaciones.

3.7.2 Estrategias de búsqueda informada

Una búsqueda informada usa conocimiento de problemas específicos más allá de la definición del problema en sí mismo. Puede encontrar soluciones más eficientes que una estrategia desinformada.

El enfoque general que se considera en este tipo de estrategias es el de *Best-First Search* (búsqueda del primero mejor). En esta búsqueda el nodo es seleccionado para expansión basado en una función de evaluación *f(n)*, donde *n* es el nodo en cuestión. En la práctica, el nodo con la evaluación más baja es seleccionado para expansión.

Este tipo de estrategias utilizan el término de heurística que se puede definir como una técnica o algoritmo para la solución eficiente de problemas [13] [15]. Esta técnica calcula un valor estimado de cada nodo hacia el nodo objetivo.

La heurística, que se suele expresar como una función *h(n)* donde *n* es un nodo, produce una solución que puede ser buena o aceptable y resuelve problemas simples que contienen la solución al problema más complejo. [15]

En la *Figura 3.6* se muestra un ejemplo para las búsquedas informadas. En él puede observarse que cada nodo cuenta con una función heurística y cada arista entre nodos tiene un costo. Ambos datos son utilizados por las estrategias de búsqueda informada. El nodo inicial es A, mientras que el nodo solución es J.

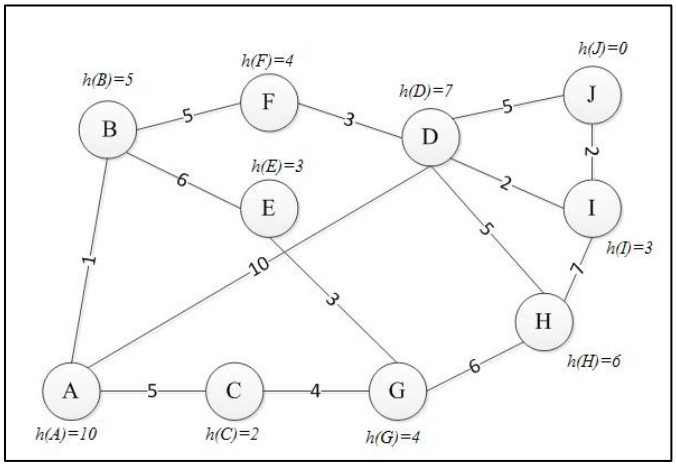
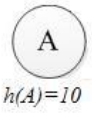


Figura 3.6. Ejemplo de grafo para búsquedas informadas.

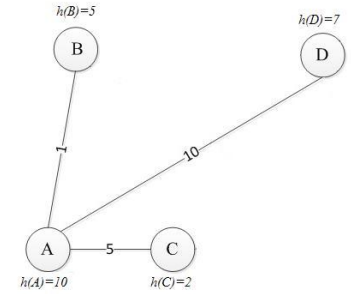
3.7.2.1 Greedy Best-First Search

Por algunos, conocida como *Greedy Search* (Búsqueda Voraz o Ávida), y por otros como únicamente *Best-First Search*, el nombre que le dan Russell y Norvig en [13] a esta estrategia es el de *Greedy Best-First Search* (Búsqueda Voraz del Primero Mejor). Es una

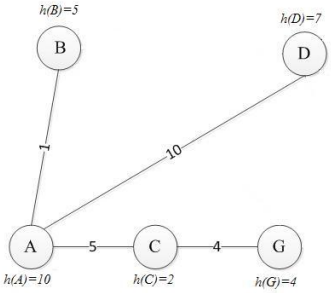
búsqueda informada en la que la decisión de expandir el nodo se hace eligiendo el nodo más próximo al objetivo en base a que esto puede llevarnos a una solución rápidamente.



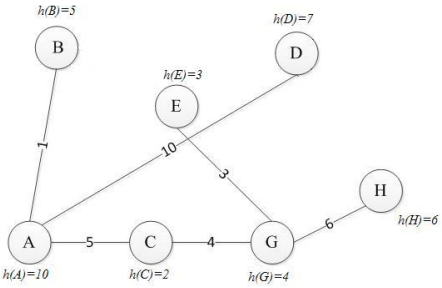
1.- Nodo inicial: A
Se expande.



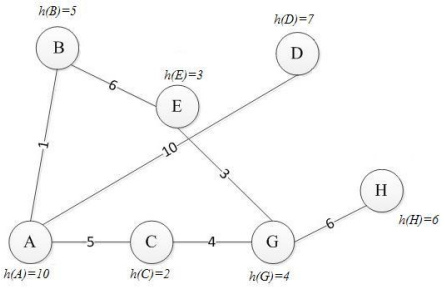
2.- Compara las funciones heurísticas de cada nodo resultante. El que tenga el menor valor de heurística será el camino a tomar, en este caso hacia C. C no es solución y se expande.



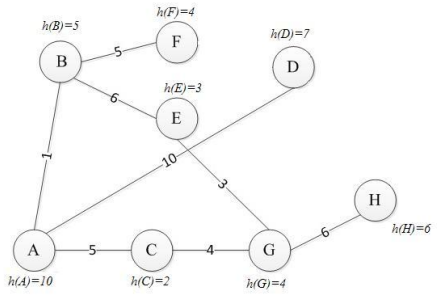
3.- Una vez expandido C, se revisan los nodos generados y se elige el de la menor heurística. El único nodo generado es G, por lo tanto, el siguiente nodo es G que no es solución y se expande.



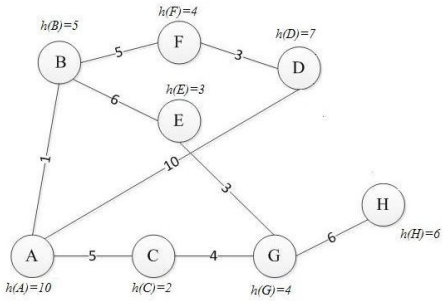
4.- Los nodos generados fueron E y H, con heurística de 3 y 6, respectivamente. El siguiente nodo será E y no es solución, por lo que se expande.



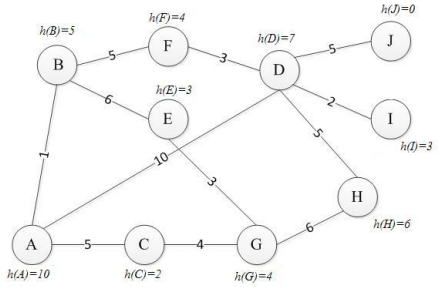
5.- El nodo resultante de E es B, pero éste no ha sido expandido. Consecuentemente, será el siguiente nodo, y como no es solución, se expande.



6.- El resultante de B es F, es el siguiente nodo y no es solución.



7.- D resulta de F, por ello, será el siguiente nodo y no es solución, así que se expandirá.



8.- Los nodos resultantes de D son H, I y J. El de menor heurística es J y su valor heurístico es 0, lo que quiere decir que J es solución.

Figura 3.7. Ejemplo de *Greedy Best-First Search*

En la *Figura 3.7*, se muestra un ejemplo de la búsqueda *Greedy Best-First*. Dicha búsqueda evalúa los nodos usando sólo la función heurística: $f(n)=h(n)$. Donde *n*: Nodo, *f*: Función de evaluación del nodo *n* (costo), *h*: Función heurística del nodo *n*.

Cabe mencionar que esta búsqueda no necesariamente nos da la solución óptima, sin embargo encuentra una posible solución de una manera más rápida. *Greedy Best-First Search* es parecido a *Depth-First Search* en la forma en la que se sigue un único camino para llegar a la meta, pero regresará cuando encuentre un nodo no expandible. Tiene los mismos defectos que DFS, no es óptimo y es incompleto (porque puede comenzar a recorrer un camino infinito y nunca regresar a intentar otras posibilidades). En la mayoría de las búsquedas, la complejidad en tiempo y espacio del peor caso será exponencial ($O(b^m)$), donde *m* será la profundidad máxima. Sin embargo, si se plantea una buena función heurística la complejidad se podrá reducir significativamente. Dicha reducción dependerá del problema en particular y en la calidad de la heurística. [13]

En el ejemplo mostrado, es evidente que se encontró la solución. Sin embargo no es la mejor, pues el costo que resulta de las aristas recorridas es el costo real para llegar a la solución. El orden de recorrido fue el siguiente: A-C-G-E-B-F-D-J. El costo real fue de 5+4+3+6+5+3+5=28. El recorrido dependerá de la función heurística, cuanto sea mejor la heurística utilizada, se obtendrá un mejor resultado.

3.7.2.2 A Search*

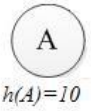
A Search* (pronunciado como *"A-star Search"*, o en español, Búsqueda A-estrella) es una búsqueda informada en la que se evalúan los nodos con la combinación del costo para alcanzar ese nodo ($g(n)$), y el costo del nodo para llegar al objetivo ($h(n)$), es decir: $f(n)=g(n)+h(n)$. Donde $f(n)$ será el costo estimado de la solución más barata a través de *n*.

De esta manera, si se pretende encontrar la solución más próxima al objetivo, algo conveniente sería intentar primero el nodo con el menor valor de $g(n)+h(n)$. Ello hace que esta estrategia de búsqueda sea demasiado razonable, pues dependiendo de que la función heurística $h(n)$ satisfaga ciertas condiciones, la búsqueda A* puede llegar a ser completa y óptima. [13]

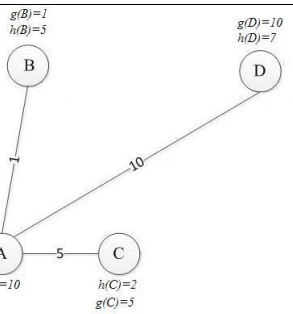
Cada que un nodo se expande en todos sus posibles consecuentes, se realiza la decisión de qué camino tomar, considerando la función antes mencionada. Una vez que se encuentra el nodo con el menor costo, se expande ese nodo en sus posibles soluciones, y así sucesivamente; hasta encontrar la solución, la cual será la óptima. La complejidad será relativa al problema en cuestión, aunque la del peor caso llegará a ser $O(b^m)$. [13]

Se muestra un ejemplo en la *Figura 3.8*, el que se observa cómo se resuelve el problema planteado anteriormente con la búsqueda A*. Vemos que se muestra el mejor camino para

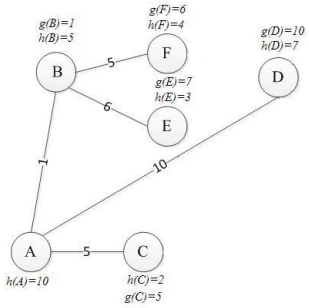
llegar a la solución, que es J, con un costo real de 13, cuyo camino recorrido fue: A-B-F-D-I-J.



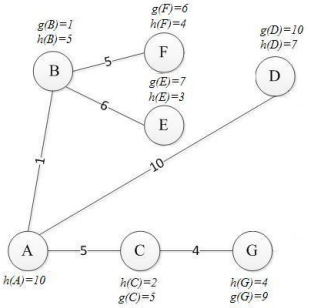
1.- Nodo inicial: A
Se expande.



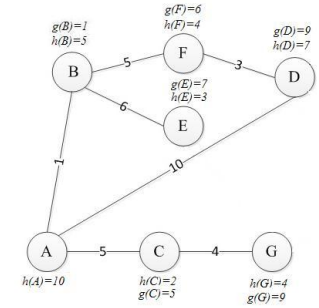
2.- Calcula *f(n)*. El que tenga el menor valor será el camino a tomar, en este caso hacia B, ya que la suma es 6.



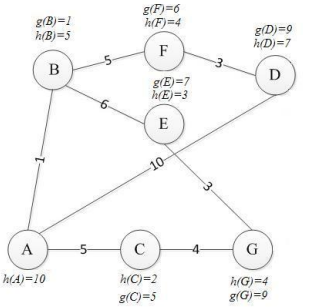
3.- Una vez expandido B, se calculan las *f(n)* y se comparan, junto a los generados anteriormente. En este caso, ninguno de los generados es el menor, el cual es C.



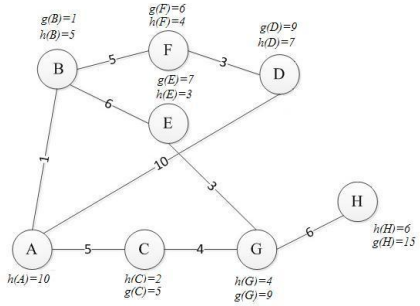
4.- El nodo generado fue G y una vez más se comparan las *f(n)*. Ahora la menor coincide en dos nodos, E y F. Se hará primero F.



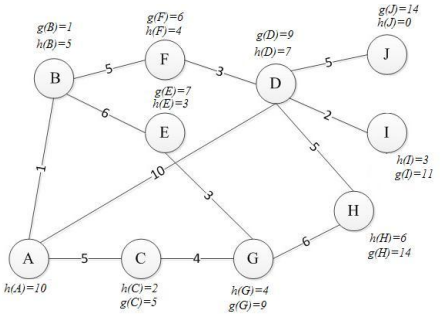
5.- El nodo resultante de expandir F es D, pero nótese que llegar a D por F es menos costoso que llegar desde A, por lo que se actualiza el valor de *g(D)*. El siguiente nodo a expandir será E.



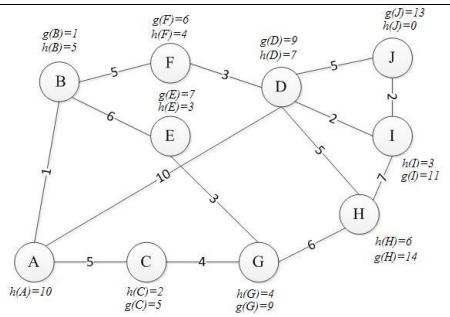
6.- El resultante es G, pero es más costoso llegar por E, así que el valor actual se mantiene. El siguiente nodo a expandir es G.



7.- H resulta de expandir G, ahora el menor *f(n)* es D, así que se expande.



8.- Los nodos al expandir D son H, I y J. Se actualizó H, e I y J tienen el mismo valor *f(n)*, pero I tiene menor *g(n)*, así que se expande.



9.- I se expande hacia H y J que ya aparecían, pero en el caso de J, su valor *g(n)* se actualiza, y al ser J el de menor valor *f(n)* y ser solución, termina la búsqueda.

Figura 3.8. Ejemplo *A* Search*

Las estrategias de búsqueda analizadas, tanto informadas como desinformadas, son sólo algunas de las que existen actualmente. Más adelante, en este trabajo, se explicará el algoritmo *Flood-Fill* que se implementa como una **búsqueda informada**, lo cual nos ayuda a llegar a una solución de una manera más rápida.

4 Robot Móvil

El robot a implementar en el laberinto MicroMouse será un robot que tendrá la capacidad de moverse por sí mismo, que pueda percibir la información de las paredes que conforman al laberinto y también que estime su orientación actual respecto del laberinto. Primeramente, para poder recorrer el laberinto se deberá utilizar algún tipo de actuador de desplazamiento las cuales, en este caso, serán llantas que no sobrepasen las dimensiones del robot (definidas en el estándar de competencia del IEEE). Para tener la posibilidad de mover estas llantas se deberán incorporar motores, los cuales permitirán hacer girar las llantas a utilizar. Adicionalmente, se usarán sensores, que permitirán la visibilidad de muros y orientación del robot a lo largo del laberinto, que posibilitarán al robot el encontrar la solución.

Para la toma de decisiones, el movimiento de los actuadores, la adquisición de información del entorno, y demás factores necesarios para la resolución del laberinto, es imprescindible el uso de un dispositivo que funcione como "cerebro" del robot. Este dispositivo tendrá que ser apto para realizar lo antes descrito. Uno de los más utilizados en aplicaciones de robótica y que se implementan en diversos sistemas pequeños son los llamados microcontroladores.

4.1 Microcontrolador

Para definir un microcontrolador, es necesario definir un microprocesador.

Un microprocesador es la unidad básica de una computadora y se encarga de realizar las operaciones necesarias para poder ejecutar una instrucción. Algunas veces se le denomina CPU (*Central Processing Unit*) y se comunica con la memoria y los dispositivos de E/S a través de buses. Está compuesto de:

- ALU (Unidad Aritmético-Lógica): Es aquella que realiza las operaciones aritméticas y lógicas.
- Registros: Son localidades definidas de memoria dentro de la CPU y sirven para almacenar datos que requieren ser utilizados para el procesamiento de alguna instrucción, pues si estos datos se tomaran desde otra unidad de memoria, la disponibilidad de ésta sería mucho más lenta.
- Unidad de Control: Es la que envía las señales de control a los diferentes componentes y dispositivos presentes en la ejecución de una instrucción. Se considera que es la máquina de estados de una computadora (de acuerdo a las instrucciones definidas).

Un microprocesador es un dispositivo de circuitería integrada que hace posible construir un procesador y que consiste tanto de una unidad de memoria (Registros), como de un control interno (Unidad de Control).

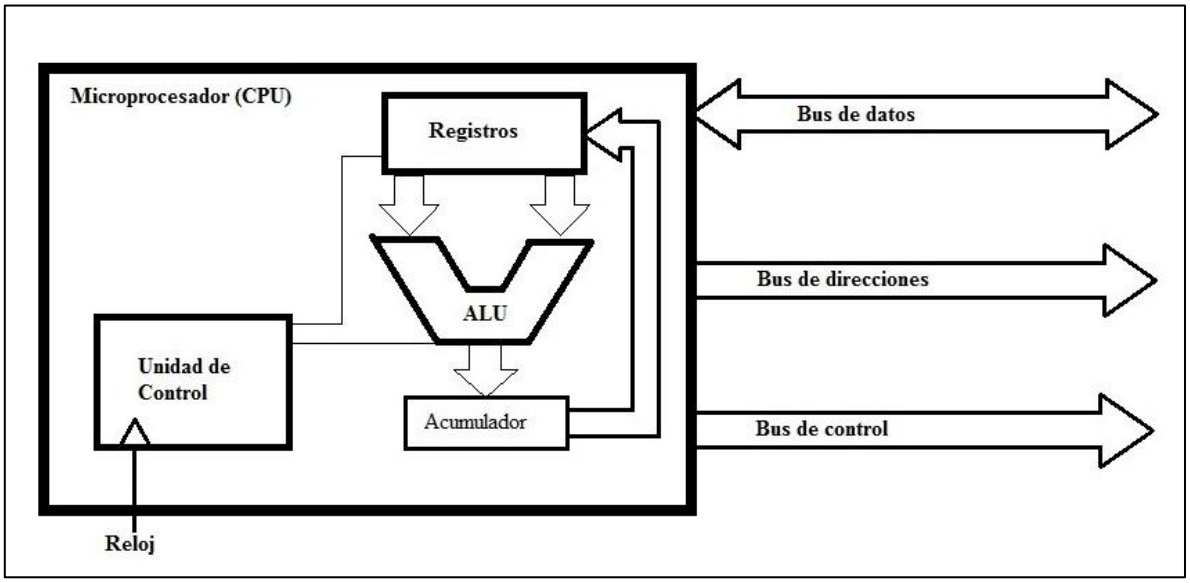


Figura 4.1. Diagrama de bloques general de un microprocesador

Un microcontrolador es una clasificación de microprocesador. Éste es un circuito integrado que se compone de una unidad de procesamiento (CPU), una unidad de memoria (en muchos casos dividida en datos y programa) y periféricos hacia dispositivos de Entrada/Salida.

Un microcontrolador también es considerado un tipo de microcomputadora que se trata de un dispositivo de circuitería integrada capaz de programarse para realizar operaciones específicas. Son utilizados en la actualidad en todo tipo de dispositivos electrónicos, desde calculadoras hasta sistemas de control para máquinas eléctricas, especialmente en sistemas de control embebidos, ya que éstos últimos implementan algoritmos de control complejos que requieren un alto procesamiento para hacer cálculos indispensables.

Hoy en día, existe gran cantidad de microcontroladores, entre los más comunes se encuentran los microcontroladores PIC® [9] de la empresa Microchip, y los AVR® [10] de Atmel. A pesar de la variedad que existe en el mercado, hay un punto en el que los fundamentos de cada uno suelen ser los mismos, implementando los mismos diseños y arquitecturas.

[9] PIC es una marca registrada de Microchip Technology Inc. *http://www.microchip.com/*
[10] AVR es una marca registrada de Atmel Corporation. *http://www.atmel.com/*

Es así que los microcontroladores combinan recursos de hardware de una microcomputadora en un único circuito integrado. Los principales recursos que se encuentran dentro de un microcontrolador son:

- CPU: Es el cerebro de la microcomputadora.
- Memoria: Unidad donde se ubica el principal almacenamiento de datos e instrucciones.
- Periféricos: Medios por los cuales el microcontrolador puede tener interacción con otros dispositivos, pudiendo realizar comunicación entre ellos. [32]

El sistema de memoria es usado primordialmente para dos propósitos:

- Proporcionar un sistema de almacenamiento para los datos y/o los operandos.
- Proporcionar facilidad para el almacenamiento del programa (grupo de comandos o instrucciones).

La porción de la memoria que almacena los operandos a utilizar en el programa, se llama memoria de datos, y como su nombre lo dice, es la parte de la memoria que guarda los datos utilizados durante la ejecución de un programa. Esta memoria suele ser memoria de tipo RAM, ya que la asignación de valores y de localidades de memoria se realizan en tiempo de ejecución, por lo que no es necesario que se mantenga almacenada la información de los datos de manera permanente.

4.1.1 Memoria RAM

La memoria RAM (*Random Access Memory*) es un tipo de memoria que se considera volátil, debido a que la memoria no se mantiene de manera permanente en el circuito integrado. Esta memoria es la memoria principal de una computadora, y en nuestro caso, de la microcomputadora. Se puede escribir y leer a través de ella, y mientras el circuito esté alimentado mantiene la información. Respecto a la memoria de datos, el dato se ingresa a una determinada localidad de memoria y es obtenido a través de esa misma localidad. [33]

4.1.1.1 SRAM

Una SRAM (*Static Random Access Memory*) es un tipo de RAM que almacena la información en una configuración de compuerta lógica con *flip-flop*. De esta manera, la SRAM almacenará la información mientras ésta se encuentre suministrada de voltaje. Las memorias SRAM permiten selección aleatoria de las localidades de memoria en un arreglo de memoria. Son volátiles y permiten almacenar datos en arreglos de memoria sin necesidad de utilizar ciclos de reloj o realizar una lógica de refresco.

Las operaciones que se realizan en una memoria SRAM están descritas a continuación:

- **Escritura:** La información a almacenar en el arreglo de memoria es conectada a la línea de E/S de datos de la RAM. La localidad a la cual será almacenada esa información está determinada por la palabra codificada en la entrada de dirección de la RAM, el valor de decodificado por el decodificador lógico de direcciones. El resultado es la activación de una de las líneas de selección de memoria conectadas a la localidad en el arreglo de memoria deseado. La línea de CS (Chip Select) es activada, conectando las líneas de datos de entrada al arreglo de memoria y, por lo tanto, la entrada de control lógica es habilitada. Entonces, el comando de escritura es generado, provocando que el contenido de la localidad de memoria seleccionada sea actualizada con la nueva palabra de datos.
- **Lectura:** La localidad de memoria seleccionada está determinada por la codificación de la entrada a través del decodificador lógico de direcciones. Esto activa la localidad de memoria seleccionada. Cuando la línea E (Enable) es activada y el comando de lectura es generado, el contenido de la localidad de memoria seleccionada está conectada a las líneas de salida de datos. [34]

4.1.1.2 DRAM

Una DRAM (*Dynamic Random Access Memory*) es un tipo de RAM y está construida de pequeños capacitores que tienen "fuga" de electricidad, dicha capacidad de los capacitores se le denomina capacitancia parásita. Una DRAM requiere de una recarga cada pocos milisegundos para mantener su información.

A diferencia de las SRAM, una DRAM es mucho más barata y lenta, sin embargo, varios diseñadores las utilizan porque son mucho más densas, pues pueden almacenar varios bits por cada chip, utilizan menos potencia, y disipan menos calor que una SRAM. Es por eso que en una computadora tradicional, ambas tecnologías son utilizadas de manera combinada. La DRAM para memoria principal y la SRAM para memoria caché.

A continuación, se enlistan las diferentes operaciones que se pueden realizar en una memoria RAM dinámica:

- **Escritura:** El bus de direcciones de la DRAM es decodificado por el renglón y columnas en un decodificador lógico de direcciones. La salida selecciona una localidad de memoria específica. Cuando el CS y el comando de escritura son activados, la entrada de datos es escrita dentro de la localidad de memoria seleccionada.
- **Lectura:** La operación de lectura es similar a la operación de escritura, excepto que el comando de lectura es generado. Esto resulta de encaminar el contenido de la

localidad de memoria seleccionada a través del circuito de E/S de datos al bus de salida de datos.

- **Modo de refresco:** Alta densidad por bit, menor disipación de calor y lectura/escritura de datos más rápida son logrados con celdas de memoria capacitivas, pero estos capacitores deben ser recargados o refrescados de manera periódica para retener los datos en el arreglo de memoria. Se deben utilizar generadores de ciclos de reloj, ya sean internos o externos, para mantener operaciones de refresco confiables dentro del circuito.

El desempeño de una DRAM depende de ciertos parámetros, según se menciona en [35]. Éstos son:

- Ciclo de tiempo de lectura
- Ciclo de tiempo de escritura
- Tiempo de acceso
- Tiempo de refresco
- Suministro de potencia
- Disipación de potencia
- Organización de memoria

4.1.2 Memoria ROM

La memoria ROM (Read-Only Memory) es otro tipo de memoria que se considera de sólo lectura y ésta es permanente en el circuito integrado, mientras no se haga algún procedimiento para borrar la información contenida dentro de la memoria. Por esa razón es que se utiliza para la memoria de programa del microcontrolador.

Existen diferentes tipos de memorias ROM:

- **PROM:** Memoria de tipo ROM Programable. Este tipo de memorias eran posibles de programarse por el usuario, pero sólo permitía guardar información dentro de ella una sola vez.
- **UVEPROM:** Esta memoria se puede programar de manera eléctrica y se puede borrar su información a través de rayos UV para poder programarla nuevamente. El número de veces que se puede programar es muy limitado.
- **EEPROM:** Esta memoria de tipo ROM mejoró el aspecto de borrar y guardar dentro de ella, ya que mantiene la opción de programarse eléctricamente, pero ahora se puede borrar de la misma manera, lo que permite aumentar el número de ocasiones que se puede programar.
- **Flash ROM:** Es un caso específico de memoria EEPROM que permite realizar operaciones de lectura y escritura de manera más rápida que una EEPROM común.

En el caso de algunos microcontroladores, se implementan memorias tipo Flash para el almacenamiento de las instrucciones de programa, además, implementan una memoria EEPROM para almacenar algo más que no sean instrucciones de programa, como pueden ser datos de información para la ejecución del programa, o en nuestro caso, almacenar la información del laberinto en caso de que el programa se reinicie.

4.2 Arquitectura de los microcontroladores

La memoria tanto de los microprocesadores, como de los microcontroladores, almacenan datos e instrucciones de programa. Las instrucciones necesitan moverse secuencialmente a través del CPU para ser decodificadas y ejecutadas. Los datos pueden ser leídos desde la memoria o escritos en la misma por el CPU. Por lo tanto, la manera en que la memoria está organizada y la manera en la que se comunica con el CPU determinan el desempeño de los dispositivos. Los dos modelos genéricos de hardware para la estructura de la memoria son arquitectura Von Neumann y arquitectura Harvard.

La arquitectura Von Neumann fue propuesta por el matemático John Von Neumann quien fue creador de la ENIAC (*Electronic Numerical Integrator And Calculator*) en la Universidad de Pennsylvania durante la Segunda Guerra Mundial. La propuso con la idea de desarrollar una computadora con programa almacenado.

La arquitectura Harvard fue propuesta por Howard Aiken cuando desarrolló las computadoras conocidas como Mark I, II, III y IV en la Universidad de Harvard. Las computadoras Mark fueron las primeras en utilizar diferentes memorias para almacenar datos e instrucciones de manera separada, siendo así una aproximación muy diferente que la computadora de programa almacenado.

La arquitectura Von Neumann utiliza una memoria para almacenar datos e instrucciones de programa. Esto significa que únicamente se implementa un bus que puede acceder a las instrucciones y a los datos. De igual manera, sólo un bus de datos puede transmitir instrucciones de programa o datos. El CPU envía la misma señal de control para leer ya sea un dato o una instrucción. No hay señales independientes para lectura de datos e instrucciones. Por otro lado, aunque la ROM es utilizada para almacenar instrucciones de programa y la RAM para datos, el CPU no reconoce esta distinción y trata ambas memorias de la misma manera.

La arquitectura Harvard utiliza diferentes memorias para almacenar instrucciones y datos. La memoria de programa tiene su propio bus de direcciones (instrucciones) y su propio bus de datos. La memoria de datos también tiene sus propios buses (datos y direcciones). La memoria de programa puede sólo ser leída, mientras que la memoria de datos puede ser leída o escrita (hablando en tiempos de ejecución).

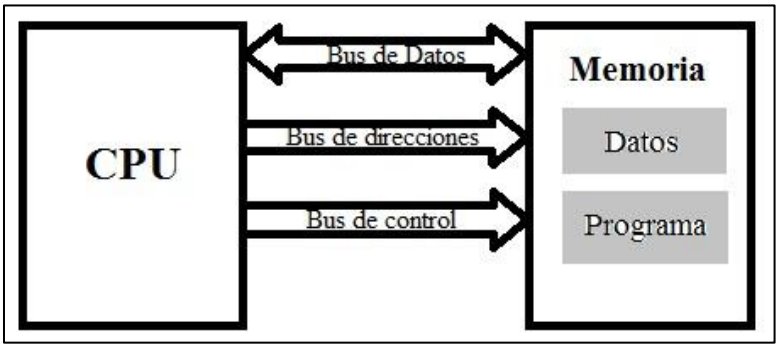


Figura 4.2. Diagrama de la arquitectura Von Neumann

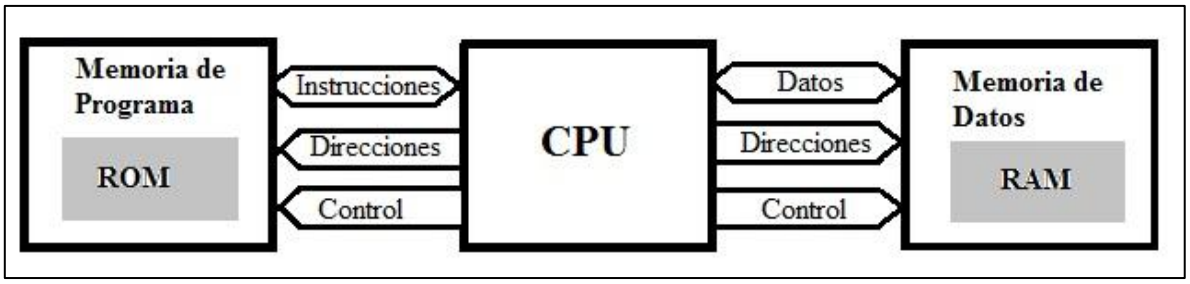


Figura 4.3. Diagrama de la arquitectura Harvard

Una ventaja de la arquitectura Von Neumann sobre la Harvard es que utiliza menos cableado, es decir, menos hardware; lo cual hace más simples las conexiones entre la memoria y el CPU. Sin embargo, la principal desventaja es la incapacidad de no permitir el manejo simultáneo de datos e instrucciones porque únicamente implementa un bus para ambos. Por otro lado, la arquitectura Harvard sí permite el manejo de datos e instrucciones simultáneamente, debido a que el bus de datos es independiente al de instrucciones. Ello quiere decir que la arquitectura Harvard puede ejecutar un programa de manera más rápida que la arquitectura Von Neumann.

En casi todas las microcomputadoras, el CPU utiliza arquitectura Von Neumann porque combina la memoria de datos e instrucciones en un solo bloque y necesita menos líneas de comunicación, lo que reducirá el tamaño del CPU. No obstante, en un microcontrolador, los componentes del sistema están localizados dentro del mismo circuito integrado y por lo tanto no es necesario minimizar los pines. Por ello, la arquitectura Harvard es la elegida para la mayoría de los microcontroladores.

4.3 Ciclo *Fetch*

También conocido como ciclo de *búsqueda-decodificación-ejecución*. Es el ciclo que realiza la CPU para poder ejecutar una instrucción. Independientemente de la arquitectura, este ciclo se realiza en cualquier computadora. El ciclo se ejecuta en 3 pasos, que son el

paso de búsqueda de la instrucción, el paso de decodificación de la instrucción y el paso de ejecución.

En el paso de **búsqueda de la instrucción** se hace la lectura de la instrucción direccionada por el registro de Contador de Programa (PC) y, se obtienen su código de operación y el dato a procesar, si es el caso. Al final del paso de búsqueda, el registro PC se incrementa en 1, de manera que apuntará a la siguiente instrucción.

En el paso de **decodificación** se interpreta la instrucción obtenida dentro del registro de instrucciones. Entonces, se procede a ejecutarla.

En el paso de **ejecución** se procesa la instrucción, y en caso de requerir datos éstos son obtenidos desde la instrucción decodificada o desde la memoria de datos.

En la arquitectura Von Neumann existe un paso intermedio entre la decodificación de la instrucción y la ejecución de la misma, y éste es el de **búsqueda de los datos**. Esto debido a que la arquitectura Von Neumann implementa una memoria para almacenar datos e instrucciones. El primer paso será de búsqueda de la instrucción que almacenará en el PC la dirección del dato a procesar y en el paso de búsqueda de datos se obtendrán los datos requeridos por la instrucción para continuar a la ejecución de dicha instrucción, posteriormente almacenará en el PC la siguiente instrucción a ejecutarse.

4.4 Pipeline

La arquitectura Harvard implementa una secuencia de pasos que logra encadenar una serie de instrucciones de manera que se pueda ejecutar una instrucción por cada ciclo de máquina que realiza la computadora, o microcomputadora, según sea el caso. Esta técnica de ejecución de instrucciones se denomina *Pipeline*. El ejemplo típico del funcionamiento del *pipeline* es el de una planta de ensamblaje de automóviles.

Esto es, supongamos que una planta de ensamblaje está dividida en tres zonas, la primera donde se ensambla el interior del automóvil, la segunda donde se colocan las llantas, y la tercera donde se arma el exterior del automóvil. Al comenzar el ensamblaje, el primer automóvil a armar se encontrará en la parte de ensamblaje del interior, las dos partes siguientes estarán libres por el momento. En el segundo paso, cuando el primer automóvil logró su ensamblaje interior, entrará al ensamblaje de llantas, pero un segundo automóvil comienza a ensamblarse en la primera zona de la planta. En el tercer paso, el primer automóvil entra ahora a la tercera zona (ensamblaje exterior) y el segundo automóvil entrará a la zona de ensamblaje de llantas, mientras que se comienza el ensamblaje de un tercer automóvil en la primera zona. En el cuarto paso, el primer automóvil ha sido finalizado, el segundo entra a la zona de ensamblaje exterior, y el tercero entra a la de ensamblaje de llantas, la primera zona está libre para comenzar el ensamblado del cuarto

automóvil. A partir del cuarto paso, en cada paso siguiente se tendrá un automóvil saliendo de la planta.

La técnica de *pipeline* permite que una computadora se comporte con paralelismo aprovechando cada una de las etapas del ciclo *Fetch*.

4.5 Set de instrucciones

El set de instrucciones es la colección de operaciones básicas para un procesador. El conjunto de instrucciones es utilizado para que un usuario pueda crear programas en lenguaje máquina para desempeñar operaciones matemáticas y/o lógicas. El set de instrucciones está cableado dentro del procesador, el cual determinará el lenguaje máquina para el procesador. Entre más complejo sea el set de instrucciones, el procesador trabajará de manera más lenta.

Los procesadores difieren uno de otro a su set de instrucciones. Si el mismo programa puede ejecutarse en dos procesadores diferentes, se dice que son compatibles. Desde que cualquier procesador tiene su set de instrucciones único, los programas en lenguaje máquina escritos para un procesador normalmente no se ejecutarán en un procesador diferente. Por lo tanto, todos los sistemas operativos y programas de software están construidos dentro de los límites del set de instrucciones del procesador. Así, el diseño del set de instrucciones para el procesador es un aspecto importante para la arquitectura de la computadora, ya que las operaciones realizadas por cada instrucción dependerán de cómo estén conectados los componentes del CPU (Registros, ALU, Unidad de Control, etc.). En base al set de instrucciones, existen dos tipos comunes de arquitecturas: CISC y RISC.

4.5.1 CISC (*Complex Instructions Set Computer*)

CISC, o Computadora de Set de Instrucciones Complejo, es una arquitectura en la cual el set de instrucciones contiene instrucciones que suelen realizar operaciones complejas.

Esta arquitectura nació como necesidad de desarrollar compiladores más fáciles y rápidos. Debido a que el lenguaje máquina y el ensamblador no son fáciles de entender para los programadores, se crearon los lenguajes de alto nivel. Esto ayudó a que la programación pudiera ser más accesible para las masas, pues es parecido al idioma inglés y fue más amigable para el usuario. Sin embargo, las instrucciones en lenguaje de alto nivel necesitaban ser convertidas en su equivalente en bajo nivel antes de que el procesador pudiera ejecutarlas. Esta traducción del programa en alto nivel a un programa en bajo nivel es realizada por el compilador. Con el desarrollo de los lenguajes de alto nivel, los compiladores se volvieron más poderosos y proporcionaban mayores características, como las funciones matemáticas complejas. Escribir compiladores para tales lenguajes de alto

nivel fue cada vez más difícil. Los compiladores tenían que traducir subrutinas complejas en grandes secuencias de instrucciones en lenguaje máquina.

De esta manera, se creó la arquitectura CISC, con el único motivo de que los fabricantes de procesadores basados en CISC desarrollaran procesadores con set de instrucciones más extensas y complejas, por lo que redujo la carga del compilador de traducir instrucciones en lenguaje máquina y se implementó dentro del procesador. Por ejemplo, en lugar de hacer un compilador que tradujera una secuencia de instrucciones larga para calcular una raíz cuadrada, un procesador CISC incorpora una circuitería cableada para desarrollar el cálculo de la raíz cuadrada en un solo paso.

Algunas de las ventajas de una arquitectura CISC [36], son las siguientes:

- Los procesadores CISC utilizan las tecnologías disponibles para optimizar el desempeño de la computadora.
- La arquitectura CISC utiliza Hardware de propósito general para llevar a cabo comandos. Por lo tanto, nuevos comandos pueden ser agregados dentro del chip sin cambiar la estructura del set de instrucciones.
- La microprogramación es tan fácil de implementar como el lenguaje ensamblador y mucho menos caro que el cableado de la unidad de control.
- Como cada instrucción se vuelve más capaz, menos instrucciones pueden ser usadas para implementar determinadas tareas. Esto hace un uso eficiente de la memoria principal relativamente lenta.
- Un set de instrucciones de microprograma puede ser escrito para emparejar la construcción de lenguajes de alto nivel, el compilador no tiene que ser muy complejo.

Las desventajas que se presentan en una arquitectura CISC [36], se enlistan a continuación:

- Los procesadores de generaciones anteriores de computadoras fueron como un subconjunto de las versiones exitosas, por ello, el set de instrucciones y el chip como hardware se volvió complejo con cada generación de computadoras.
- Instrucciones diferentes toman diferente cantidad de tiempo de reloj en ejecutarse, y así reduce el tiempo de todo el desempeño de una máquina.
- La arquitectura CISC requiere reprogramación continua del hardware on-chip (dentro del circuito integrado).

El diseño CISC incluye la complejidad del hardware necesaria para desarrollar varias funciones, y la complejidad del software on-chip necesaria para hacer que el hardware realice las acciones correctas. Es decir, modificar el diseño de la arquitectura CISC tendrá repercusiones tanto en hardware como en software.

4.5.2 RISC (*Reduced Instructions Set Computer*)

RISC, o Computadora de Set de Instrucciones Reducido, es una arquitectura que utiliza un set de instrucciones pequeño y altamente optimizado. El concepto detrás de esta arquitectura es un pequeño número de instrucciones que son más rápidas en ejecución, comparado a una instrucción compleja. Para implementar esto, la arquitectura RISC simplifica el set de instrucciones del procesador, lo cual ayuda a reducir el tiempo de ejecución. La optimización de cada instrucción en el procesador está hecha a través de la técnica de *pipelining*, lo cual permite que el procesador trabaje en diferentes pasos de la instrucción al mismo tiempo; usando esta técnica, más instrucciones pueden ser ejecutadas en un tiempo más corto. Esto se logra traslapando los ciclos de búsqueda, decodificación y ejecución de dos o más instrucciones. Para evitar las interacciones con memoria o reducir los tiempos de acceso, la arquitectura RISC incorpora un mayor número de registros.

Como cada instrucción es ejecutada directamente usando el procesador, no se necesita una circuitería de mayor cableado (que se usa en instrucciones complejas). Ello hace posible que la arquitectura RISC sea más pequeña, consuma menor potencia y disipe menos calor que una arquitectura CISC. Gracias a estas ventajas, los procesadores RISC son ideales para aplicaciones embebidas, como teléfonos celulares, cámaras digitales, tabletas, etc. Además, el simple diseño de un procesador de este tipo reduce el tiempo de desarrollo comparado con el de un CISC.

Las ventajas de una arquitectura RISC [36], son:

- El *pipeline* logrado con el set de instrucciones reducido permite que un procesador RISC tenga de dos a cuatro veces un mejor desempeño que un procesador CISC
- Como el set de instrucciones es simple, utiliza menor espacio en el chip. Hace posible otro tipo de funciones dentro del mismo chip, tal como las unidades de gestión de memoria o unidades aritméticas de punto flotante. Los chips más pequeños permiten que el fabricante pueda agregar más partes en una sola pieza de silicio, lo cual disminuye el precio de manera significativa.
- Pueden ser diseñados con mayor rapidez y pueden tener más ventajas sobre otros desarrollos tecnológicos que los diseños de CISC.

Algunas desventajas que presenta una arquitectura RISC [36], son:

- El desempeño de un procesador RISC depende estrictamente del código que se está ejecutando. Si el compilador utilizado realiza un trabajo pobre de temporizar instrucciones, el procesador puede desperdiciar tiempo esperando por el resultado de una instrucción antes de que proceda con la instrucción siguiente.

- Los procesadores RISC requieren un sistema de memoria muy rápido para alimentar instrucciones. Los sistemas basados en RISC regularmente contienen grandes memorias caché, usualmente en el chip mismo.

4.6 Módulos

Los microcontroladores están diseñados con una serie de módulos que permitirán diseñar sistemas relativamente complejos.

Para nuestro caso, sólo se hablarán de algunos de los módulos que contiene el PIC18F4520 de Microchip y serán los que se implementen en este trabajo, ya que la cantidad de módulos es amplia como para profundizar en cada uno de ellos.

4.6.1 Comunicación Paralela

Los microcontroladores cuentan con puertos digitales de entrada/salida que permiten conectar otros dispositivos o cualesquiera elementos para interactuar con el mismo microcontrolador. Estos puertos suelen utilizarse como comunicación paralela.

Los puertos paralelos son circuitos internos utilizados para su interacción con periféricos o dispositivos externos. Generalmente, esta conexión tiene un número *n* de líneas (de manera típica, *n=8*) para transferir datos y puede o no tener *m* líneas adicionales para el control de transferencia de datos (no suelen ser necesarias).

Desde el punto de vista de programación, los puertos son localidades de memoria de datos. Por ello, al menos una dirección es necesaria para representar la entrada de datos.

Una ventaja que tiene este tipo de comunicación es que se puede enviar una gran cantidad de datos en un tiempo relativamente corto, esto debido a que por cada ciclo se envían *n* bits de datos. La desventaja principal es que las líneas de comunicación abarcan más hardware, pues la comunicación paralela requerirá mayor cableado.

4.6.2 Comunicación Serial

La comunicación serial es un protocolo muy común en las computadoras actuales para la comunicación entre dispositivos.

Básicamente, el concepto de comunicación serial se basa en el envío y recepción de bytes de información bit por bit (un bit a la vez). A diferencia de la comunicación paralela que permite enviar bytes completos a la vez, la comunicación serial es más sencilla y alcanza distancias más largas, aunque es más lenta al ser bit por bit.

Una ventaja que posee la comunicación serial respecto a la paralela es el uso de hardware. Para realizar comunicación paralela, se necesitan varias líneas de transmisión y/o recepción, mientras que con la comunicación serial son suficientes una línea de transmisión y otra para recepción, además de la referencia. En algunos casos, también se requiere una línea de reloj (pulsos) que hace que los dispositivos se sincronicen en tiempos.

Las características más importantes en este tipo de comunicación con:

- **Velocidad de transmisión (*baud rate*):** Es el número de bits por segundo que se pueden transmitir y se mide en baudios (*bauds*). La velocidad de transmisión es inversamente proporcional a la distancia de la línea de transmisión, es decir, cuanto más sea la velocidad de transmisión de los dispositivos seriales, menor será la distancia máxima que pueda haber en la línea de transmisión de dichos dispositivos.
- **Bits de datos:** Hace referencia a la cantidad de bits que se están transmitiendo en la comunicación entre dispositivos. Cuando se envía un paquete de información a través de la línea de comunicación, no necesariamente serán de 8 bits. Las cantidades más comunes en esta comunicación son de 5, 7 y 8 bits.
- **Bits de parada:** Se utiliza para indicar el fin de la comunicación de un paquete. Los valores típicos son 1, 1.5 y 2 bits. Se utiliza a manera de sincronizador entre dispositivos cuando sus relojes no necesariamente se encuentran sincronizados. Indica al dispositivo receptor que la transmisión de un paquete ha finalizado
- **Paridad:** Se utiliza para verificar si en la transmisión existieron errores o no, esto indicará si hubo alguna señal de ruido durante la transmisión. Existen cuatro tipos de paridad: par, impar, marcada y espaciada. Para paridad par o impar, el protocolo fija un bit que indica la paridad de los datos (un bit al final, posterior a los bits de datos, 0 lógico para paridad par y 1 lógico para paridad impar). El número de bits de datos que se encuentran en 1 lógico son los que indican si existe paridad par o impar. Por ejemplo, supongamos la transmisión del byte 00101100. El número de bits en 1 lógico son 3, lo que indica que la paridad es impar, por ello, el bit de paridad será 1 y se agrega después de los bits de datos, es decir, se enviará 001011001. Para paridad marcada o espaciada, no se verifican los bits de datos, únicamente se envía un 1 lógico para paridad marcada y un 0 lógico para paridad espaciada. [37]

Se considera que existen dos tipos de comunicación serial que dependen de la manera en la que se comunican los dispositivos, específicamente en la manera en que se sincronizan.

4.6.2.1 Síncrona

La comunicación serial síncrona es aquella en la que es necesario que los relojes de pulsos de cada dispositivo conectado se encuentren sincronizados a un solo reloj. La transferencia de bits de forma continua se realiza a la velocidad que dicta el reloj de pulsos.

Sin embargo, en comunicación serial de larga distancia, cada dispositivo se maneja con relojes independientes sincronizados a una misma frecuencia, aunque de manera periódica se envía una señal de sincronización entre dispositivos para mantener las señales sincronizadas. [38]

4.6.2.2 Asíncrona

En la comunicación serial asíncrona no necesariamente los relojes de pulsos deben estar sincronizados, pero debe haber una sincronía entre los dispositivos conectados. Los relojes de pulsos de cada dispositivo son independientes y no será necesario que tengan frecuencias iguales. [38]

El ejemplo de esta comunicación en microcontroladores es la del protocolo RS-232, en la que no es necesario utilizar un reloj común entre dispositivos y la cual hace posible la comunicación entre una computadora y el microcontrolador. Únicamente requiere dos líneas (transmisión y recepción) y una línea de acoplamiento de tierras. Las frecuencias de los dispositivos son diferentes, mientras que la frecuencia de la computadora está en GHz, la del microcontrolador usualmente se encuentra en MHz.

La transmisión serial asíncrona, detallada en [38], se realiza de la siguiente manera:

a) Mientras no hay transferencia de datos, la línea de transmisión se mantiene en 1.
b) Para iniciar la transmisión de un dato, se envía un bit de inicio, que siempre será 0 lógico.
c) Los bits de datos siempre van inmediatamente después del bit de inicio.
d) Para finalizar la transmisión, después del último bit de datos, la línea regresa a 1 lógico para indicar el bit de paro.
e) La línea permanecerá en estado lógico de 1 hasta que se realice la siguiente transmisión de un dato.

Existen circuitos integrados disponibles que proporcionan una interfaz entre la computadora y cualquier otro dispositivo compatible con el protocolo. Estos circuitos se denominan UART (*Universal Asynchronous Receiver-Transmitter*, o Receptor-Transmisor Asíncrono Universal). La mayoría de microcontroladores lo tienen integrado en el mismo circuito.

4.6.2.3 MSSP (Master Synchronous Serial Port)

El módulo MSSP es una interfaz que permite la comunicación serie de manera síncrona con otros dispositivos o microcontroladores y que actualmente es muy utilizada. Muchos dispositivos ya utilizan este módulo, por ejemplo, memorias, acelerómetros, convertidores A/D, etc. El modo de operación de MSSP puede ser a través de dos protocolos distintos:

SPI™ (*Serial Peripheral Interface*)[11], e I²C™ (*Inter-Integrated Circuit*)[12]. Ambos protocolos implementan funciones de maestro y esclavo. [39]

4.6.2.3.1 SPI

SPI es un protocolo de comunicación serial síncrona. Permite realizar la transferencia de datos entre dos o más dispositivos, de manera que uno de ellos se considera el maestro, que comienza la comunicación y establece la transferencia, y los demás dispositivos se consideran esclavos.

El protocolo de comunicación SPI requiere de cuatro líneas de transmisión principales, una línea de reloj de sincronización, SCK; una línea de transferencia de datos de salida, SDO; y una línea de recepción de datos (entrada), SDI. Además, es necesario definir otra línea de comunicación llamada *Slave Selection* (SS#) que es utilizada por el dispositivo maestro para seleccionar alguno de los dispositivos que se encuentren conectados como esclavos. La señal de reloj es proporcionada por el dispositivo maestro, pues será el que establezca la frecuencia de comunicación del protocolo. [40]

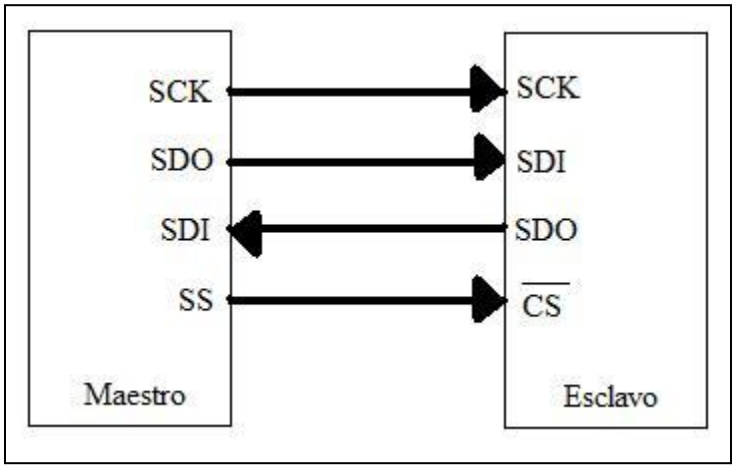


Figura 4.4. Diagrama de conexión en SPI

Un dispositivo que se comunica por el protocolo SPI puede operar de dos maneras distintas, éstas son como maestro o como esclavo. Cuando el dispositivo es maestro, éste establecerá la frecuencia de la línea de reloj a través de la línea SCK. Si por el contrario, el dispositivo es esclavo, éste tendrá como entrada de reloj la línea SCK que será establecida por el dispositivo maestro. Es importante mencionar que en esta comunicación es necesario que siempre exista un dispositivo maestro y, al menos, un dispositivo esclavo.

[11] SPI es una marca de Motorola Corporation.
[12] I²C es una marca de Philips Corporation.

El protocolo SPI permite comunicación *full-duplex*, que se hace posible gracias a las dos líneas que utiliza para la transferencia de datos, pues podría existir envío y recepción de datos de manera simultánea.

4.6.2.3.2 I^2C

En la electrónica de consumo, telecomunicaciones y electrónica industrial, existen varias similitudes entre diseños de esos tipos. Algunas similitudes se perciben en el control inteligente que se presenta en la mayoría de los casos con un microcontrolador, y en circuitos de propósito general como puertos de E/S, memorias RAM y ROM, convertidores A/D y D/A, relojes de tiempo real, etc. [41]

Para aprovechar estas similitudes para beneficio de los diseñadores y de los fabricantes, además de maximizar la eficiencia del hardware y la simplicidad de los circuitos, *Phillips Semiconductors* desarrolló un bus simple bidireccional de 2 cables para hacer eficiente el control interno de circuitos integrados. Este bus es llamado I^2C.

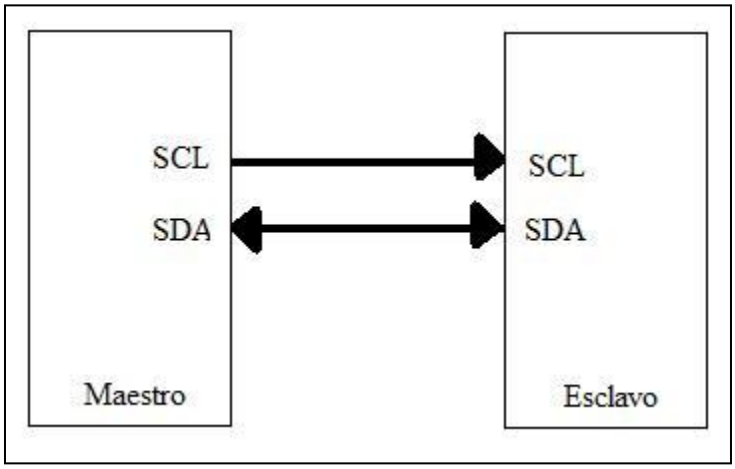


Figura 4.5. Diagrama de conexión en I^2C

La comunicación a través de un bus I^2C es un estándar mundial que actualmente está implementado en más de 1000 dispositivos de circuitería integrada. Además, el bus I^2C es utilizado en una gran variedad de arquitecturas de control, tales como: SMBus (*System Management Bus*), PMBus (*Power Management Bus*), IPMI (*Intelligent Platform Management Interface*) y en ATCA (*Advanced Telecom Computing Architecture*). [41]

Al igual que en el protocolo SPI, el protocolo I^2C requiere que en la comunicación entre dispositivos siempre exista un dispositivo considerado como maestro y al menos un dispositivo esclavo para poder establecer la comunicación. Pero éste, a diferencia de SPI, permite conectar a más de un maestro que pueda establecer la comunicación para gestionarla de manera que ese dispositivo tenga comunicación con uno y sólo un esclavo.

Aunque cabe mencionar que cuando se está realizando la comunicación, sólo se permite un dispositivo maestro. [42]

El modo de comunicación del protocolo I^2C es *half-duplex*, ya que el envío y recepción de datos no se realizan de manera simultánea, sino que se realizan en diferentes instantes de tiempo, esto se debe a que únicamente se implementa una línea para recepción y transmisión de datos, por lo que no se pueden hacer ambas acciones al mismo tiempo a través de la misma línea de comunicación. La ventaja de ello es que implementa menos hardware, una desventaja es que se hace de manera más lenta, pues si un dispositivo requiere enviar datos deberá esperar a que la línea de datos se desocupe para iniciar el envío.

Algunas de las características que presenta el bus I^2C, mencionadas en [41], son:

- Sólo se requieren dos líneas del bus; una línea de datos serial (SDA) y una línea de reloj serial (SCL).
- Cada dispositivo conectado al bus es direccionable por software por una única dirección y una relación maestro/esclavo simple.
- Incluye detección de colisión de datos y arbitrariedad de envío para prevenir la corrupción de datos si dos o más maestros inician transferencia de datos simultánea (en casos de más de un dispositivo maestro).
- Comunicación Serial
- Orientada al envío de 8 bits de datos, aunque existen dispositivos que permiten realizar comunicación de 9 bits de datos.
- Transferencia bidireccional de hasta 100 Kbits/s en *Standard-mode*, hasta 400 Kbits/s en *Fast-mode*, hasta 1Mbit/s en *Fast-mode Plus*, o hasta 3.4 Mbits/s en *High-speed mode*.
- Contiene filtro en chip, lo cual permite rechazar picos de voltaje en la línea de datos del bus para preservar la integridad de datos.
- El número de circuitos integrados que pueden ser conectados al mismo bus está limitado sólo por una capacitancia de bus máxima, regularmente de 400 pF.

Protocolo I^2C

Cada dispositivo es reconocido por una única dirección y puede operar como transmisor o receptor, dependiendo de la función del dispositivo. Además de ello, los dispositivos pueden ser considerados como maestros o esclavos cuando realizan transferencia de datos. Un maestro es el dispositivo que inicia una transferencia de datos sobre el bus y genera las señales de reloj para permitir dicha transferencia. A la vez, cualquier dispositivo direccionado es un esclavo. [41]

El protocolo especifica que mientras no exista comunicación entre dispositivos, las líneas de SDA y SCL se deben mantener en nivel de voltaje alto. Ello indica a todos los dispositivos conectados a través del bus I^2C que no hay comunicación o que el bus se encuentra libre. Para lograr este nivel de voltaje alto las líneas SDA y SCL se conectan a la línea de alimentación a través de resistencias de *pull-up*. [41]

La línea SDA encárguese encarga de transmitir los valores que corresponden a cada bit de transferencia entre los dispositivos. Además lleva la dirección del dispositivo esclavo con el que el maestro busca realizar la comunicación.

Para que el dispositivo reconozca un bit en la línea de datos, la línea de reloj SCL debe estar en alto (para lectura), es decir, que mientras la línea de SCL se encuentre en nivel de voltaje bajo la línea SDA puede cambiar de estado y cuando SCL se ponga en alto se reconocerá como un bit de dato el valor de la línea SDA en ese preciso instante.

Otro aspecto definido en el protocolo I^2C son las condiciones para iniciar la comunicación o para finalizarla. Dichas condiciones permiten puntualizar a cada dispositivo cuando el bus se libera o se ocupa y éstas deben ser establecidas por el dispositivo que actúa como maestro. Éstas se denominan condición de inicio y condición de paro. [42]

- **Condición de inicio:** Es la transición de la línea de SDA de nivel de voltaje alto a voltaje bajo mientras la línea de SCL se encuentra en voltaje alto.
- **Condición de paro:** Es la transición de la línea de SDA de nivel de voltaje bajo a alto cuando la línea de reloj SCL se encuentra en alto.

Es decir que cuando un dispositivo maestro desea comenzar la comunicación, mientras el bus está libre, primero deberá colocar la línea SCL en voltaje alto y, posteriormente, cambiar de estado la línea SDA; después de ello, establecer la frecuencia de la señal de reloj y realizar la transferencia de datos. De igual manera, cuando se desea la finalización, la señal de reloj se establece en nivel alto y se hace el cambio de estado de SDA de bajo a alto. Se mantendrá la línea de voltaje alto en ambas líneas.

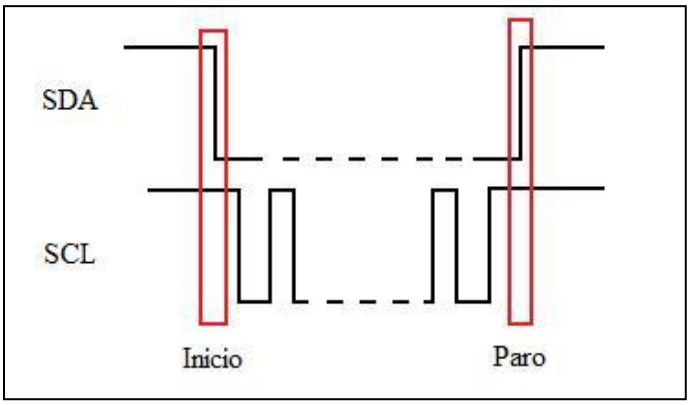


Figura 4.6. Condiciones de inicio (*Start*) y paro (*Stop*)

Existe una condición más que se denomina de inicio repetido, en la que en lugar de realizar una condición de paro para finalizar la comunicación, únicamente que requiere reestablecerla. Ésta, en términos prácticos, es idéntica a la condición de inicio, sin embargo, es importante mencionarla conceptualmente, pues de ella dependerán algunos factores del protocolo de comunicación.

El formato de la información a transmitir a través del bus I^2C es por medio de un byte. Inmediatamente después de cada byte transmitido se debe enviar un bit que indica que la transmisión del byte de datos ha finalizado, dicho bit se considera de *Acknowledge* (reconocimiento). Dicha señal de reconocimiento podrá ser de ACK (*Acknowledge*) o de NACK (*Not Acknowledge*), en nivel de voltaje alto o de bajo, respectivamente.

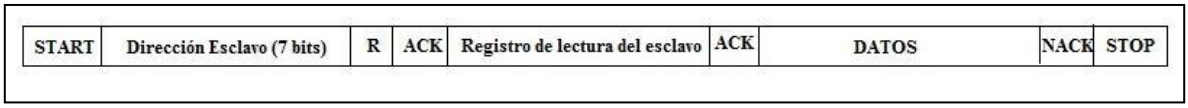


Figura 4.7. Trama de Comunicación del protocolo I2C

	Comunicación	Núm. Líneas	Vel. transmisión
SPI	Full-dúplex	4 (ó más)[13]	Hasta 10 Mbps
I^2C	Half-dúplex	2	Hasta 3.4 Mbps

Tabla 4-1. Comparación entre SPI e I2C

4.6.3 ADC (*Analog-Digital Converter*)

Un ADC (Convertidor Analógico/Digital) es aquel dispositivo que tiene como entrada una señal analógica (continua en el tiempo) y su salida es una señal digital (discreta en el tiempo) representada en unos y ceros, de manera eléctrica.

Las fuentes primarias de señales analógicas son generalmente componentes mecánicos, de los cuales sus salidas son convertidas a señales eléctricas. Por ejemplo, los termoacopladores son utilizados para medir temperaturas, que son convertidas a salidas de señales eléctricas. La presión de un gas puede ser medida por transductores de presión cuyas señales de salida son analógicas. Un ADC será capaz de medir esas señales físicas para convertirlas en señales eléctricas que representen estados discretos y niveles de voltajes alto y bajo. [43]

Para realizar la conversión de una señal analógica a una digital se puede hacer de diferentes maneras. Dos de las técnicas más comunes para realizarlo son el método de la rampa de

[13] El protocolo SPI ocupa 4 líneas: SDO, SDI, SCK y SS, aunque requerirá mayor cantidad de líneas cuando se presenten más de 2 esclavos conectados.

voltaje y el método de aproximación sucesiva. En ambos casos es necesario hablar de un comparador, que es muy utilizado para la conversión. [43]

Un comparador es simplemente un amplificador diferencial cuya salida es una función de la diferencia entre el voltaje de dos señales aplicadas como entrada. Una de esas dos señales de entrada es la señal de referencia. La otra señal de entrada será la que se desea digitalizar. [43]

4.6.4 PWM (*Pulse-Width Modulation*)

La técnica de PWM (Modulación por Ancho de Pulso) es una técnica que se utiliza para generar señales cuadradas con una determinada frecuencia. Una señal de PWM es un tren de pulsos con una frecuencia fija y un periodo de voltaje alto variable.

El tren de pulsos puede estar caracterizado por su ciclo de trabajo, que será el porcentaje en que la señal estará activa y su periodo. [32]

El uso de PWM permite un voltaje promedio de DC variable que puede controlar un motor de DC, por ejemplo. El voltaje de DC promedio es logrado por la variación de encendido/apagado en un ciclo. Por ejemplo, si tenemos una señal con frecuencia de 10 Hz (que tarda 100 ms en completar un ciclo) y un ciclo de trabajo de 70%, quiere decir que la señal de voltaje se mantendrá en alto durante 70 ms y en los 30 ms restantes la señal se encontrará en voltaje bajo. Si el voltaje de la señal es de 5 V, el voltaje promedio será de 3.5 V, ya que $5\,V \cdot 70\% = 3.5\,V$. Sin embargo, si el ciclo de trabajo fuese de 30%, la señal se mantendrá en alto durante 30 ms, y los 70 ms restantes en bajo. El voltaje promedio será de 1.5 V [44]

4.7 Interrupciones

Una interrupción, o también conocida como solicitud de interrupción, es un evento interno o externo que provoca que un procesador interrumpa la ejecución de un programa para atender dicho evento, ejecutando otro programa. Regularmente, cuando el procesador ha terminado de ejecutar el programa atendido cuando ocurrió la interrupción, el procesador recupera la dirección de memoria de la instrucción en donde se interrumpió el primer programa y continúa su ejecución.

Las interrupciones, en un sistema basado en microprocesador, se utilizan cuando se conectan dispositivos que proporcionan o requieren datos a velocidades de transferencia muy bajas y que éstos sean atendidos por el CPU.

Por ejemplo, un microprocesador requiere procesar la entrada de un teclado (se escribió un carácter). Una manera de procesarla es que cada cierta cantidad de tiempo se haga una comparación en la entrada para saber si existen datos disponibles a la entrada. Si una

persona presiona una tecla por segundo, esto quiere decir que el procesador tendrá una espera de 1 segundo para poder procesar la entrada del teclado y continuar con otras tareas, y volverá a hacer la comparación en un momento dado. Lo cual es un desperdicio de tiempo.

Para evitar el problema anterior, se diseñó el *procesamiento de interrupciones*. Esta manera de manejar las acciones críticas (interrupciones) permite que mientras no haya una acción crítica, el procesador se encontrará realizando alguna tarea. Es decir que cuando se presente una acción crítica, el microprocesador detendrá la acción actual para intervenir la interrupción y solventarla. De esta manera, en el ejemplo del teclado, mientras el operador del teclado se encuentra pensando qué tecla pulsará la próxima vez, el microprocesador estará ocupado realizando alguna otra tarea. Cuando se haga la pulsación del teclado, en cualquier momento, el microprocesador detendrá el programa que se encontraba ejecutando y atenderá la interrupción generada por la pulsación del teclado. [45]

Las interrupciones son eventos asíncronos al programa que está siendo ejecutado, lo que significa que pueden ocurrir en cualquier momento. Difícilmente se puede predecir la instrucción en la que está siendo interrumpido el procesador. No obstante, cuando una interrupción se presenta mientras el procesador está ejecutando una instrucción, antes de atenderse esa interrupción, el procesador deberá finalizar la ejecución de la instrucción. [32]

Para los microcontroladores, el uso de interrupciones es el mismo, ya que cuentan con un procesador o CPU.

Para almacenar el valor del registro PC y guardar la última dirección de la instrucción en la que el procesador detuvo el programa antes de atender la interrupción, se implementa una pila o stack. El conjunto de instrucciones que se ejecutarán cuando una interrupción ocurra se le denomina *rutina de interrupción*. Aunque el procesador no sabe directamente en qué parte de la memoria de programa se encuentra esa rutina, es necesario indicarlo en la programación. Existen dos maneras de hacer esto, por solicitud de interrupciones fijas o por solicitud de interrupciones vectorizadas. [32]

En una solicitud de interrupción fija, la dirección de la rutina de interrupciones está determinada en el sistema. Es decir, que cuando ocurra una interrupción, de cualquier tipo, se le asignará al PC la dirección de la primera instrucción de la rutina de interrupción. Sin importar de qué interrupción se trate, siempre se asignará la misma dirección. Esta es la técnica más utilizada en microcontroladores debido a que es muy simple y no requiere demasiado procesamiento. [32]

En una solicitud de interrupción vectorizada, el procesador obtiene la dirección de la rutina de interrupción. Esta dirección tiene como información un vector de interrupciones, el cual

indica que la rutina de interrupción se encuentra en cualquier lugar de la memoria de programa. Aquí, la rutina de interrupción puede tener una estructura elaborada, de manera que la estructura más simple será una dirección de memoria; o también se puede tratar de un vector o tabla de direcciones localizadas en memoria, en las que se encuentra la dirección de la rutina. Las interrupciones vectorizadas son más comunes en microprocesadores. [32]

4.8 Microcontroladores PIC®

En el mercado existen varios fabricantes de dispositivos microcontroladores.

Los microcontroladores PIC®, fabricados por la empresa Microchip Technology Inc., son dispositivos con un desempeño que va desde los 8, 16 y 32 bits (haciendo referencia al bus de datos). Son muy utilizados para implementar en sistemas embebidos que requieren control de otros dispositivos, como son: LEDs, transductores, motores, sensores, memorias, etc.

Los microcontroladores PIC® corresponden a la arquitectura Harvard, con memorias de programa y datos separadas, además de buses de instrucciones y datos para cada memoria. Esta arquitectura le permite al microcontrolador tener concurrencia de instrucciones de programa y permite paralelismo (*pipeline*). En la mayoría de microcontroladores, la memoria de programa es mayor a la de datos, y en los PIC no es la excepción. Además, el bus de instrucciones suele estar organizado en palabras de 12, 14 ó 16 bits, mientras que la memoria de datos se compone por registros de 8 bits. Algunos PIC cuentan con cierta cantidad de memoria EEPROM para almacenamiento no volátil de datos.

Otra parte importante es que los microcontroladores PIC® tienen arquitectura RISC, pues cuentan con un número pequeño de instrucciones que consta de entre 33 y 83 instrucciones. Todas las instrucciones son del mismo tamaño. El PIC cuenta con un registro de trabajo (registro W) que funciona como registro acumulador para almacenar operandos de las instrucciones, además de que puede manejar registros de la memoria de datos.

Otra característica relevante de los PIC es la manera en la que se implementa la pila. La pila no se encuentra en la memoria de datos, sino que está implementada de manera independiente y tiene un número limitado de localidades según sea el modelo de PIC, aunque cabe mencionar que éstos no implementan un apuntador a la pila (SP, o Stack Pointer).

Todos los elementos antes mencionados en este trabajo están integrados dentro del microcontrolador PIC.

4.8.1 Clases

Existen diferentes clases de microcontroladores PIC basadas en sus arquitecturas internas, especialmente en su bus de datos. Éstos son los que tienen 8 bits en bus, los que poseen 16 bits y los que utilizan 32 bits.

Las diferentes clases de microcontroladores PIC tienen variantes, además del bus de datos, en la cantidad de periféricos que son capaces de manejar. Cada clase también está agrupada por el tamaño de sus microinstrucciones, las familias de PIC son las siguientes: [39]

- *Base-Line*: Con una longitud de palabra de instrucción de 12 bits.
- *Mid-Range*: Con una longitud de palabra de instrucción de 14 bits.
- *High-End*: Con una longitud de palabra de instrucción de 16 bits.

El microcontrolador a utilizar será el PIC18F4520 que se encuentra en los microcontroladores PIC® de clase baja y en la subclase *High-End*.

El PIC18F4520 está compuesto de 20 fuentes de interrupción.

4.9 Sensores

Los sensores son dispositivos que son capaces de detectar magnitudes, ya sean de tipo físico o químico, en función del tiempo y que pueden transmitirla de manera eléctrica para su lectura por otros dispositivos. Los sensores se pueden definir como dispositivos transductores, es decir, que reciben un dato analógico de cualquier tipo, del entorno en el que se encuentran, y lo convierten en una señal de un tipo diferente.

Existen sensores para diferentes usos, los hay de temperatura, presión, luz infrarroja, ultrasonido, aceleración, proximidad, humedad, etc.

Los sensores que se utilizan en la actualidad convierten una señal de tipo física para convertirla en una señal eléctrica que puede ser procesada por otro tipo de dispositivo, como puede ser un microcontrolador. Por ejemplo, los sensores infrarrojos son usados para determinar la presencia de un objeto a una distancia determinada, si éstos son analógicos, dependiendo de la distancia a la que se encuentre el objeto será el voltaje de salida del sensor; en cambio, si los sensores son digitales, la presencia del objeto indica que existirá voltaje alto a la salida, si no es así, habrá voltaje bajo.

El uso de sensores permite el manejo de señales en tiempo real, ya que el funcionamiento dependerá de las circunstancias actuales del entorno de trabajo de los mismos. [46]

Los sensores que se pretenden utilizar en el trabajo son los siguientes:

- Infrarrojos, para detectar la presencia de muros dentro del laberinto. Además, es el tipo de sensores que se utilizan en los encoders para detectar el giro de los motores.
- Magnetómetro, para uso de una brújula que indicará la orientación del robot.

4.10 Motores

Los motores son máquinas, generalmente térmicas, de combustión, eléctricas; que convierten el tipo de energía correpondiente en energía mecánica rotacional. En este trabajo, el enfoque se hará únicamente a los motores eléctricos.

Un motor eléctrico es una máquina (con eficiencia del 90%) que convierte la energía eléctrica en energía mecánica.

Los motores están en todo tipo de aparatos, en los automóviles, en los electrodomésticos, en los robots, etc. En la mayoría de los dispositivos donde hay movimiento, éste es producido por un motor eléctrico, que puede ser de AC (*Alternating Current*) o DC (*Direct Current*). [47]

Para fines prácticos, este trabajo tratará los motores de corriente directa (DC). Un motor eléctrico de DC simple consta de 6 partes [47]:

- Armadura o rotor
- Conmutador
- Cepillos
- Eje
- Campo magnético
- Suministro de DC de algún tipo

Para comprender el funcionamiento de los motores eléctricos, es necesario entender algunos conceptos básicos. Los motores eléctricos basan su funcionamiento en la relación que existe entre la electricidad y el magnetismo, aprovechando dichas propiedades para producir energía mecánica.

4.10.1 El magneto

El uso del magneto es el principio fundamental del magnetismo. Un magneto es una sustancia que atrae el hierro y produce un campo magnético. Hay 2 tipos de magnetos, permanentes y temporales. Los permanentes son fenómenos naturales. Por ejemplo, la magnetita es un material que es atraído al hierro. El magneto se considera como dipolo por naturaleza, ya que está dividido en dos polos diferentes, polo norte y polo sur. Empero, si un magneto tiene 2 polos, no quiere decir que éstos se puedan separar, puesto que si se separan cada una de las piezas resultantes mantendrán las mismas propiedades que el magneto original. [48]

Los polos norte y sur de un magneto permanente están unidos por un campo invisible de líneas magnéticas de flujo que envuelve al magneto entero. El patrón de las líneas de flujo puede ser visto poniendo un magneto permanente bajo un vidrio y colocar limadura de hierro sobre el vidrio. [48]

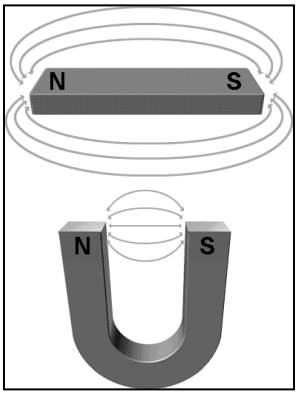


Figura 4.8. Flujo del campo magnético en magnetos permanentes [49]

Si una pieza de hierro suave es colocada en el campo (permanente) magnético, ésta toma las propiedades magnéticas del magneto permanente. No es necesario que toque el magneto. El campo magnético del magneto permanente induce un campo magnético sobre la pieza de hierro llamado **inducción**. La pieza de hierro, entonces, se vuelve un magneto temporal y permanece así mientras se encuentre al alcance del campo magnético. [48]

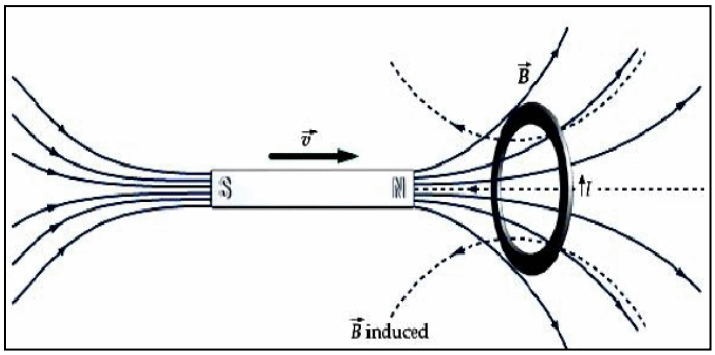


Figura 4.9. Inducción de un campo magnético sobre un cuerpo metálico (circular) [50]

Si dos magnetos se juntan, se pueden atraer o repeler el uno al otro, dependiendo de su orientación. La regla básica del magnetismo es "polos iguales se repelen y polos diferentes se atraen". Esta parte es la base esencial en el funcionamiento de los motores. [48]

Los magnetos temporales pueden ser creados también por el flujo de corriente a través de un conductor. Una ley básica de la física es que un campo magnético es creado alrededor de un conductor cuando una corriente eléctrica fluye a través de él. Los magnetos temporales no retienen magnetismo.

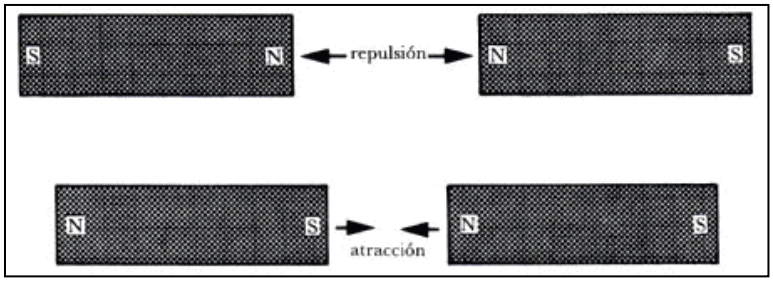


Figura 4.10. Efectos de atracción y repulsión en magnetismo [51]

En Eléctrica, cuando se induce un campo magnético (temporal) alrededor de un conductor a partir del flujo de corriente, a éste se le llama **electromagneto** y se usa en motores eléctricos. La fuerza del campo puede incrementarse aumentando la cantidad de corriente o voltaje. [48]

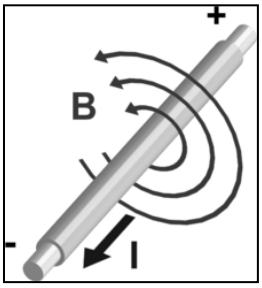


Figura 4.11. Creación de un campo magnético *B* a través del flujo de corriente *I* en un electroimán [52]

4.10.2 Tipos de motores

Existen diferentes tipos de motores. De acuerdo a su aplicación, cada motor puede tener distintas capacidades. Una característica fundamental en ellos es el "par motor", o *torque*, que es la fuerza con la que gira un motor. Otra característica es el número de revoluciones por minuto (rpm) que, en otras palabras, es la rapidez de giro del motor. La mayoría de los motores sacrifican fuerza a cambio de mayor rapidez.

Los motores más comunes en electrónica son los motores de DC, los motores a pasos y los servomotores.

4.10.2.1 Motor de DC

Un motor de corriente directa cuenta básicamente con dos componentes, embobinado de campo y el embobinado de armadura. Estos dos embobinados tienen sus ejes montados en una cuadratura espacial eléctrica y ambos son alimentados con potencia desde las fuentes de corriente directa. Un conmutador en el rotor está conectado a los conductores de la armadura rotacional. Esto actúa como un cambiador o rectificador de frecuencia mecánica

para mantener la corriente del circuito de la armadura de manera unidireccional a través de los cepillos en todas las velocidades. [53]

4.10.2.2 Stepper Motor (Motor a pasos)

Los motores a pasos son una clase de dispositivos electromecánicos usados para producir movimiento discreto o no continuo.

El término "a pasos" significa que el proceso de conversión de energía es realizado en pasos discretos o también llamados incrementos. Por esta razón se consideran dispositivos de movimiento incremental, los cuales incluyen actuadores incrementales y otros dispositivos de control digital usados en control de movimiento.

Existen varios tipos de motores a pasos, como son, hidráulicos, neumáticos y eléctricos. Típicamente, los motores a pasos hidráulicos y neumáticos tienen mucho más par motor que los eléctricos utilizados en aplicaciones de robótica.

Sin embargo, en aplicaciones donde se requiere posición, control de velocidad o aceleración precisas y cuando es necesario el control de varios pasos pequeños, los motores a pasos eléctricos tienen muchas más ventajas sobre los hidráulicos o neumáticos. Pues son muy compatibles con sistemas de control digital.

La entrada al motor a pasos es digital, con un pulso eléctrico o una serie de pulsos discretos. Si el tren de pulsos de entrada es continuo y aplicado a intervalos de tiempo apropiados, el motor a pasos puede ser operado a velocidad constante y puede tener el comportamiento de un motor de DC. [54]

Algunas de las aplicaciones que tienen estos motores son los dispositivos de precisión, como pueden ser discos duros o unidades de disquete. También se ocupan en impresoras.

4.10.2.3 Servomotor

Un servomotor es un tipo de motor eléctrico que se utiliza en aplicaciones donde se requiere una carga muy grande. Los motores de DC y a pasos no tienen las mismas capacidades de par motor que los servomotores. Los servomotores se utilizan, sobre todo en la industria. La mayoría de estas aplicaciones, además requieren control de velocidad y de posición precisos.

Algunas de las características de los servomotores son las siguientes:

- Produce alto par motor a cualquier velocidad.
- Son capaces de mantener una posición estática.
- A velocidades bajas o cargas pequeñas, no se sobrecalientan.

- Son capaces de revertir su dirección de manera rápida.
- Acelera y desacelera rápidamente, para llegar a una posición o velocidad determinada.

Otra característica principal de los servomotores es que las líneas de alimentación del servomotor no van directamente a los contactos del mismo, sino que van hacia un circuito de control que se encuentra en el motor.

4.10.3 Diferencia entre motores de DC y AC

La principal diferencia entre motores de DC y AC será la aplicación que se le pueda dar. El uso de motores de DC se ha ido incrementando desde que se implementaron los dispositivos de estado sólido, ya que permitieron crear sistemas de control y convertidores de potencia más pequeños y baratos y que se pueden utilizar de una manera simple con motores de este tipo. Mientras que los motores de AC, los sistemas de control y convertidores de potencia son más caros y complejos para utilizar.

Otra razón por la que se han dejado de utilizar motores de AC es por la necesidad de desarrollar nuevas tecnologías para diseños de métodos y sistemas de control, cuya aplicación es más simple en DC.

A continuación, se enlistan algunas de las ventajas y desventajas de los motores de AC y los motores de DC:

DC drive	AC drive
Ventajas	
• Tecnología bien establecida • Convertidores de potencia simples y baratos. • Sistemas de control simple • Sin problemas de velocidad cero • Rangos de velocidad amplios • Respuesta rápida	• Confiabilidad del motor • Costo del motor (tamaño/peso) • No son sensibles ambientalmente • Buen factor de potencia en la línea de AC (en control PWM)
Desventajas	
• Costo del motor (tamaño/peso) • Sensibles al ambiente • Mantenimiento del motor • Factor de potencia pobre	• Convertidores de potencia complejos • Sistemas de control complejos • Control de velocidad cero (en lazo abierto) • Tecnología en desarrollo

Tabla 4-2. Ventajas y desventajas del drive para motores de DC y AC [53]

4.10.4 Consideraciones para motores de DC

Algunas de las consideraciones que se suelen tomar en cuenta cuando estamos trabajando con motores de Corriente Directa, de acuerdo a la empresa MICROMO en su página web [55], son las siguientes:

a) Ruido audible.

En algunas aplicaciones, el ruido audible puede ser una consideración importante, pues puede afectar el funcionamiento de ciertos artefactos. Por ejemplo, en aplicaciones de Medicina, para pacientes sensibles al ruido. Las buenas prácticas de diseño requieren que se reduzca el ruido de los motores al mínimo posible.

b) Interferencia Electromagnética (EMI).

Los motores de DC son una fuente eléctrica y, por consiguiente, de campos magnéticos. La interferencia electromagnética se produce en las terminales del motor y en el embobinado, y puede causar problemas con otros componentes que se encuentren alrededor. También es posible que los picos de voltaje provocados por las interferencias se acoplen a alguna línea de datos o salida de un encoder, por ejemplo, en caso de contar con ellos, debido a que el encoder trabaja junto al motor. El resultado de esto puede ser una lectura de datos errónea.

c) Vida de servicio del motor

La vida de servicio del motor se refiere a la vida útil que tendrá el motor de acuerdo a su funcionamiento cotidiano. También se le puede considerar como su tiempo de vida y éste dependerá de varios factores a tomar en cuenta

Consideraciones ambientales: Las condiciones ambientales pueden tener un efecto profundo en la vida de servicio del motor. Un ejemplo es el secado pronto y el desgaste de cepillos basados en grafito en un vacío o en un ambiente muy seco. Condiciones muy calientes y secas también apresuran la descomposición del rodamiento y los lubricantes del conmutador. La temperatura ambiente tiene un efecto acumulativo en la temperatura operacional del motor y puede reducir su desempeño. El enfriamiento externo por contacto, aire o aire forzado puede producir ganancia significante en el desempeño del motor. Por el contrario, condiciones muy frías incrementan la viscosidad de los lubricantes y provocan que el motor corra a una corriente mayor. [55]

4.11 Control de motores

Un sistema de control se describe como un grupo de componentes que se encuentran organizados de tal manera que la operación del sistema a controlar se mantenga en un nivel

deseado. Se compone de 3 elementos básicos: una función de entrada *r(t)*, una función de salida *c(t)* y una función de transferencia *h(t) (Ver Figura 4.12).* Nótese que dichas funciones se encuentran en el dominio del tiempo *t*, aunque se suelen manejar en el dominio de la frecuencia *s*.

Los sistemas de control se pueden clasificar en sistemas de control analógico y sistemas de control digital. Los sistemas de control analógico implementan señales analógicas y la adquisición de esas señales se realiza a partir de métodos de procesamiento de señales analógicas. Los sistemas de control de señales digitales implementan adquisición de datos a través de procesamiento de señales digitales.

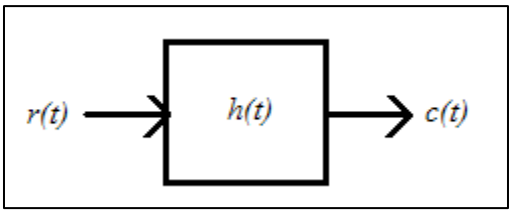


Figura 4.12. Sistema de control básico

Los sistemas de control también están clasificados de acuerdo a la presencia de retroalimentación. Es así que un sistema de control se llamará de lazo abierto cuando no emplee retroalimentación respecto a la salida, y será de lazo cerrado cuando el proceso se monitorea constantemente a través de un lazo de retroalimentación.

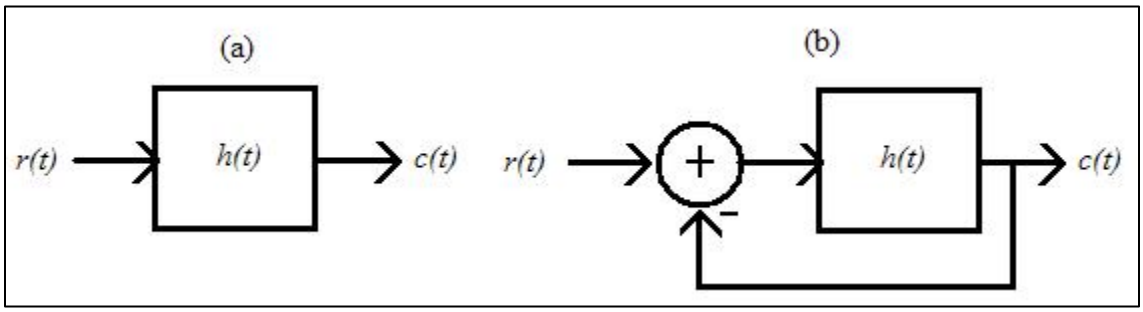


Figura 4.13. Sistemas de Control: (a) con lazo abierto y (b) con lazo cerrado.

Lo primordial que se busca en un sistema de control es obtener una determinada salida con respecto a cierta entrada. Lo que supone que cuando se introduce una señal a un sistema, ya sea analógica o digital, siempre existirá una salida para cada entrada.

Existe una serie de controles básicos para motores que se ocupan en muchos sistemas. El mantener el control de un motor permitirá manipularlo de modo que podamos operar su funcionamiento.

4.11.1 Control ON/OFF

El control ON/OFF (o de Encendido/Apagado), también llamado, por algunos autores como un control de doble cable, es un tipo de control para motores que únicamente requiere de dos cables para controlar un switch y un controlador para la bobina. Este sistema de control se considera de lazo abierto, pues no requiere retroalimentar el sistema con la magnitud de salida.

La ventaja principal del control ON/OFF es su simplicidad y a que no requiere gran cantidad de componentes, además de que su operación es sencilla, por medio de una línea de control. Una de las desventajas es que mientras el motor se opera, cada vez que se enciende, se presenta gran demanda de corriente por la carga del actuador, por lo que la potencia es alta y el gasto de energía también lo es. [56]

4.11.2 Control START/STOP Seal

Otro control, muy similar al ON/OFF, es el llamado START/STOP Seal (o, sello de Inicio/Paro), en el que se requieren 3 líneas de control, para controlar una estación de Inicio/Paro, y el controlador para la bobina del motor. El circuito utiliza dos botones (push-button), uno para indicar el inicio y el otro para el paro. Esto es, cuando el botón de inicio es presionado, el motor es energizado y estará operando. Para lograr detenerlo se deberá cortar la energía al motor apagándolo o presionando el botón de paro. En caso de apagarlo, para que se reinicie la operación del motor se deberá presionar el botón de inicio nuevamente.

Este tipo de control, al igual que el control ON/OFF, es de lazo abierto, no requerirá retroalimentar con su valor a la salida del sistema. [56]

El diseño de módulos PWM para el control de motores dio lugar a la eliminación de otros componentes que se utilizaban para los drivers de motores. Eliminar circuitos de conmutación auxiliares resultó en reducciones de costos. Además, las estrategias de conmutación implementadas para el control de voltaje y corriente fueron factor principal para el diseño de inversores de PWM, pues el uso de velocidades de conmutación más rápidas y de frecuencias de conmutación más altas, permitió el decremento de distorsión armónica y disminuyó el tamaño de los componentes del filtro.

4.11.3 Control PID

El control PID suele ser un sistema de control muy aplicado en la industria debido a su diseño simple y a que resulta ser bastante eficiente. Se compone de 3 partes: Proporcional (P), Integral (I) y Derivativa o Diferencial (D). [57]

Es un sistema de lazo cerrado, lo que quiere decir que retroalimenta la salida del sistema con la entrada del mismo para poder generar un valor de salida mucho más preciso al deseado. Se basa en la corrección de errores, por ello la retroalimentación. Esto es, el valor obtenido a la salida se suma al de entrada para obtener una diferencia o error *e(t)*, que será procesado y reducido al mínimo de manera que en el próximo instante de tiempo la diferencia se aproxime a cero. Cuanto más cerca de cero se encuentre, el sistema será más estable.

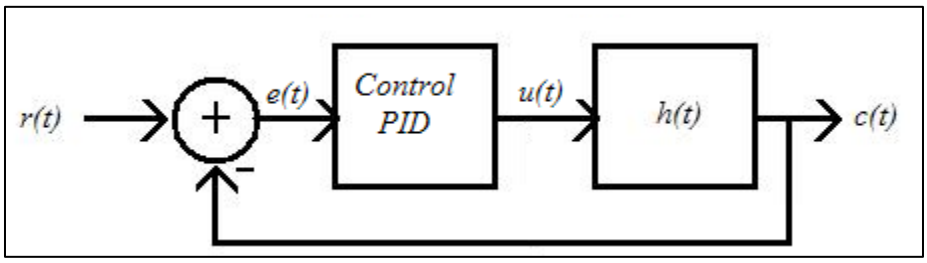


Figura 4.14. Estructura de un sistema de control PID.

En cuanto a las diferentes partes del control PID, la parte proporcional permite establecer un límite de desempeño del controlador, la parte diferencial proporciona mejoras a altas frecuencias y la parte integral incrementa el desempeño en frecuencias bajas. [58]

El control PID consta de variantes en las que cada parte del controlador puede o no estar presente, lo que puede tener ventajas o desventajas según el sistema que estemos tratando de controlar.

4.12 Familia de controladores PID

La familia de los controladores PID es la mezcla de los diferentes componentes de un control PID. Es así que la combinación de los controles proporcional, integral y diferencial permiten crear otros tipos de controles. Dependiendo del control utilizado, pueden existir ventajas o desventajas a comparación de utilizar otro.

4.12.1 Controlador Proporcional (P)

El control P es el controlador más básico de la familia PID. Sólo está compuesto de la parte proporcional, lo que implica que la parte integral y la parte diferencial son iguales a cero. Se utiliza para mantener un control más justo en cuanto a errores producidos en el sistema. En este tipo de control, el sistema tiene una respuesta lineal, la cual dependiendo de la entrada, la salida será directamente proporcional a ella. La entrada del controlador será un error *e(t)*, resultante de la diferencia de la salida con la entrada obtenida en un instante de tiempo. La salida del controlador *u(t)* será proporcional a ese error. Es decir:

$$u(t) = K_P \cdot e(t)$$

Donde K_p es la ganancia proporcional.

Una desventaja de este tipo de control es que en la mayoría de los casos existirá un error.

4.12.2 Controlador Integral (I)

El control I sólo se compone de la parte integral, lo que significa que la parte proporcional y la parte diferencial son iguales a cero.

En este control, la salida *u(t)* es proporcional al error acumulado en cierto instante de tiempo. De esta manera, el error acumulado se obtiene de la integral definida (de un tiempo *0* a un tiempo *t*) de la función error *e(t)*.

$$u(t) = K_I \int_0^t e(\tau)\, d\tau$$

Para este caso, cuando *e(t)=0*, la salida del controlador *u(t)=0*. La constante K_I es una constante proporcional y en particular, para este caso, $K_I = \frac{1}{T_i}$; donde T_i es el tiempo de integración.

4.12.3 Controlador Proporcional-Integral (PI)

El control PI se compone de la parte proporcional y de la parte integral. En éste, la parte diferencial es igual a cero. Se expresa de la siguiente manera:

$$u(t) = K_P \cdot e(t) + K_I \int_0^t e(\tau)\, d\tau$$

Para este caso, la constante K_I cambia respecto al control I, puesto que se considera la constante de proporción dada por K_P. Por lo tanto, $K_I = \frac{K_P}{T_i}$.

4.12.4 Controlador Proporcional-Diferencial (PD)

El control PD se compone de la parte proporcional y de la parte diferencial. La parte integral es igual a cero.

$$u(t) = K_P \cdot e(t) + K_D \frac{d\, e(t)}{dt}$$

K_D es una constante de proporción derivativa y es igual a $K_D = K_P \cdot T_d$, donde T_d es el intervalo de tiempo derivativo.

4.12.5 Controlador Proporcional-Integral-Diferencial (PID)

El control PID es el más completo e implementa las 3 acciones (proporcional, integral y diferencial), además de tener las ventajas de las 3.

La salida de un control PID está dada por la siguiente función:

$$u(t) = K_P \cdot e(t) + K_I \int_0^t e(\tau)\, d\tau + K_D \frac{d\, e(t)}{dt}$$

Es decir, la suma de las 3 acciones conjuntas.

Aparentemente, implementar un sistema de control PID en cualquier tipo de controlador podría ser suficiente. Sin embargo, se ha llegado a demostrar que el control PID tiene ciertas deficiencias y no es aplicable a todos los sistemas. Tampoco es conveniente aplicarlo en los que sí es posible implementarlo sin hacer un análisis previo, pues en ocasiones es suficiente con algún otro controlador que no involucre las 3 acciones. Algunos controladores industriales únicamente involucran PI, pues es apropiado para procesos de primer orden. [57]

5 Diseño e implementación del algoritmo *Flood-Fill*

5.1 Explicación del algoritmo

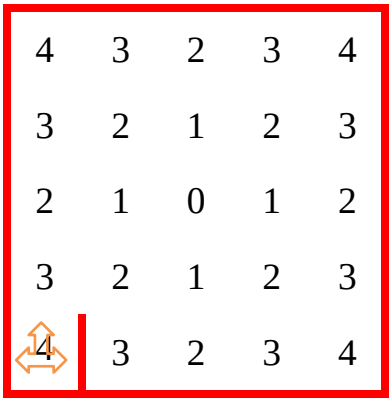
En este trabajo se utilizará el algoritmo *Flood-Fill*, o de Inundación, para resolver un laberinto de paredes. Este algoritmo es una implementación de una búsqueda informada, ya que se utilizan funciones heurísticas en cada celda del laberinto para saber la distancia a la que el robot se encuentra de la solución, que está situada en el centro del laberinto, y así poder llegar a ella.

El algoritmo se basa en dividir el laberinto en celdas, cada una con un valor heurístico. Los valores se calculan a partir de la solución del laberinto. Es decir, la solución o celda objetivo del laberinto tendrá un valor de 0, lo que quiere decir que se requieren 0 pasos para llegar a ella. Posteriormente, las celdas adyacentes al objetivo, ya sean de manera vertical u horizontal, tendrán un valor de 1 (primer nivel); las celdas adyacentes a las del primer nivel tendrán un valor de 2 (segundo nivel); las celdas adyacentes a éstas últimas tendrán valor de 3 (tercer nivel), y así sucesivamente. Sin embargo, si existe un muro entre dos celdas, éstas no se considerarán adyacentes visibles. Cabe mencionar que como condiciones iniciales se tienen las dimensiones del laberinto y la posición de la celda objetivo.

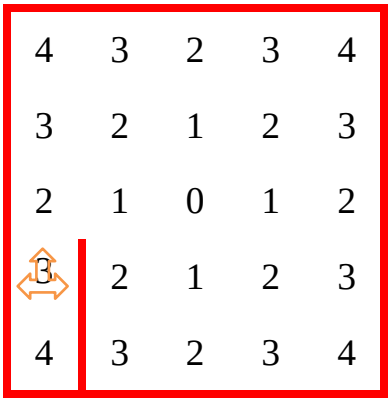
El término de inundación se le da a este algoritmo precisamente porque se encarga de "inundar" el laberinto con los valores, como si se vertiera agua desde la celda solución y se fuera desplazando a través del laberinto por diferentes caminos, pasando por diferentes niveles.

De principio, el robot no conoce el mapa del laberinto, por lo que tendrá que navegar a través de él utilizando los actuadores (motores) y usando los sensores para detectar muros. El avance del robot será celda por celda y los movimientos son verticales u horizontales.

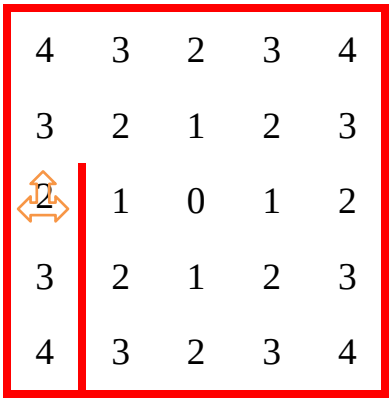
A continuación, se muestra una serie de imágenes en las que puede observarse el comportamiento del algoritmo *Flood-Fill*. Para fines prácticos, en el ejemplo habrá un laberinto de orden *5x5*.



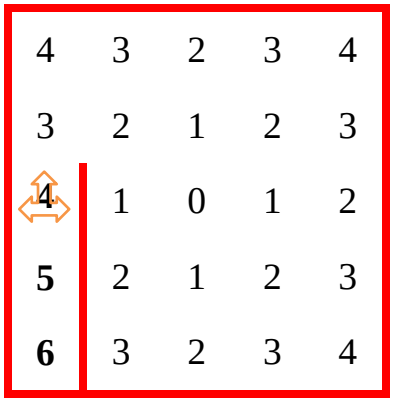
1.- El robot comienza en la esquina inferior izquierda. De acuerdo a las reglas de la competencia, la primera celda sólo puede tener un camino abierto en orientación al norte.



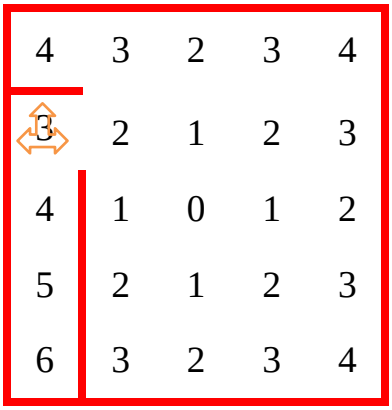
2.- El robot avanza a la siguiente celda. Revisa si existe una celda adyacente con menor valor y que no haya muro entre ellas. Sí lo hay y es al norte.



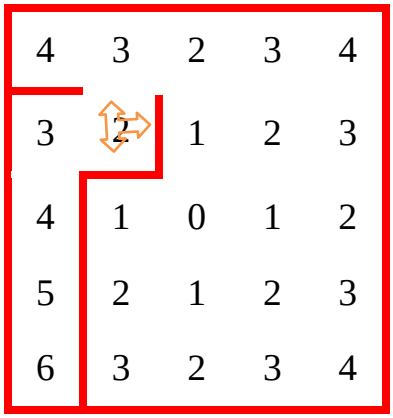
3.- El robot ha llegado a una celda que no tiene una adyacente menor que sea visible. Por lo tanto, se recalculan los valores con *Flood-Fill*, ahora con mayor información del laberinto.



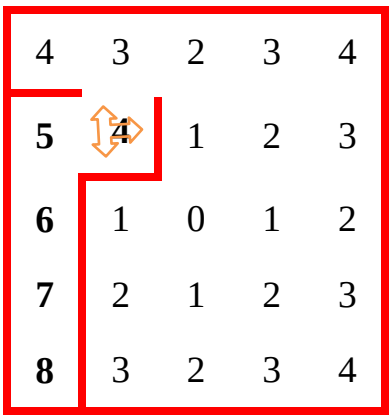
4.- Con la inundación se actualizó el valor de la celda actual y ahora ya es posible hacer el siguiente movimiento.



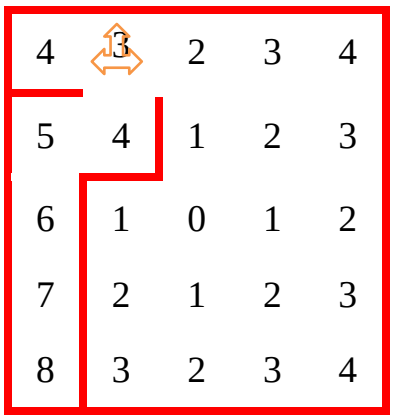
5.- El robot continúa avanzando. Ahora la celda siguiente está a la derecha, así que procede a girar a la derecha y después avanzar.



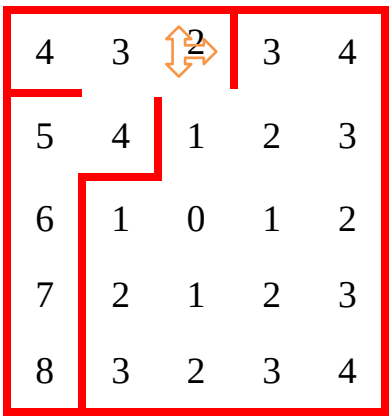
6.- Ya no es posible avanzar, pues se interponen los muros a las celdas con menor valor. Por lo que es necesario recalcular los valores con *Flood-Fill*.



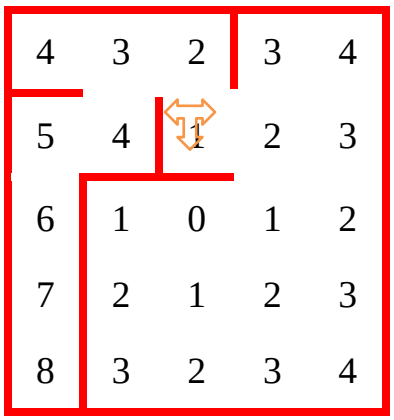
7.- Nuevamente, ya que se recalcularon los valores, es posible hacer el siguiente movimiento. Giro a la izquierda y avanza.



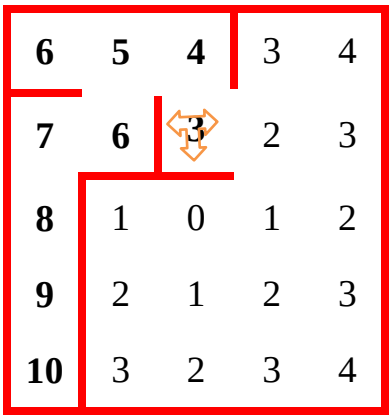
8.- El siguiente movimiento es girar a la derecha y avanzar.



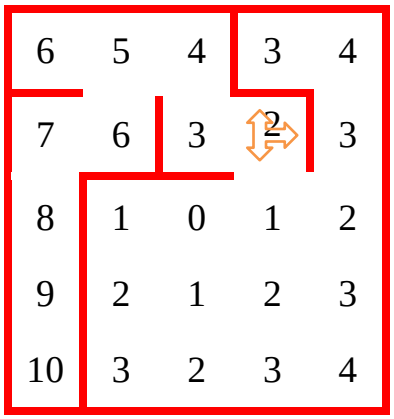
9.- El movimiento siguiente será un giro a la derecha y avanzar.



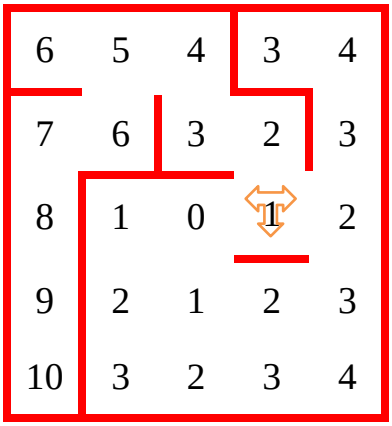
10.- Una vez más no puede elegir a dónde ir, así que se resuelve el problema aplicando *Flood-Fill*



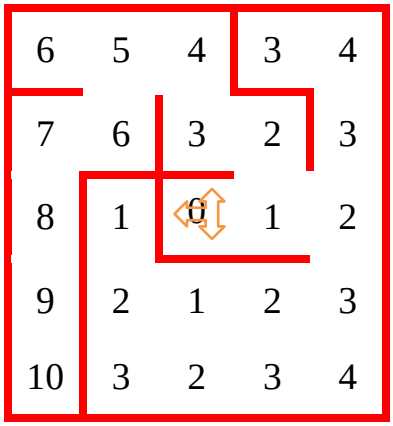
11.- Ahora puede avanzar. Giro a la izquierda y avanza.



12.- Continúa avanzando, giro a la derecha y avanza.



13.- El robot está a un paso de llegar a la solución, siguiente movimiento: giro a la derecha y avanza.



14.- Ha llegado a la solución

5.2 Software

La parte de Ingeniería de Software estará basada en la aplicación de la Ingeniería de Software Orientada a Agentes. Como se mencionó en la *Sección 2.6* del presente trabajo, existen metodologías para el desarrollo de sistemas orientados a agentes. Se explicaron las bases de algunas metodologías, incluso se hicieron algunas comparaciones entre las mencionadas.

Para el desarrollo del sistema del robot MicroMouse con ayuda de las metodologías AOSE, he decidido basarme en la metodología *Tropos* (*Sección 2.6.1*). Las ventajas importantes de esta metodología son:

- El ciclo de vida es iterativo e incremental, pues permite cambios constantes en cualquiera de sus etapas sin que esto afecte al sistema por completo.
- La perspectiva de desarrollo del ciclo de vida de la metodología es *Top-Down*, siguiendo la secuencia de etapas de manera serial y en un orden común.
- No está basada completamente en alguna metodología orientada a objetos ya que se basa en el framework *i**, lo que hace posible modelar el sistema de agentes de manera radical. Permite que no haya confusión en el modelado.
- Las etapas de análisis y diseño son lo bastante completas para realizar un modelado de sistema conveniente.
- Cuenta con una herramienta CASE para el apoyo del modelado de la metodología en todas sus fases (hablando del análisis y el diseño).

No obstante, existe la posibilidad de recurrir a alguna otra metodología en cualquiera de las etapas para complementar el modelado en caso de que *Tropos* no admita abarcar algún concepto en particular.

Antes de comenzar con el diseño y modelado con *Tropos*, es necesario comprender algunos conceptos. Dichos conceptos son la base del *framework i** y están definidos en [59]:

Actor

Es el concepto central del *framework i**. Es la abstracción de una entidad activa que es capaz de realizar una acción independiente.

Cabe mencionar que Eric Yu especifica que un actor no es lo mismo que un agente. Mientras que un actor es una entidad abstracta, un agente se define como una "*encarnación física de un actor"*[14].

Meta (goal)

Se puede definir como un fin o un objetivo que se debe lograr.

Meta suave (softgoal)

Es un objetivo que no necesariamente se debe conseguir, aunque es preferible lograrlo. En este contexto, se refiere a las cualidades de un objeto o actor, como pueden ser rápido, lento, seguro, inseguro, confiable, etc.

Tarea

Son acciones que deben ser realizadas por un actor.

Recurso

Se refiere a una entidad, física o de información.

Descomposición

Este término se utiliza para referir la división, ya sea, de una meta en submetas o una tarea en subtareas, y que deben ser realizadas para lograr la meta o tarea, según sea el caso. Se pueden separar de manera disyuntiva (OR), donde basta con que se ejecute una o más de las subdivisiones; o conjuntiva (AND), donde es necesario que se ejecuten todas. [60]

Medios y fines (means-ends)

Es un enlace que conecta a una tarea, meta o recurso, con una meta, que indica la manera en la que dicha meta puede ser lograda (al realizar la tarea, lograr la meta u obtener el recurso).

[14] Eric Yu explica en [59]: *"the term agent is used to refer to actors with physical embodiment"*, que se traduce como "el término 'agente' es usado para referirse a actores con encarnación física". Empero, como ya se ha mencionado, un agente no es necesariamente físico, sino también puede ser lógico, por ejemplo, un virus de computadora.

Contribución

Es un enlace que indica que una meta, tarea o una meta suave, pueden contribuir en la consecución de una meta suave. Puede ser positiva o negativa. [60]

Necesidad de recurso

Cuando una tarea o meta necesitan de un recurso. [60]

Producción de recurso

Cuando una tarea o meta producen un recurso. [60]

Dependencia

Una dependencia se presenta cuando un actor (*depender*) depende de otro (*dependee*) para algo (*dependum*)[15]. Los tipos de dependencia son:

- **Dependencia de metas (*goal dependency*):** Cuando el actor dependiente necesita de una meta conseguida por otro actor, sin importar cómo se consiga.
- **Dependencia de tareas (*task dependency*):** Cuando el actor depende de una actividad realizada. Ésta debe ser realizada como haya sido especificada.
- **Dependencia de recursos (*resource dependency*):** En este caso, el *dependum* es una entidad, ya sea de datos o un objeto material. Se presenta cuando el actor dependiente requiere que el segundo actor proporcione el *dependum* como un recurso, pero no concierne la manera en la que se obtiene esa entidad.
- **Dependencia de metas suaves (*softgoal dependency*):** Ésta se presenta cuando el *dependum* se trata de una cualidad o atributo. Esta dependencia es similar a la dependencia de metas con la diferencia de que la meta suave no es una característica *a priori* del modelo. Se refiere a los requerimientos funcionales en Ingeniería de Software.

Socialidad

Un actor es social dentro de un ambiente cuando tiene contacto con otro u otros. En muchas ocasiones, un actor depende de otro para lograr sus metas, para desempeñar tareas o para proveer algún recurso. Para estos casos, se utilizan las dependencias ya mencionadas.

Intencionalidad

La intencionalidad de un actor es la caracterización de su comportamiento, es decir, la conducta del actor tiene como fin un propósito. La intencionalidad de un actor explica "*el porqué ejecuta ciertas acciones o prefiere una alternativa sobre otra*". [59]

[15] La dependencia entre actores es relativa, ya que una meta o una tarea también pueden ser *dependers*. El *dependee* en la mayoría de los casos es un actor.

Racionalidad

Se basa específicamente en la relación de las acciones y el comportamiento del actor para conseguir sus metas. Está relacionado con la intencionalidad, aunque el concepto de racionalidad explica el cómo.

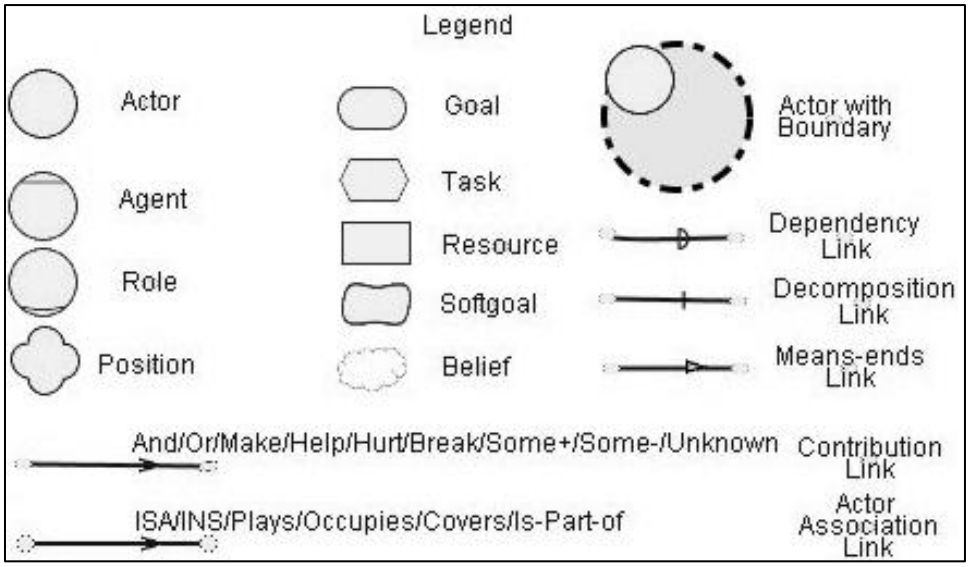


Figura 5.1. Notación definida para el *framework i [59]**

La herramienta a utilizar para el modelado del sistema con *Tropos* será TAOM4E.

5.2.1 Análisis

Se requiere realizar un programa para resolver un laberinto de ciertas dimensiones (16x16 celdas). Debido a que nuestro sistema a desarrollar se basará en el concepto ya definido de lo que es un agente, se tomará como base la programación orientada a agentes.

5.2.1.1 Requerimientos Tempranos

En *Tropos,* la etapa de requerimientos tempranos abarca el análisis del sistema en sus diferentes actores, y sus respectivas metas. En esta etapa se utilizan los diagramas de actores, así como un modelo de dependencias en el que pueden observarse los actores involucrados en el sistema y cómo dependen unos de otros.

Para el modelado del sistema MicroMouse se considerará una arquitectura propuesta en [61], que menciona que un robot autónomo móvil puede ser considerado como un grupo de agentes que se les asigna una tarea en específico a cada uno de ellos y los cuales trabajan en conjunto para un mismo fin, que será el del robot. De esta manera, permite realizar un diseño de un sistema que al principio era de un único agente en un sistema multiagentes.

Analizando el sistema completo, se puede dividir en los siguientes actores:

- Sensores
- Brújula
- Motores
- Encoders
- Puerto Serie
- Microcontrolador
- Programador

Cada uno deberá interactuar con los demás para poder realizar el objetivo principal que es el de resolver el laberinto de paredes.

El primer paso para el diseño en *Tropos* de un sistema multiagente es analizar las dependencias que existen entre los actores en el entorno. Este análisis se realiza con el llamado **modelo de dependencias** que es posible realizar en TAOM4E.

Algunas de las dependencias que pueden verse a simple vista en cada uno de los actores son las siguientes:

- Sensores
  - Ser calibrados por el usuario programador.
  - Dar información acerca de las paredes que se vayan encontrado en el recorrido del laberinto al cerebro del robot (microcontrolador).
- Brújula
  - Establecer comunicación con el microcontrolador.
  - Enviar los datos de los registros *x,y,z* de la brújula al microcontrolador.
- Motores
  - Deben ser activados por el microcontrolador.
  - Permitirán avisar al microcontrolador que el robot se encuentra avanzando.
  - Interactuar con los encoders para generar los pulsos de éstos.
- Encoders
  - Con ayuda de los motores, generar pulsos para indicar el avance o movimientos del robot.
  - El microcontrolador capturará los pulsos generados por los encoders para procesar información de posición.
- Puerto Serie
  - Ser la interfaz para mostrar información acerca de los diferentes sensores utilizados en el robot para el microcontrolador.
  - Enviar la información a un dispositivo compatible con el puerto serie para mostrarla a través de él y que el usuario programador pueda comprenderla.

- Programador
  - Proporcionar un programa fuente al microcontrolador para la resolución del laberinto de paredes.
  - Hacer la debida calibración de los sensores.
  - Percibir la información del estado de los sensores para realizar cambios convenientes, a través del puerto serie.
- Microcontrolador
  - Ejecutar el programa que realice la resolución del laberinto de paredes.
  - Procesar la información obtenida de los sensores para construir un mapa del laberinto con ayuda de la memoria.
  - Utilizar los protocolos correspondientes para la comunicación con la brújula digital implementada.
  - Enviar las señales de movimiento a los motores, para que éstos, a su vez, activen los encoders y los pulsos que se generen sean capturados por el microcontrolador.
  - Enviar información a través del puerto serie para su visualización por parte del usuario programador.

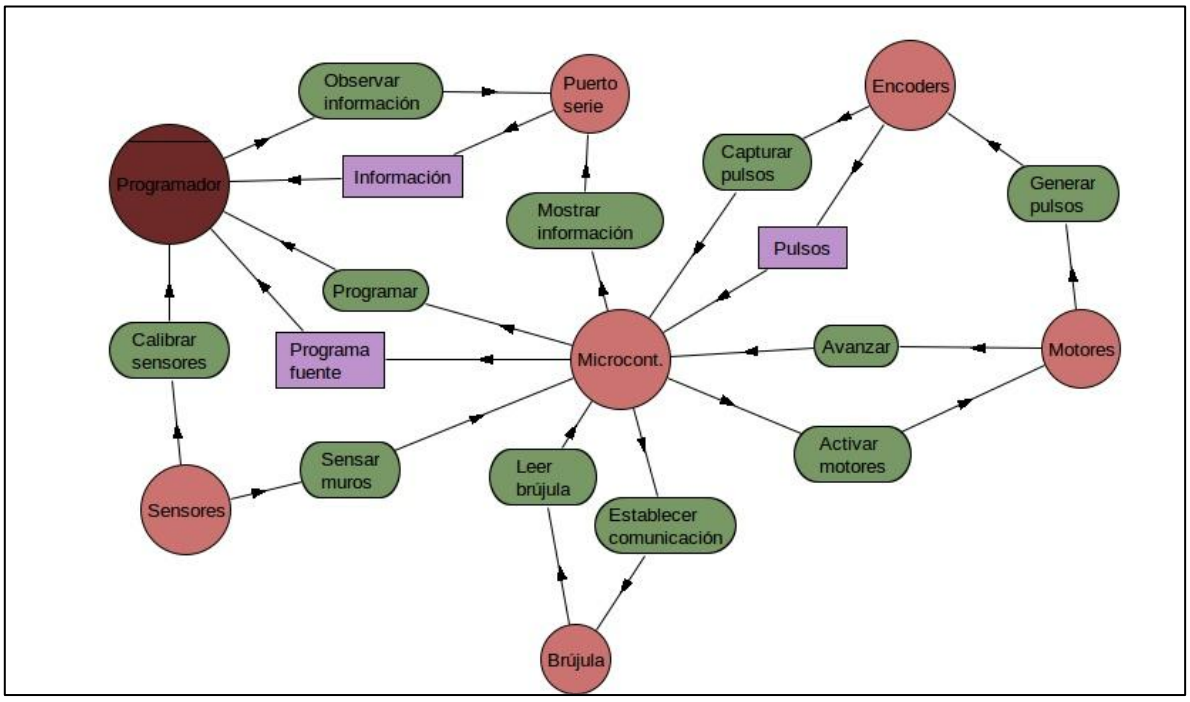


Figura 5.2. Modelo de dependencias de actores en TAOM4E

El siguiente paso es identificar a cada actor individualmente. Cada actor tiene metas y tareas específicas. Éstas se organizan en **modelos de actores**, en el que se definen las actividades de cada actor y las metas que tienen que seguir, además de los recursos que sean necesarios para cumplirlas.

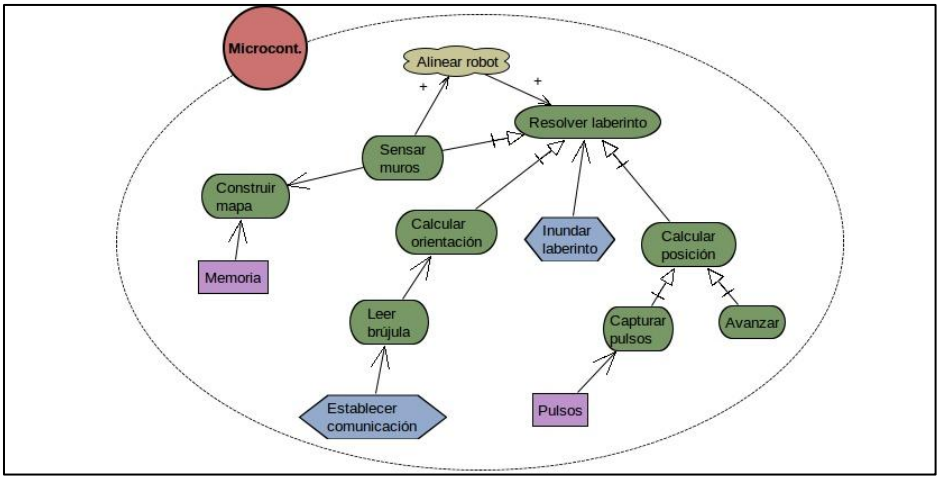


Figura 5.3. Ejemplo de Modelo de actor.

Modelos de actores

- Sensores
  - Su objetivo principal es generar las señales analógicas necesarias para la lectura de los sensores.
  - Para generar la señal analógica, necesita procesar una señal resultante de la incidencia de una señal digital y de la calibración de los sensores. Si se requiere de una señal mucho más confiable, será necesario evitar en la manera de lo posible la luz ambiental que afecte a los sensores.
  - La tarea de calibrar depende del usuario programador.
  - La tarea de recibir la señal digital incidente se hace por parte del actor.
- Brújula
  - Su objetivo principal es establecer la comunicación con el microcontrolador.
  - Para establecer la comunicación, es necesario que se pueda configurar correctamente la brújula.
- Motores
  - La meta a seguir de los motores será el poder hacer que el robot se mueva.
  - Para que lo motores puedan ejecutar la acción de mover al robot, es necesario que se activen. Cuanto más rápido giren, la velocidad del robot será mucho más alta.
  - Los motores, para poder ser activados, necesitan de una señal generada por el microcontrolador.
- Encoders
  - Su objetivo es generar pulsos.
  - Para poder generar los pulsos, es necesario que los motores giren, así que dependerá de los motores que el objetivo se cumpla.

- Puerto Serie
  - Su objetivo principal es el de permitir observar información del sistema al usuario programador a través del puerto.
  - Para que la información se pueda observar, es necesario que se haga la indicación de mostrar información. Como requisito para realizar la muestra, se debe tener la información a mostrar.
- Programador
  - Su objetivo principal será programar el microcontrolador. El producto de esto será un programa apto para el microcontrolador.
  - Para hacer posible el objetivo principal, se deben cumplir dos objetivos previos. En principio se deben calibrar los sensores de manera adecuada para obtener datos precisos. En segundo lugar, es necesario que se haga un programa fuente.
  - Calibrar los sensores requerirá de obtener la información a través del puerto serie para conocer su estado.
  - La realización del programa fuente requiere del programa fuente mismo y de un editor de texto para elaborarlo.
- Microcontrolador
  - El objetivo a seguir del microcontrolador será resolver el laberinto de paredes, que requerirá cierta información para lograrlo, conocer su orientación dentro del laberinto, saber en qué posición se encuentra y la información acerca de los muros. Una manera de mejorar el funcionamiento del robot será alinearlo dentro del laberinto, para mantenerlo alejado de los muros y que su lectura sea optimizada.
  - Una acción necesaria para lograr el objetivo principal es la de realizar la inundación del laberinto para determinar el camino a seguir.
  - Conocer la orientación del robot en el laberinto depende de las lecturas obtenidas a través de la brújula. Éstas, a su vez, dependen de que se haya establecido comunicación entre el microcontrolador y la brújula.
  - La información de los muros depende de los sensores, esta lectura ayuda, por un lado, a alinear el robot en el laberinto y a tener la información acerca del mapa del laberinto.

Una vez hecho el análisis de actores, se puede proceder a realizar el diagrama correspondiente a la etapa de requerimientos tempranos con TAOM4E. Este diagrama se denomina de **modelo mixto**, pues contempla el modelo de dependencias junto con el modelo de actores (*Ver Figura 5.4*).

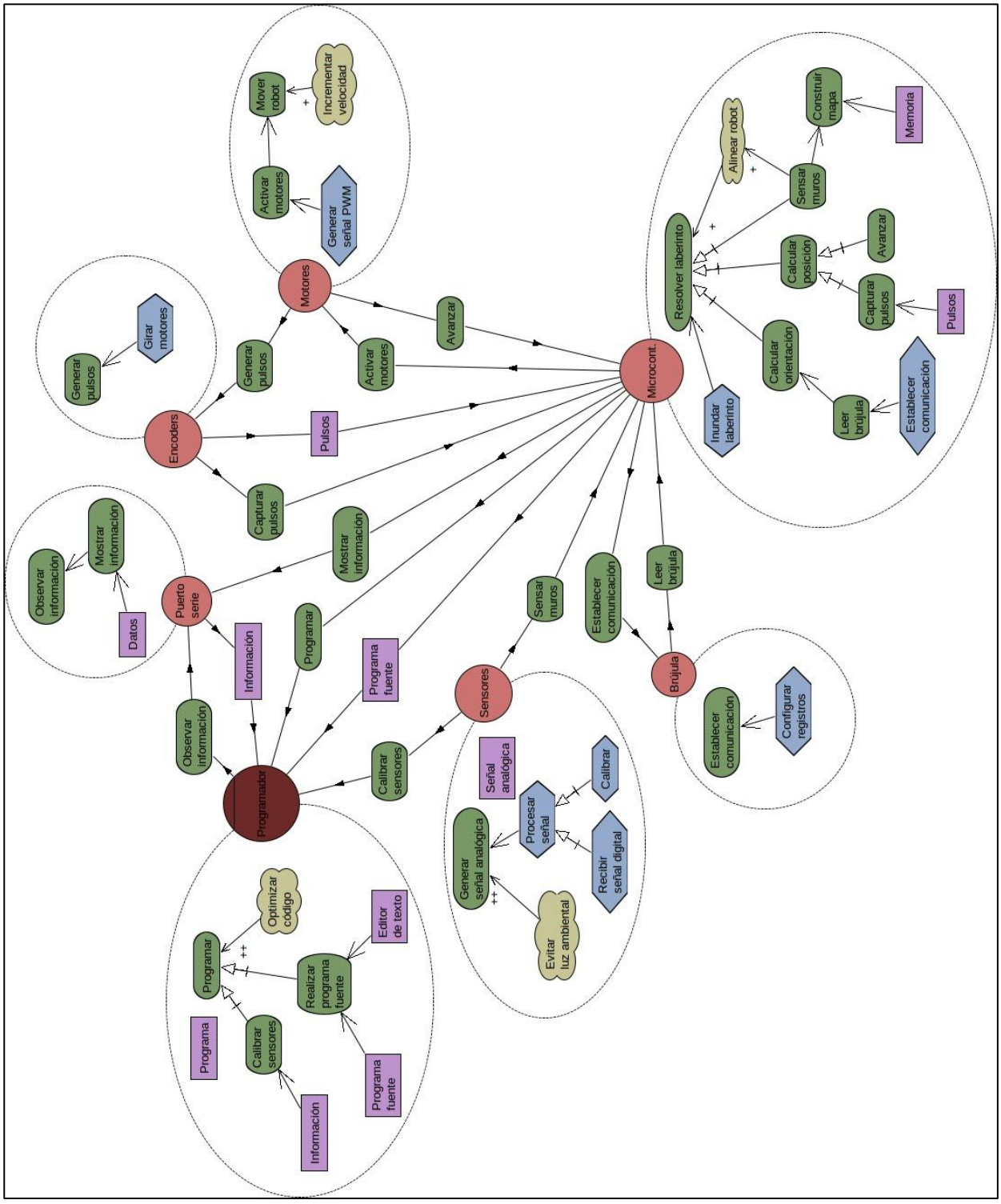


Figura 5.4. Modelo mixto para la etapa de requerimientos tempranos en *Tropos*

5.2.1.2 Requerimientos Tardíos

En la etapa de requerimientos tardíos de *Tropos* se introduce un nuevo actor en el entorno que será el sistema tal cual funcionará. Este sistema se deberá analizar al igual que los actores revisados en la etapa de requerimientos temprana, utilizando los conceptos básicos de *Tropos*. [60]

Primeramente, se deberá analizar el modelo del actor para el sistema. Ésta será la base para el nuevo modelo creado. Nuestro sistema se denominará MicroMouse y el modelo del actor del sistema será el mismo que el del microcontrolador que antes ya fue analizado, es decir, se sistematiza el modelo del microcontrolador debido a que el sistema residirá en él, pues será el que almacene los datos requeridos por el MicroMouse. Cabe destacar que esto no modifica en lo absoluto el modelo mixto (*Figura 5.4*), únicamente el actor *Microcontrolador* pasa a ser el sistema *MicroMouse*.

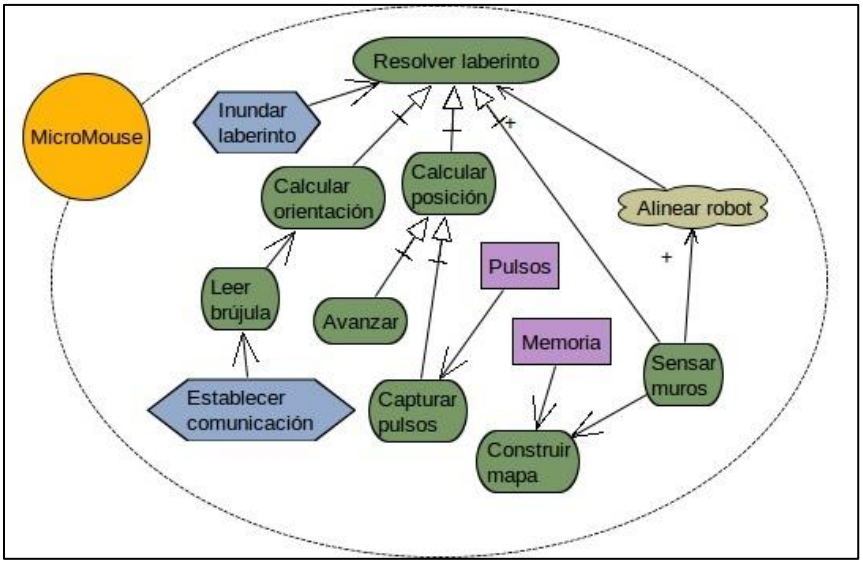


Figura 5.5. Modelo del actor del sistema en la etapa tardía de requerimientos

5.2.2 Diseño

La parte del diseño involucra la definición del actor primordial del sistema, además de definir sus diferentes tareas, capacidades, planes y metas necesarias para cumplirse, aunque delimitándolo como sub-actores, los cuales pueden realizar las acciones del actor de manera separada.

5.2.2.1 Diseño Arquitectónico

En la etapa de Diseño Arquitectónico será necesario crear en el actor del sistema un nuevo diagrama que lo dividirá en sub-actores. Esto permitirá observar de manera detallada cómo

es que funciona el sistema en general dividiéndolo en pequeños subsistemas y que, al mismo tiempo, son simples. Como es el caso, cada subsistema tendrá interacción con los demás, así como dependencias, que se deben determinar en cada subsistema.

Los sub-actores que se han podido distinguir en el sistema MicroMouse con sus respectivas dependencias son:

- Módulo Inteligente
    - El encargado de manejar todas las posibles acciones que ejecute el sistema MicroMouse.
    - Es el que proporciona la información al puerto serie para que ésta sea mostrada al usuario programador.
    - Genera las señales para el movimiento de los motores.
    - Calcula la posición en la que se encuentre el robot dentro del laberinto.
    - Será el encargado de obtener los pulsos del encoder para realizar las acciones que lo requieran.
    - Generará la señal necesaria para el sensado de los muros dentro del entorno del laberinto.
    - Capturará la información generada por el módulo de sensores para determinar la presencia y la distancia de los muros al robot.
    - Solicitará la información de la brújula para conocer su orientación y será capaz de leer lo que ésta haya obtenido.
- Módulo de brújula
    - Se encarga de obtener la orientación del robot con ayuda de la brújula. Asimismo, enviará la información obtenida al módulo inteligente.
- Módulo de sensores
    - Obtienen la información acerca de las paredes alrededor del robot y una vez obtenida, se le envía al módulo inteligente para su procesamiento.
- Módulo de encoders
    - Se encarga de enviar la información acerca de los pulsos generados por los encoders al módulo inteligente para que sean procesados y determinar la posición en la que se encuentra el robot en el laberinto.
- Módulo de motores
    - Será el que interactúe con los actuadores para generar una señal necesaria para el movimiento de los motores. Dicha señal deberá ser generada por el módulo inteligente.
- Módulo serial
    - Es el módulo que hará posible que se envíe la correcta información a través del puerto serie para su visualización por parte del usuario programador. La información que reciba este módulo tendrá que haber sido enviada por el módulo inteligente.

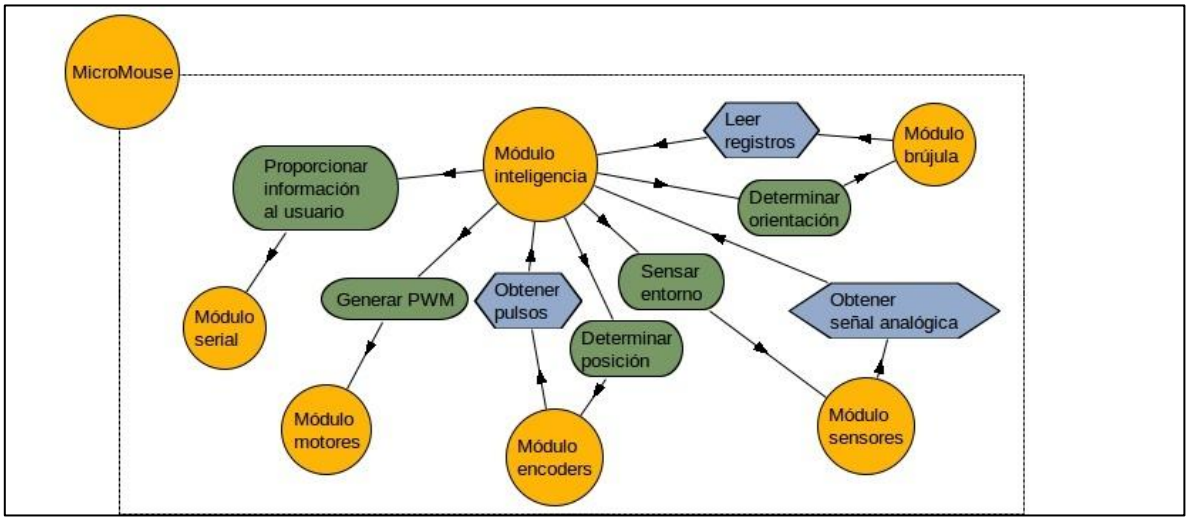


Figura 5.6. Sub-actores del sistema. Se consideran subsistemas

Al igual que pasa con los actores presentes en la etapa de requerimientos tempranos, en el diseño arquitectónico se desmenuza cada sub-actor en sus diferentes metas, tareas y recursos necesarios. En esta ocasión, como cada sub-actor salió de un actor principal (*MicroMouse*), éstos tendrán que delegar las tareas, metas, recursos, etc., del actor principal, de tal modo que deberán integrarse dichas actividades a los sub-actores.

- Módulo Inteligente
    - Este módulo delegará la meta de resolver el laberinto por parte del actor *MicroMouse*. Para conseguir esta meta, dependerá en qué dirección se encuentre realizándolo, ya que como se ha mencionado, una corrida dentro del laberinto consta de ida y vuelta a la celda inicial, así que el robot puede estar buscando el centro del laberinto, o el inicio del mismo. Para ello deberá encontrarse avanzando.
    - Si se requiere que el robot avance, deberá localizar la celda adyacente más barata. Para lograrlo, deberá corroborar que hay una celda en el momento para decidir avanzar, y si no es así, deberá realizar la inundación para recalcular los costos de las celdas que hará posible tener una celda hacia dónde avanzar.
    - Decidir las siguiente celda a la cual moverse, requerirá que el robot sepa su posición y su orientación.
    - Calcular la posición del robot depende de los pulsos capturados.
    - La orientación depende de los valores leídos de la brújula.
    - En el caso de la inundación, conocer el laberinto permitirá obtener mejores resultados respecto a los movimientos posteriores del robot. Por lo tanto, construir el mapa del laberinto en memoria es una contribución positiva.
    - Construir el mapa, dependerá de los muros que se van sensando.

  - Procesar la señal obtenida de los sensores (analógica a digital) deberá contribuir para sensar los muros y, como meta suave, para alinear el robot.
  - Para poder procesar una señal se debió haber obtenido la señal analógica
- Módulo de brújula
  - Para determinar la orientación del robot, se deben hacer las lecturas correspondientes de la brújula.
  - La lectura de la brújula implica leer los 3 registros que almacenan la información de las componentes (*x*,*y*,*z*) del dispositivo físico.
- Módulo de motores
  - Los motores permiten realizar la acción de avanzar al robot, para ello dependerá de una señal de avance y de otra señal que le dé la dirección de giro a cada motor.
- Módulo de encoders
  - Para determinar la posición y enviarle la información correspondiente al módulo inteligente, deberá determinar cuánto ha avanzado, de manera que sabrá si ha llegado a otra celda.
  - Conocer si ha pasado a otra celda dependerá de los pulsos capturados.
  - Obtener información de los pulsos necesita la presencia de los pulsos mismos.
- Módulo de sensores
  - Su meta principal es sensar el entorno en el que se encuentra, así que para ello, deberá de ejecutar alguna de las siguientes opciones: debe sensar la presencia de muros en una celda o puede sensar para realizar la alineación del robot. En ambos casos, se requiere de la lectura de los sensores izquierdo y derecho, únicamente para determinar la presencia de muros en una celda y almacenar en memoria, se utilizará el sensor frontal.
- Módulo serial
  - Solamente depende de una acción que es transferir información a través del puerto serie para que sea mostrada al usuario. Como condición de lo anterior, deberá recibir la indicación por parte del módulo inteligente y la información que será mostrada.

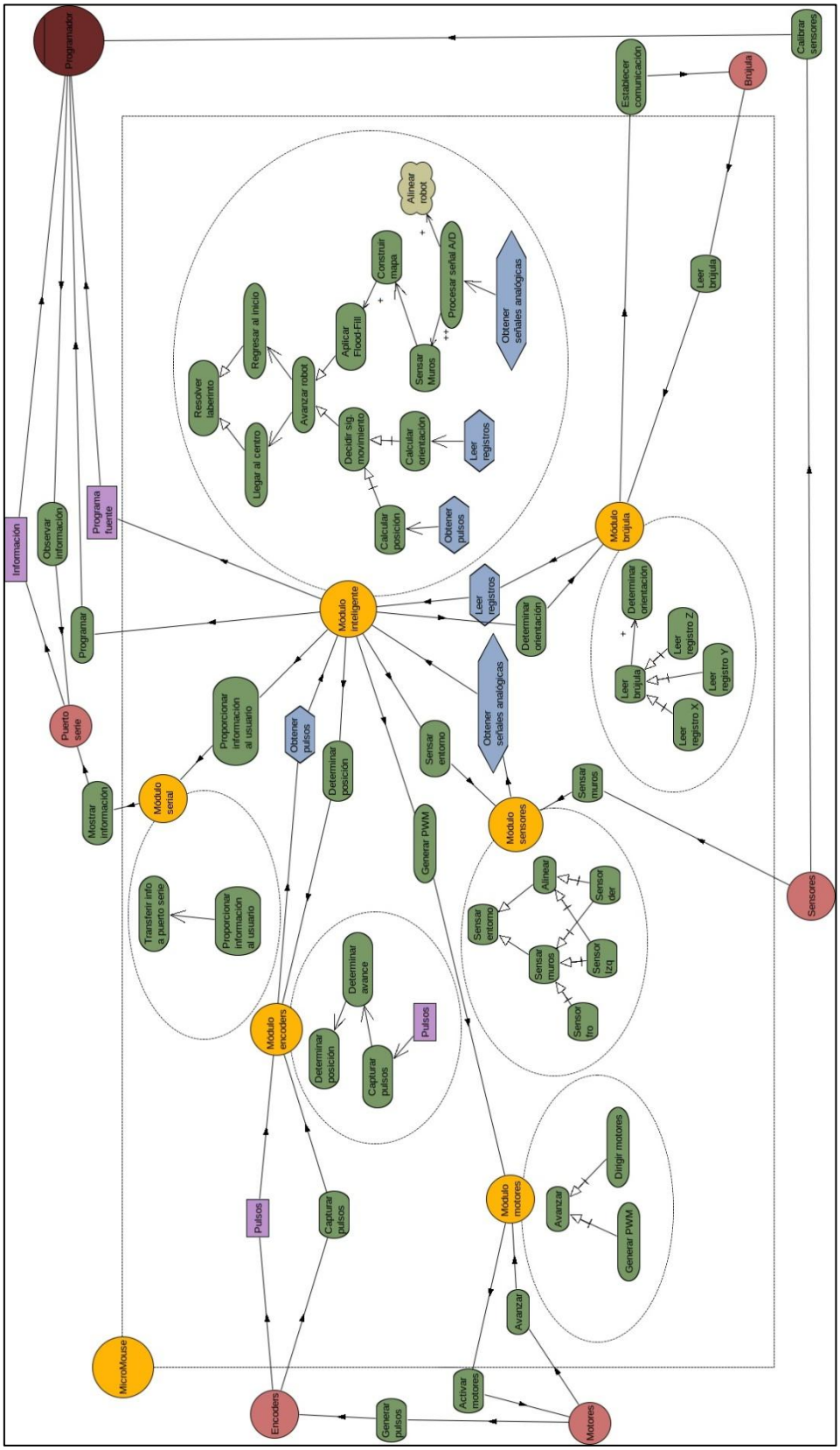

Programador
Calibrar sensores
Establecer comunicación
Brujula
Alinear robot
Construir mapa
Procesar señal A/D
Obtener señales analogicas
Leer brujula
Regresar al inicio
Aplicar Flood-Fill
Sensar Muros
Resolver laberinto
Avanzar robot
Llegar al centro
Decidir sig movimiento
Calcular orientación
Leer registros
Calcular posición
Obtener pulsos
Información
Observar información
Programa fuente
Modulo brujula
Modulo inteligente
Leer registros
Determinar orientación
Determinar orientación
Leer registro Z
Leer registro Y
Leer registro X
Programar
Puesto serie
Proporcionar información al usuario
Obtener pulsos
Determinar posición
Sensar entorno
Obtener señales analogicas
Sensar muros
Mostrar información
Modulo serial
Modulo sensores
Generar PWM
Sensores
Transferir info a puerto serie
Proporcionar información al usuario
Alinear
Sensor der
Sensar entorno
Sensar muros
Sensor izq
Sensor fro
Determinar avance
Modulo encoders
Determinar posición
Capturar pulsos
Pulsos
Modulo motores
Dirigir motores
Avanzar
Generar PWM
Pulsos
Capturar pulsos
Avanzar
MicroMouse
Encoders
Generar pulsos
Activar motores
Motores

Figura 5.7. Modelo de Diseño Arquitectónico completo

5.2.2.2 Diseño Detallado

Una vez obtenido el modelo del sistema principal junto a todos los actores dentro del sistema y que se identificaron las capacidades de cada uno de los actores y sub-actores participantes con ayuda de los diagramas de metas y de actores del sistema, ahora es momento de conocer cómo es que funcionan e interactúan entre ellos. Para ello, en la etapa de diseño detallado se definen las reglas de interacción de todos los involucrados, así como sus esquemas de capacidades y planes.

Es en esta etapa en donde *Tropos* se liga a las notaciones utilizadas en UML, pues para diseñar los esquemas de planes y capacidades utiliza la notación de los diagramas de capacidades y para las reglas de interacción utiliza diagramas de secuencia [60] [20].

Diagramas de secuencia

Los diagramas de secuencia son diagramas que muestran la interacción entre diferentes componentes, sistemas y/o actores.

En principio, se analizarán los diagramas de interacción con los actores presentes en el sistema. En cuanto a algunos de los actores que se encuentran fuera del actor principal, es decir el sistema *MicroMouse*, no se tomarán en cuenta, puesto que sus acciones e interacciones están implícitas en los módulos del sistema principal. Solamente deberá considerarse el actor *Programador* como un actor independiente del sistema principal.

El actor que comenzará con la interacción del sistema será precisamente el actor *Programador*. La primera acción será *Programar* y para ello, el actor *Programador* deberá hacer ciertas peticiones antes de lograr establecer el programa fuente al sistema principal.

El programador hará la petición al puerto serie para que le sea mostrada la información de los sensores, puesto que la meta es *Calibrar sensores*. Del mismo modo, el puerto serie deberá hacer la petición al sistema *MicroMouse* para que se le proporcione la información. Así, el sistema *MicroMouse* hará la lectura de los sensores para obtener la información. Una vez teniendo la información, se le dará al puerto serie y éste último deberá mostrarla al programador.

Ya que el programador tiene la información de los sensores puede proceder a calibrarlos directamente. Es importante mencionar que esta acción se hace físicamente y conlleva a mejorar la lectura de los sensores. La calibración de los sensores tiene un objetivo específico, que es el de revisar los valores de lectura para cuando el robot está detectando un muro, además, será necesario capturar un valor preciso para mantener cierta distancia de los muros con el fin de alinearlo dentro del laberinto. La idea es evaluar ese valor preciso.

Finalmente, para terminar con la intervención del programador en el sistema, lo último que le corresponde es programar el dispositivo para su funcionamiento. Para conseguirlo, el programador requerirá de ciertos elementos necesarios y que no son considerados en el modelado, éstos son, computadora, entorno de desarrollo, dispositivo programador con interfaz ICSP, cables, etc.

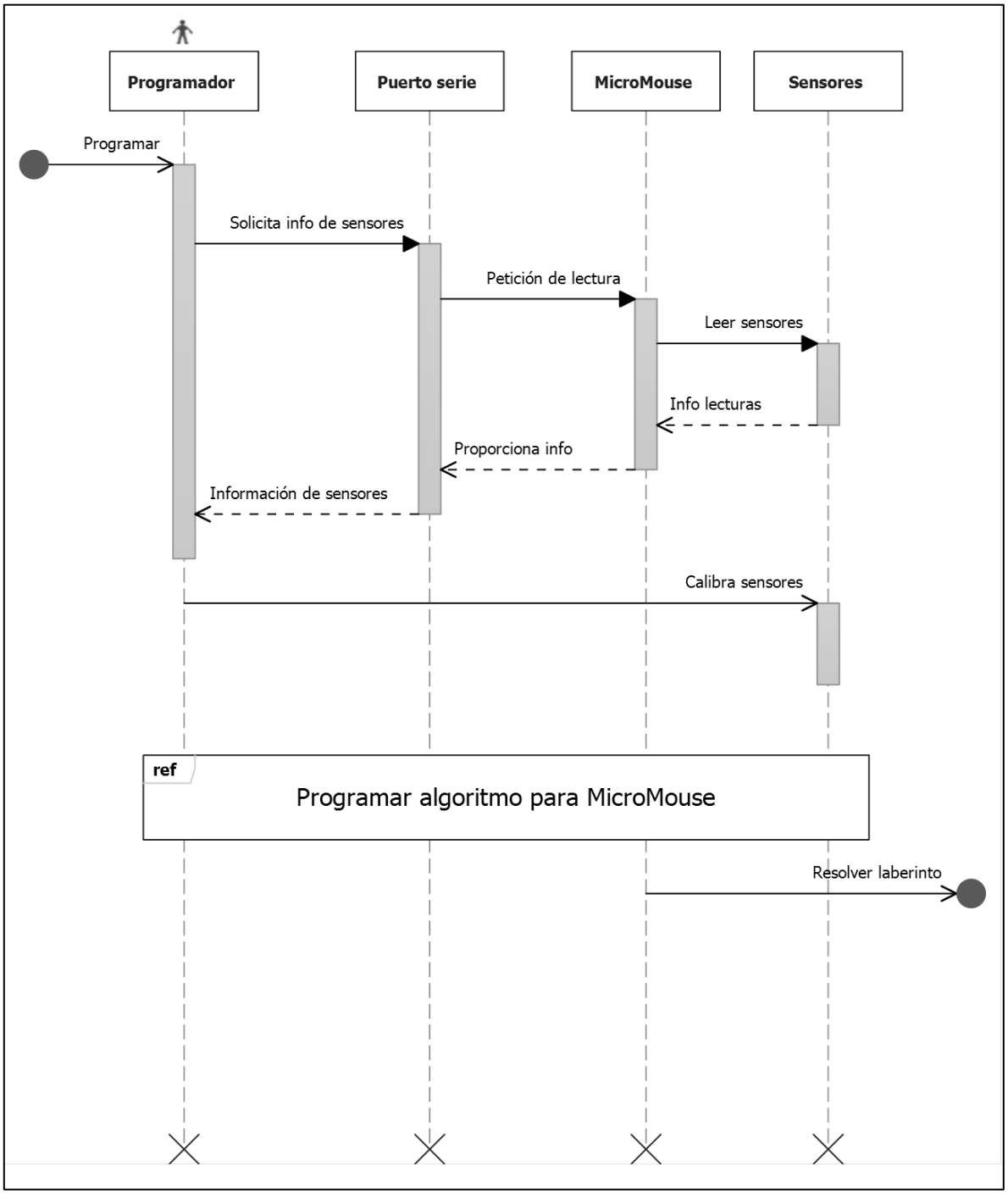


Figura 5.8. Diagrama de interacción entre el actor *Programador* y el sistema *MicroMouse*

Ahora se analizarán cuáles son las reglas de interacción dentro del sistema *MicroMouse*. En este caso, el actor encargado de hacer las peticiones más importantes es el *Módulo Inteligente* el cual administrará las funciones que realicen cada uno de los módulos contenidos en el sistema.

El inicio de la interacción comienza con la petición de resolver el laberinto que le hace al *Módulo Inteligente*. Para conseguir esto es importante que se valga de las interacciones que existen con los demás módulos dentro del sistema *MicroMouse*. El módulo inteligente pide la lectura de los pulsos obtenidos por el módulo de encoders, pero para haber leído algún pulso es necesario que antes gire el motor, por ello, el *Módulo Encoders* solicita al módulo inteligente ejecutar el movimiento de los motores.

Para lograr que los motores tengan movimiento, el módulo inteligente generará una señal de PWM para obtener un giro en los motores y así obtener los pulsos. La señal PWM variará dependiendo de algunos parámetros. Una vez que los pulsos se obtuvieron por parte del módulo correspondiente, el módulo envía la información solicitada anteriormente al módulo inteligente.

Mientras tanto, cuando se van obteniendo los pulsos a través de los motores y encoders, el módulo inteligente está solicitando más información, una de estas solicitudes se hace al módulo de los sensores. La petición de información a este módulo es hecha debido a que se necesita la información acerca de los muros, la cual es la meta de esta solicitud. Otra petición realizada en este diagrama es la lectura de la brújula, que como ya se ha mencionado, depende de leer cada uno de los registros del dispositivo. Cuando la lectura se ha realizado satisfactoriamente, el módulo de la brújula solicita el cálculo de la orientación del robot, que quiere decir que ha terminado su interacción con este dispositivo.

Un aspecto importante a destacar es que las peticiones a estos actores dentro del sistema no necesitan ser coordinadas de ningún modo, esto quiere decir que no importa el orden en el que se realicen, siempre y cuando así sea.

En el momento en el que se han hecho las peticiones por parte del módulo inteligente y éste ha logrado obtener la información correspondiente, el paso siguiente es determinar los movimientos necesarios para avanzar a la solución del problema. Para esto, el módulo inteligente buscará la solución próxima y, de ser el caso, requerirá realizar una inundación dentro del laberinto para asignar valores a las celdas.

Por último, si el problema no se resuelve, se procede a ejecutar nuevamente esta serie de interacciones.

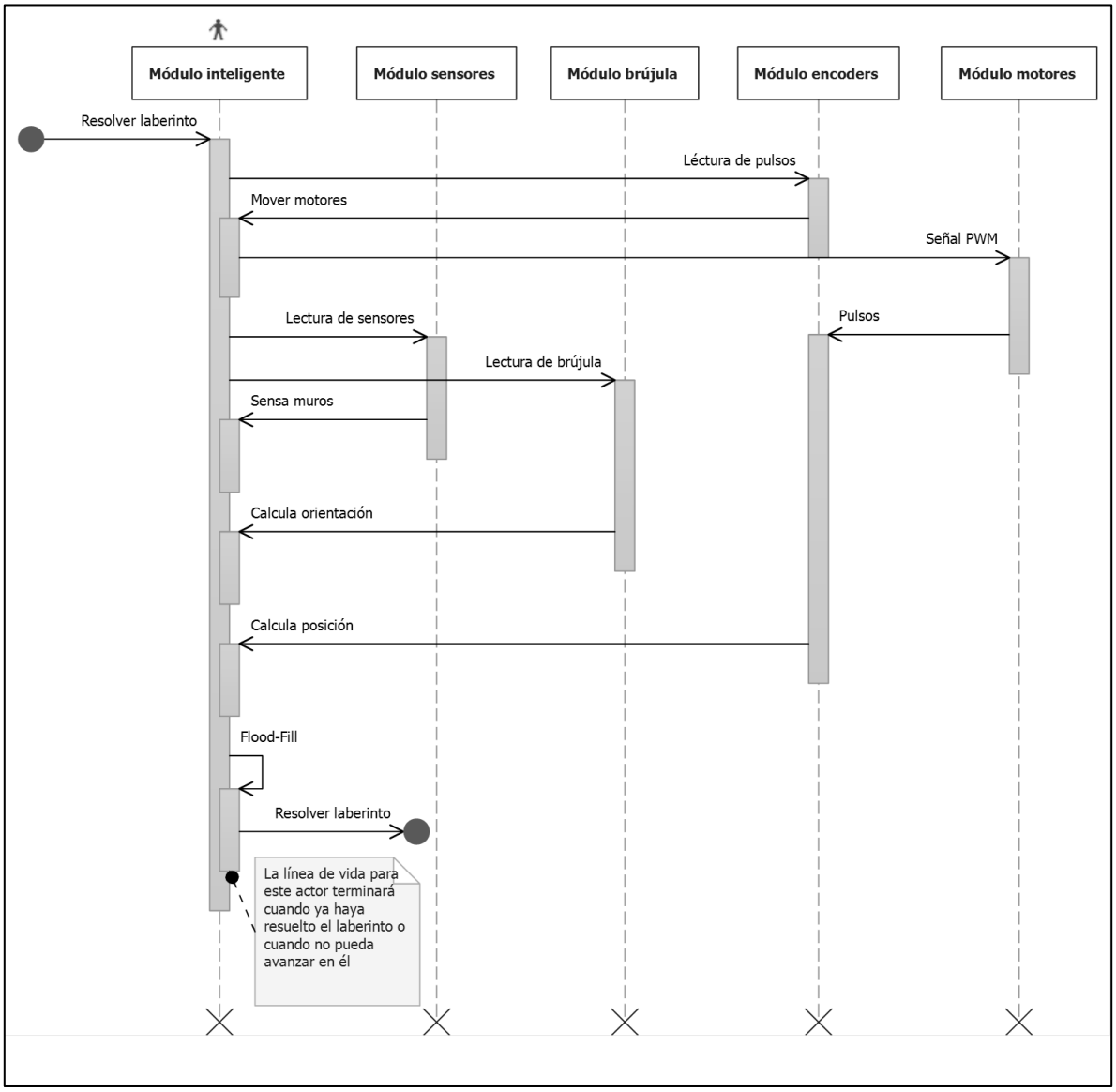


Figura 5.9. Diagrama de interacción entre agentes del sistema *MicroMouse*

Diagramas de actividad

Los diagramas de actividad pueden representar ciertas acciones que se ejecutan por una persona o entidad de software. Para el caso de los agentes, los diagramas de actividad de UML representan las metas y los planes que un agente puede ser capaz de procesar.

Por lo regular, un diagrama de actividad es utilizado para simbolizar una meta o una tarea en un sistema multiagente. Para fines prácticos de este trabajo, sólo se representarán las metas y tareas más relevantes del sistema. También se hará el diagrama de cómo están conectadas estas actividades dentro de cada módulo.

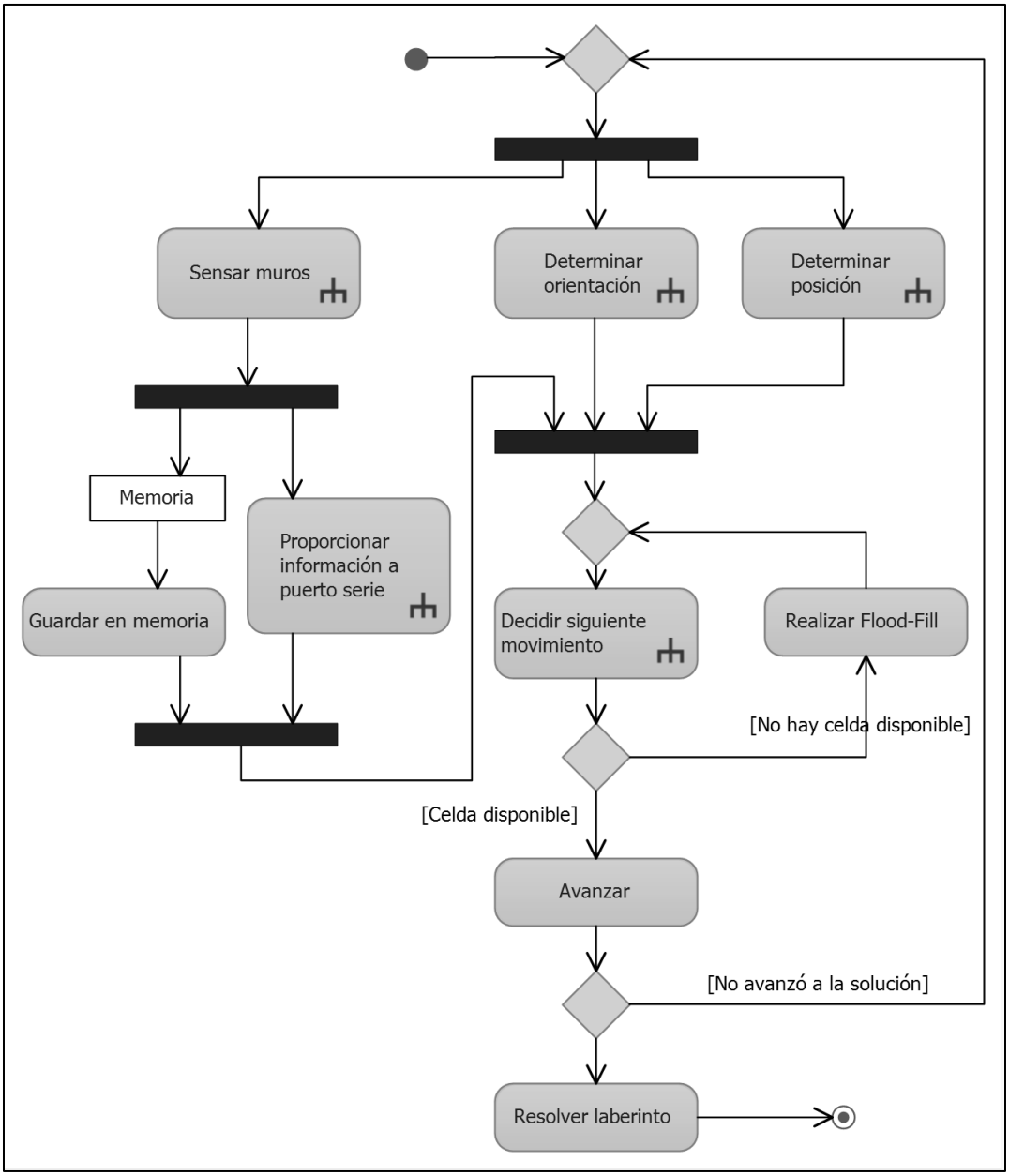
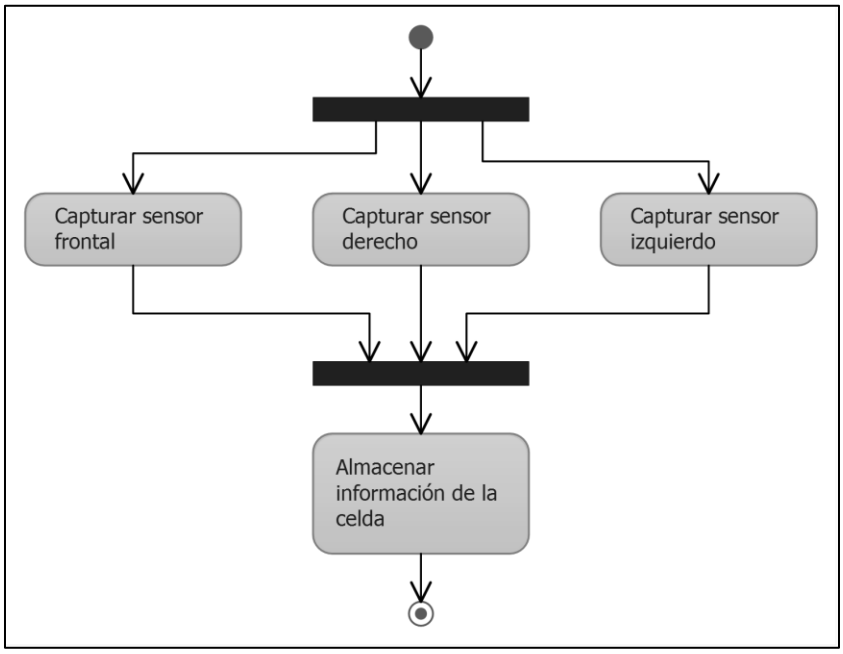


Figura 5.10. Diagrama de capacidades del *Módulo Inteligente*

En el diagrama de actividades del módulo inteligente, se comienza separando las actividades que son "divisibles" y que no dependen una de la otra. Así ocurre con las actividades de lectura para los cálculos correspondientes. Si se observa desde cierto punto de vista, estas actividades se podrían realizar de manera paralela, ya sea teniendo otros actores interactuando entre sí, o en este caso, sub-actores que se dividen las actividades y cuyo actor principal se encarga de reunir los resultados.

Las actividades que tienen un símbolo en la parte inferior derecha son actividades que están definidas independientemente del diagrama en el que se encuentran. Por ejemplo, las

actividades *Sensar Muros* y *Determinar orientación* están definidas en los siguientes diagramas de actividad.



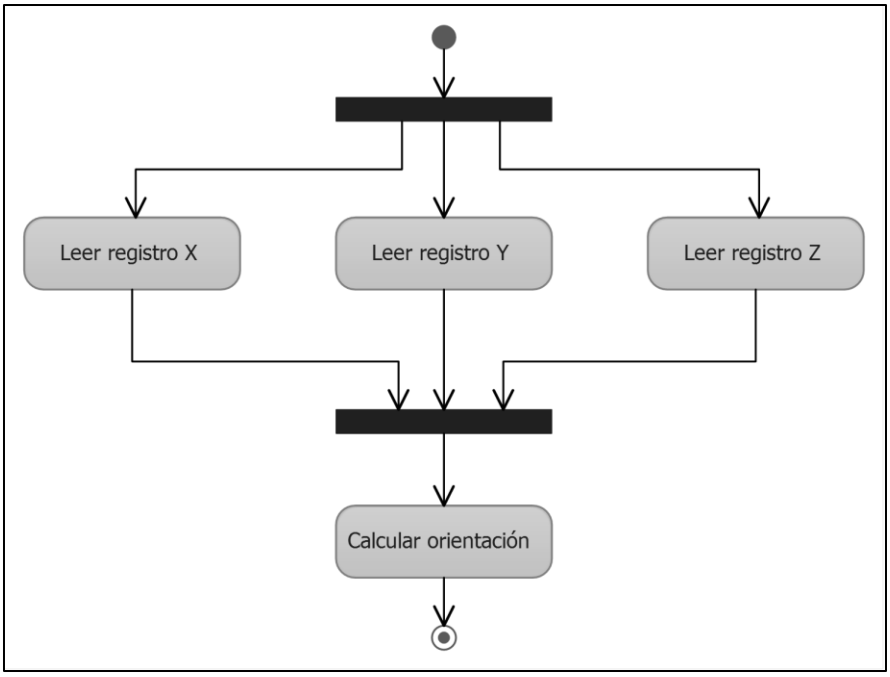


Figura 5.11. Acciones *Sensar muros* (arriba) y *Leer brújula* (abajo)

Los nodos de bifurcación y de unión se utilizan para actividades que pueden realizarse sin dependencia una de otra, pero en el nodo de unión, el flujo de control se detiene hasta que todas las actividades antes de él se hayan completado.

Es posible observar en el diagrama de *Sensar muros* que antes de la actividad *Almacenar información de la celda* existe un nodo de unión, y que esta actividad no podrá ser ejecutada hasta que se hayan completado las anteriores al nodo, lo que significa que para almacenar información de la celda, debió haberse capturado las lecturas de los sensores. Algo similar sucede con la actividad *Calcular orientación*.

Otra actividad dependiente es la de *Decidir siguiente movimiento*. Ésta se encuentra definida en el siguiente:

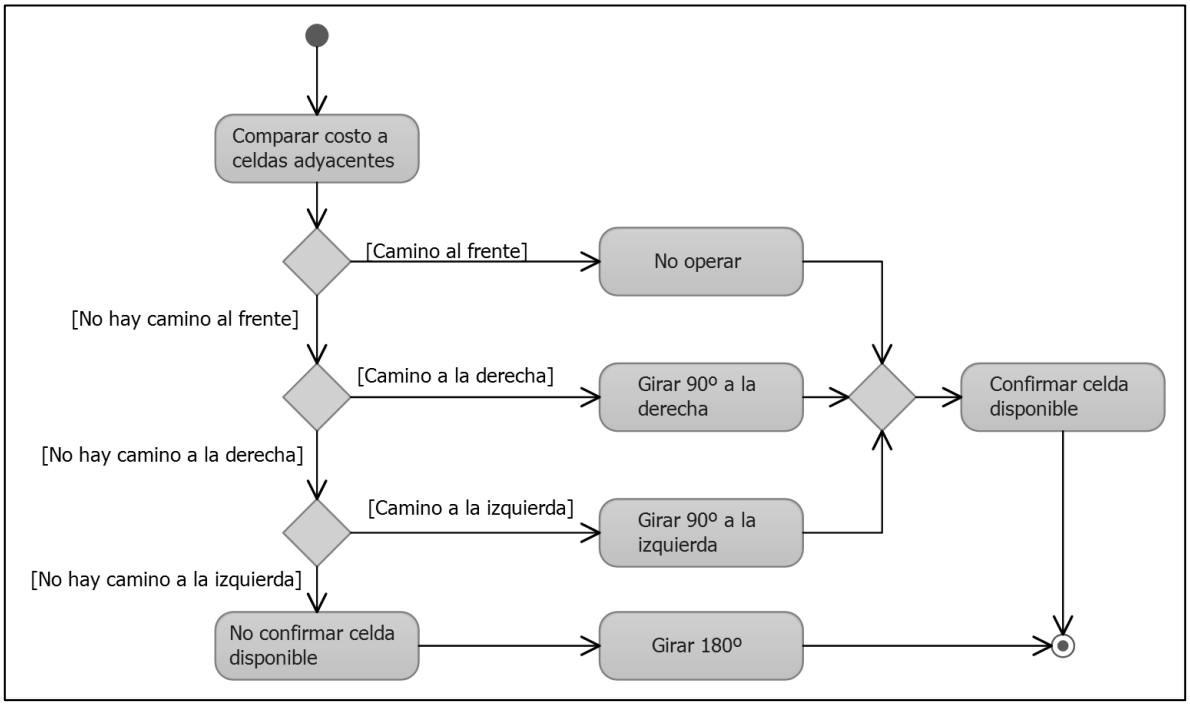


Figura 5.12. Acción de *Decidir siguiente movimiento*

Esa actividad determinará a qué dirección deberá girar el robot para avanzar cuando ha llegado a una nueva celda.

Para el siguiente diagrama, se pueden observar las actividades principales realizadas por el actor *Programador*. Igual que algunos ejemplos anteriores, tiene bifurcaciones para indicar que las actividades no dependen entre sí, pero que deben ser ejecutadas para poder continuar después de un nodo de unión.

El siguiente diagrama del lado izquierdo define la actividad *Determinar posición*. Al mismo tiempo, el diagrama del lado derecho define la actividad *Determinar Avance*, presente en el diagrama izquierdo.

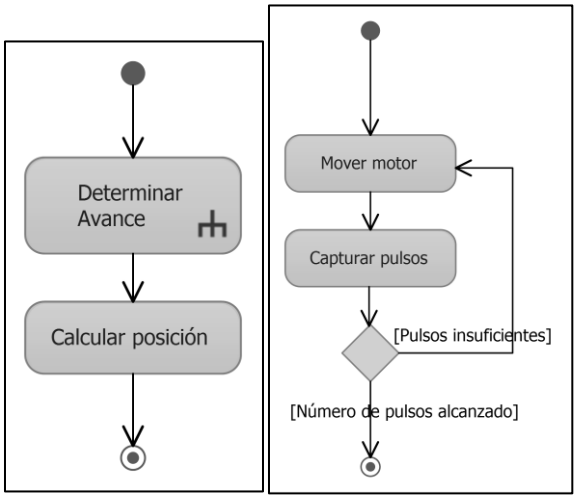


Figura 5.13. Acciones: *Determinar* posición (izquierda) y *Determinar Avance* (derecha)

Para poder programar el dispositivo principal, antes de realizarlo, es necesario escribir un programa fuente y lograr la calibración de los sensores. Se puede ver que la actividad *Observar información de sensores* está definida en otro diagrama (lado derecho).

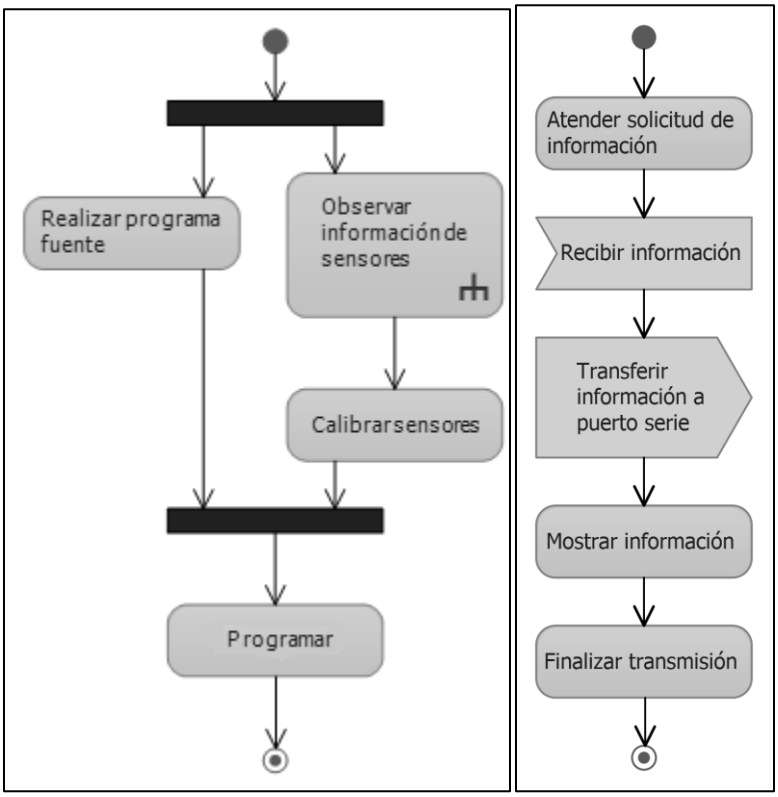


Figura 5.14. Diagrama de capacidades de *Programador* (izquierda) y la acción *Observar información de sensores* (derecha)

Por último, el diagrama de actividad definido para el módulo de motores es el siguiente:

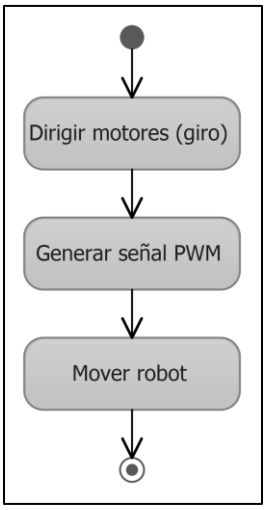


Figura 5.15. Diagrama de capacidades del *Módulo motores*

Éste es muy sencillo, pues depende de 3 acciones secuenciales. En primer lugar, la acción *Dirigir motores* determinará en qué sentido girarán cada uno de los motores. La segunda acción *Generar señal PWM* permitirá definir la velocidad a la que girarán. Al final, *Mover motores* proporcionará las señales pertinentes para el movimiento de los motores.

5.2.3 Implementación

Esta etapa de implementación considerará la codificación de lo ya antes modelado. Para lograr esto es conveniente hacer un análisis por medio de diagramas de flujo.

El lenguaje de programación utilizado para el robot MicroMouse, y en específico, para el microcontrolador seleccionado es el popular lenguaje C. La herramienta o entorno de desarrollo a utilizar para la elaboración del código para el microcontrolador será MPLAB® de Microchip, con una herramienta de complemento para la compilación del lenguaje C en dicha herramienta de nombre MCC18, que admite la programación de dispositivos PIC de la serie 18. Además, el código utilizado en el microcontrolador está basado en una plantilla creada para la asignatura de Sistemas Embebidos de la Facultad de Ingeniería, en el semestre 2012-I.

Para comenzar, el agente iniciará en una posición definida en el laberinto. Los sensores servirán para localizar los diferentes muros que se encuentran dentro del laberinto. Para ello, se mecesita una función o método capaz de definir la presencia de muros mientras el agente se encuentra en una celda. Una vez que esto haya sucedido, el agente almacenará en memoria la celda con sus respectivos muros.

Recordando el algoritmo *Flood-Fill*, para utilizar este algoritmo al comenzar la búsqueda es necesario contar con 2 condiciones iniciales, las dimensiones del laberinto y las coordenadas de la(s) celda(s) objetivo[16].

El algoritmo sugiere que se le den costos a cada celda. Dichos costos serán calculados respecto a la(s) celda(s) objetivo. El costo que tendrá la celda objetivo será un valor de 0, a partir del objetivo, las celdas que se encuentren adyacentes (norte, sur, este y oeste) del objetivo tendrán un valor +1 al del objetivo, es decir, será de 1 y se considerarán del primer nivel; las celdas adyacentes a éstas tendrán un valor +1 a ellas, quiere decir que será 2 y se encontrarán en el nivel 2; las adyacentes del segundo nivel nivel tendrán peso de 3 y serán del tercer nivel, y así sucesivamente. De esta manera, cuando el algoritmo haya finalizado la inundación de valores, el laberinto tendrá un valor en cada celda.

El avance del robot MicroMouse dependerá de si en la celda en la que se encuentre en ese momento hay alguna celda adyacente que tenga un valor menor y ésta sea accesible (que no haya muro hacia ella). Si no es así, se tendrá que realizar el algoritmo de inundación para recalcular los valores del laberinto y que de esa manera exista una celda adyacente con un valor menor al de la celda actual. Por lo tanto, la toma de decisión del algoritmo de qué camino tomar hacia el objetivo dependerá de que la celda sea accesible y que tenga un costo menor al que tenga la celda actual.

Para la percepción del entorno, o laberinto, se contará con sensores infrarrojos que determinarán la presencia de muros en la celda actual. Para obtener la orientación del robot se utilizará una brújula digital (magnetómetro).

Los actuadores, que permitirán al robot MicroMouse realizar algunos de los comportamientos definidos, son motores que cuentan con encoders, un tipo de sensor infrarrojo que nos ayuda a conocer la posición del robot.

Algo importante a considerar es que cada celda almacenará dos bytes en memoria. El primer byte corresponde al costo que tenga la celda para llegar a la solución. El segundo byte contendrá la información de los muros dentro de esa celda, de modo que se hará uso de la información a nivel de bits para conocer en qué puntos hay muros. Para lo anterior, se ocuparán 4 bits de los 8 disponibles en el byte, uno será para indicar el Norte de la celda, otro el Sur, uno más el Este y el último será el Oeste. Un bit más será considerado para indicar que esa celda ha sido visitada. Por último, el sexto y séptimo bits son usados al momento de realizar el algoritmo *Flood-Fill*, de lo cual se hablará más adelante.

[16] Una consideración que se hizo para el laberinto fue que si el laberinto tiene dimensiones *nxn*, donde *n* es par, las celdas objetivo serán aquellas 4 que se encuentren en la zona central del laberinto; si *n* es impar, entonces únicamente existirá una y sólo una celda objetivo, situada en el centro del laberinto.

8	7	6	5	4	3	2	1
N/A	En cola	Calculado	Visitado	Muro Sur	Muro Oeste	Muro Este	Muro Norte

Figura 5.16. Estructura del byte usado para construcción del laberinto en memoria en cada celda

A continuación se explicarán las funciones más importantes correspondientes a acciones de los modelos mostrados anteriormente.

Función: *SensarMuros*

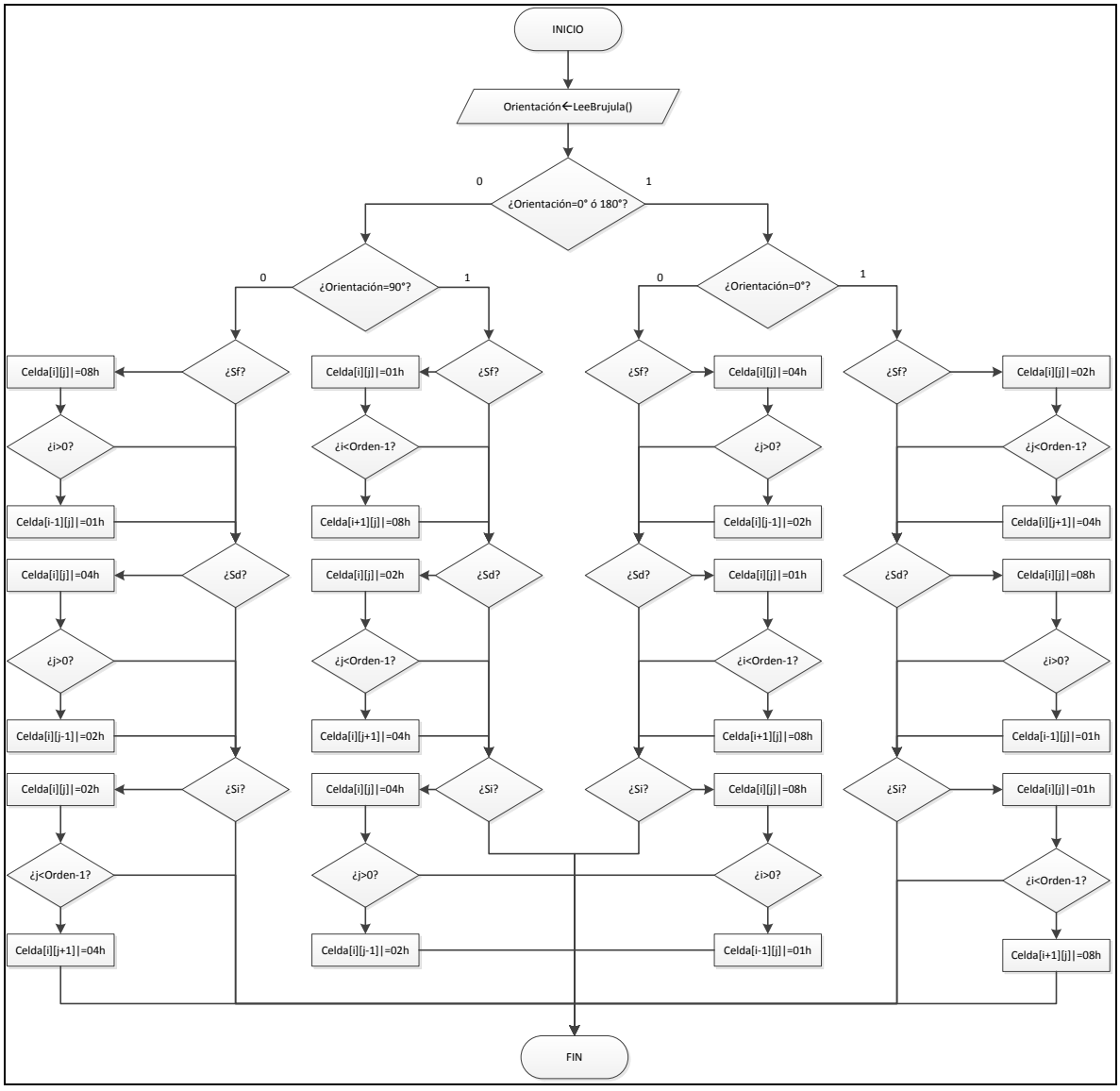


Figura 5.17. Diagrama de flujo de la función *SensarMuros*

Para realizar el sensado de los muros del laberinto, se utiliza una función que implementa los sensores infrarrojos para detectar los muros. Dentro de esta función, el primer criterio para determinar si es necesario realizar el sensado será si la celda en la que se encuentre en ese momento el robot MicroMouse ya ha sido visitada, es decir, si se ha sensado. Si es así,

no es necesario realizar el sensado nuevamente, de lo contrario, se revisan los sensores infrarrojos. El sensor tiene un rango de valor de 20 a 100 cuando hay presencia de un muro de 1 cm. a alrededor de 10 cm. La comparación de presencia de muro en cada sensor se considera de 50, cuando la lectura analógica de cada uno de las entradas utilizadas (AN0, para el sensor izquierdo; AN1, para el sensor frontal; y AN2, para el sensor derecho) es menor a 50, quiere decir que detecta un muro en ese sensor. En cada sensor que encuentre muro se encenderá el bit correspondiente a ese muro en la celda actual y en su celda adyacente.

La orientación en la que se encuentre el robot MicroMouse es indispensable, pues la lectura de los muros será relativa. Esto quiere decir que cuando la orientación del robot sea de 90°, el sensor frontal se encontrará colocado hacia el muro norte, pero cuando el robot esté a 180°, el sensor frontal se encontrará orientado hacia el muro oeste.

Función: *MoverMotores*

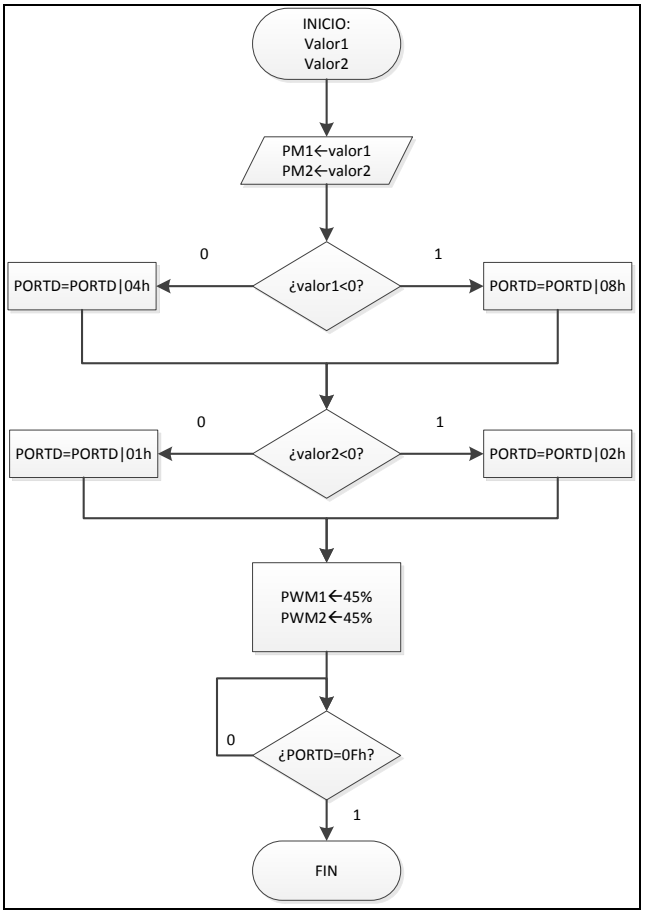


Figura 5.18. Diagrama de flujo de la función *MoverMotores*

Esta función tendría parámetros con el fin de generalizar cualquier combinación de giro de los motores, puesto se tienen dos motores y dos posibilidades de giro de cada uno, hacia adelante o hacia atrás.

Los parámetros *valor1* y *valor2* serán el número de pulsos a alcanzar por parte de los encoders en cada motor. Es decir que si *valor1=20*, el motor 1 girará hacia adelante hasta alcanzar 20 pulsos en el encoder 1. Esos valores son enteros signados y pueden ser negativos, si alguno es negativo quiere decir que el motor correspondiente girará hacia atrás y capturará pulsos negativos. Precisamente, como es posible observar en el diagrama anterior, la comparación de los valores nos dará como resultado la configuración correspondiente para establecer la dirección de giro del motor, que en el microcontrolador a utilizar implementaré el puerto D para enviar las señales a los motores.

También es posible observar que la generación de la señal PWM se encuentra dentro de esta función. La señal generada tendrá un ciclo de trabajo de 45%.

La función terminará cuando los encoders hayan llegado al número de pulsos solicitado y los motores estén detenidos. La captura de pulsos se realizará en la rutina de interrupción, así como la comparación y la operación de detener los motores.

La rutina de interrupción sólo estará presente cuando exista un pulso, en cualquier encoder, en el sistema. Solamente determinará los pulsos del encoder que se obtienen con interrupción por cambio de estado del puerto B.

A través de una operación XOR, el microcontrolador sabrá si las llantas del robot MicroMouse están girando hacia adelante o hacia atrás. Cuando gira hacia adelante, el número de pulsos es incremental, cuando gira hacia atrás, el número de pulsos irá en decremento.

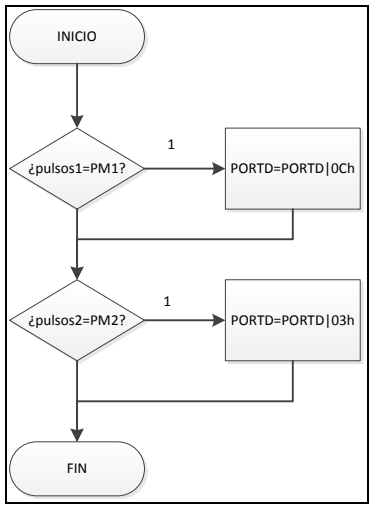


Figura 5.19. Diagrama de flujo de la rutina de interrupción

Función: *AvanzarACelda*

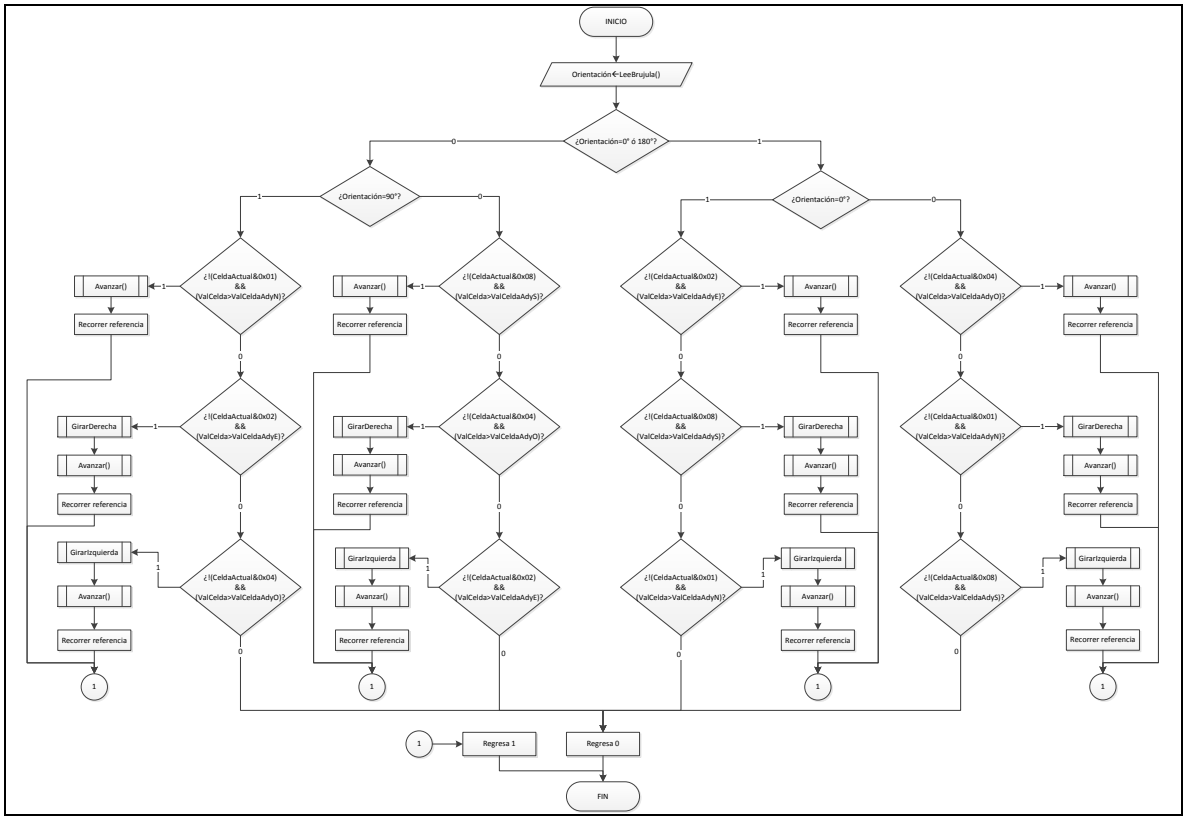


Figura 5.20. Diagrama de flujo de la función *AvanzarACelda*

Para poder avanzar de una celda a otra, es necesario conocer la orientación actual del robot, así puede determinarse hacia dónde avanzar. Cuando el mejor camino está hacia adelante respecto a la orientación del robot, no es necesario hacer el giro, sino que solamente avanzará en línea recta. La función *AvanzarACelda* regresa un valor al final. Si encuentra un muro disponible y avanza hacia él, entonces el valor regresado es 1. Si no encontró un muro disponible, entonces regresará 0.

Las comparaciones realizadas en el diagrama para la función *AvanzarACelda* son sencillas. El primer parámetro de comparación es saber si los sensores detectan muro en la orientación correspondiente a la celda. El segundo parámetro es saber si la celda adyacente tiene un valor menor que el de la celda actual. Ambas condiciones deben ser cumplidas para ejecutar la acción de moverse a la siguiente celda.

Los movimientos en esta función son los fundamentales para poder ir recorriendo el laberinto y encontrar una solución.

Función: *LeerBrujula*

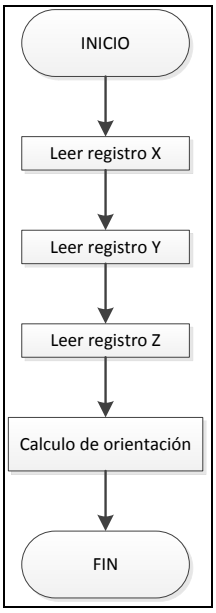


Figura 5.21. Diagrama de flujo de *LeerBrujula*

En la *Figura 5.21* se detalla la función *LeerBrujula* que es utilizada para determinar la orientación actual del robot. En esta función están implementadas las lecturas de los registros de la brújula y, posteriormente, el cálculo de la orientación.

Función: *Sensado*

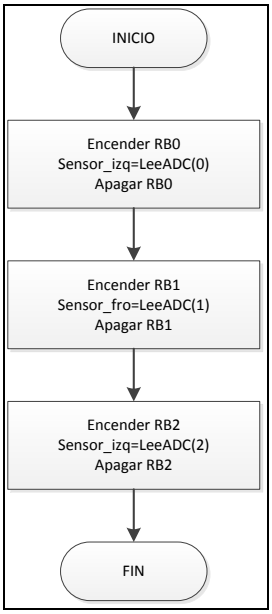


Figura 5.22. Diagrama de flujo de *Sensado*

La función de *Sensado* (*Figura 5.22*) es similar a la de *LeerBrujula*, donde sólo se obtiene la lectura de los sensores. Estos datos son importantes para otras funciones, pues obtienen la información de los muros para construir el mapa del laberinto en memoria, por ejemplo.

5.3 Hardware

En la parte del análisis de hardware, se observarán las diferentes especificaciones necesarias para el diseño físico y la construcción del robot MicroMouse. Del mismo modo, se deberán hacer las pruebas de los múltiples mecanismos de hardware implementados para examinar su funcionamiento, y que éste sea el correcto.

5.3.1 Análisis

- Microcontrolador.
- Sensores de muros.
- Sensor de orientación.
- Motores.
- Encoder.
- Fuente de alimentación

Los sensores a utilizar en el modelo serán:

Sensores infrarrojos: Para revisar la presencia de los muros en cada celda del laberinto. Se realizó un circuito a base de leds infrarrojos y fototransistores, y cuyo arreglo permite obtener una señal analógica a la salida del sistema dependiendo de la distancia a la que se encuentre el objeto, en este caso el muro. Se utilizan 3 (adelante, izquierda y derecha).

Magnetómetro (brújula): Para conocer la orientación del robot mientras se encuentra en el laberinto. Este dato se utiliza principalmente para saber en qué dirección está orientado el robot al momento de construir en memoria el laberinto y conocer los muros presentes en la celda correspondiente.

Encoder: Se utilizará un encoder en cada llanta del robot (2 en total) y éstos servirán para conocer su posición. El tipo de encoders que se implementarán serán incrementales, ya que son más sencillos de utilizar al haber un desplazamiento sobre una superficie.

El robot MicroMouse deberá contar con las medidas estrictas definidas en el reglamento de la competencia para este tipo de robots (en la *Sección 2.2,* del presente trabajo).

Para la elección del microcontrolador a utilizar, existen varias características que pueden beneficiar el funcionamiento del robot. Por ejemplo: la velocidad de procesamiento, el tamaño de la palabra, la cantidad de memoria RAM y ROM que maneja, el número de periféricos que posee, entre otras.

Un factor que se requiere, y considero que es el más importante, es la cantidad de memoria RAM, es decir de datos, que tenga el microcontrolador. El diseño del algoritmo *Flood-Fill* que se ha propuesto, requiere 512 bytes de memoria para el manejo de las celdas del laberinto y sus respectivos valores, más las que requiera el programa por manejo de variables y funciones.

5.3.2 Diseño

El diseño del robot MicroMouse se ha pensado en dos bases, una superior y una inferior. La base superior contendrá el circuito de control del robot, mientras que la base inferior tendrá en ella los actuadores.

En primera instancia, se ha decidido realizar el circuito del robot en un circuito impreso de doble cara. Con ello, se espera el ahorro de espacio y aprovechar la circuitería de doble cara para hacer uso de la mayoría de componentes posible en una sola placa.

El modo de alimentación del circuito se ha diseñado para que se pueda ejecutar de dos maneras, a través de una fuente lineal o de una fuente conmutada. La fuente de alimentación lineal es un circuito convertidor de voltaje. Hay diferentes conversiones de voltaje ya que el voltaje puede ser corriente continua, o directa, y de corriente alterna [62]. Para nuestro caso, se utilizará un convertidor de voltaje de corriente continua a corriente continua. La fuente lineal es simplemente un regulador que mantiene una salida de voltaje continua, siempre que el voltaje de entrada sea igual o mayor. La fuente conmutada es también un circuito convertidor de voltaje pero que contiene un circuito de control y regulación del voltaje a la salida [63]. Es una fuente mucho más sofisticada, pues hace más eficiente la disipación de potencia. [64]

La configuración de los motores se ha pensado de manera diferencial [65] con una tercera rueda, sin tracción, que permitirá un apoyo frontal al robot. La tercera rueda deberá poder hacer el giro en cualquier dirección. El eje de los motores cruzará el centro del robot, con el fin de que los movimientos angulares, en los giros, sea homogéneo en cualquier dirección y sin mayores complicaciones.

Para poder realizar pruebas del robot y obtener valores de los sensores para observar lo que realmente se está sensando en tiempo real, se hará la comunicación con el microcontrolador por medio del protocolo RS-232. Se realizará la conexión serial a una computadora y así se podrá monitorear el resultado de las lecturas obtenidas por cada uno de los sensores que se implementarán.

La programación del dispositivo se realizará por ICSP™ (*In-Circuit Serial Programming*)[17] para evitar el montaje y desmontaje continuos del microcontrolador e impedir dañar el

[17] ICSP es una marca de Microchip Technology Inc.

circuito integrado o alguno de los componentes que conforman al robot. La programación se hará por medio de un kit de programación para dispositivos microcontroladores de la marca PIC® de Microchip Technology Inc. Este dispositivo de programación es el conocido como PICKit2® y se utiliza principalmente para la programación a través de ICSP, aunque se puede realizar la programación a través de otros métodos.

Justificación de los componentes utilizados

Las razones por las que se utilizan los componentes mencionados se enlistan a continuación:

- **Microcontrolador:** Se decidió utilizar el microcontrolador PIC18F4520. Dicha decisión fue tomada porque cuenta con los elementos necesarios y suficientes para poderlos implementar en un robot MicroMouse. Cuenta, además, con memoria suficiente para almacenar un programa y para guardar datos. También, contiene una memoria EEPROM útil para guardar información del laberinto de manera permanente.
  La velocidad de procesamiento es relativamente rápida, ya que tiene un oscilador interno de hasta 32 MHz y permitirá realizar alrededor de 8 MIPS. Otra razón por la que decidí utilizar este microcontrolador es por su bajo costo.
  Por otro lado, la desventaja principal es el tamaño del circuito integrado. Se usará DIP de 40 pines. La longitud del dispositivo es aproximadamente de 5 cm. (2 pulgadas). [66]
- **Fuente de alimentación:** La fuente de alimentación debe ser una que suministre 5V a la salida para el sistema completo. El diseño de la fuente fue pensado para tener dos alternativas: una fuente lineal y una conmutada. La idea principal fue la de realizar una fuente conmutada que pudiera incrementar el voltaje a 5V cuando la entrada fuera menor, pero la adquisición de los componentes no es fácil, además de ser más costosos. La otra alternativa resulta ser más económica. De cualquier modo, el circuito se diseñará para tener las dos posibilidades.
- **Sensores infrarrojos:** Estos sensores son utilizados para sensar los muros, debido a que permite hacer la lectura de la presencia de un objeto, además que su uso es relativamente sencillo y son demasiado económicos, a comparación de otros. Además, se implementó una circuitería para su uso de manera analógica, pues de este modo nos permitirá obtener la distancia a la que el robot está de los muros. Dichos circuitos son conocidos como *Side Looking Sensors*.
  Los sensores infrarrojos también se utilizan en los encoders para conocer el número de pulsos que ejecuta el robot al avanzar. Con la lectura de éstos se

puede aproximar la posición en la que se encuentra el robot, hablando de celdas.

- **Magnetómetro:** Este sensor es un dispositivo que reconoce la orientación, en 3 ejes: *x*, *y* y *z*, en la que se encuentra el sensor respecto al campo magnético de la Tierra. Es utilizado como brújula digital y el circuito a usar será el *LSM303DLHC* de la empresa STMicroelectronics[18], y que se puede conseguir en un circuito construido por la empresa Pololu[19] que incluye circuitería necesaria para su implementación. Es un dispositivo con una buena resolución, que hará que las lecturas sean mucho más precisas. Además, éste cuenta con acelerómetro dentro del circuito integrado, la ventaja de lo anterior es que hace posible escalar o implementar el elemento de aceleración para otro tipo de aplicaciones dentro del mismo robot MicroMouse.
- **Diseño de la placa fenólica de doble cara:** La intención de utilizar una placa fenólica de doble cara en el robot es ahorrar espacio y utilizar el menor posible.

Diseño circular del robot: El robot MicroMouse está pensado con una figura circular para aprovechar el espacio de la celda. El diseño circular sería de 10 centímetros de diámetro, lo que implica que los giros del robot siempre se harán en un diámetro de 10 cm. cuyo eje de rotación es el del robot. Permite mayor movilidad a la hora de realizar giros dentro del laberinto. Tomando en cuenta que las dimensiones de cada celda son de 18 cm. de ancho por 18 cm. de largo.

Los sensores utilizados para la lectura de muros serán infrarrojos con una configuración llamada *Side Looking Sensors*. Esta configuración permite conectar sensores infrarrojos para su implementación analógica y serán útiles para determinar la presencia de un muro, además de saber a qué distancia se encuentra alejado el robot. La configuración mencionada es muy inmune a altas cantidades de luz ambiental. [65]

5.3.3 Implementación

Para comenzar, se realizó la base circular que contendrá las llantas con un diámetro de 10 cm. En esta base, la posición de las llantas con todos los componentes que conllevan (motor, encoder, motorreductor, chassis y llanta) no deberá superar los 10 centímetros, ya que sería contradictorio respecto al diseño pensado.

La longitud de las llantas con todos sus componentes es alrededor de 3.5 cm., por lo que el diámetro mínimo de llanta a llanta es de aprox. 7 cm. Por lo tanto, la distribución de las llantas estará en el rango de 7-10 cm. (*Ver la Tabla 5-1*)

[18] *http://www.st.com/*
[19] *http://www.pololu.com/*

Por otro lado, utilizando Geometría, la llanta tiene 4.2 cm. de diámetro (d_{llanta}). La circunferencia de la llanta se calcula con la ecuación: $C_{llanta} = 2\pi r = \pi d_{llanta}$ [cm].

Sustituyendo d_{llanta} en la ecuación anterior: $C_{llanta} = \pi(4.2) \cong 13.1946$ [cm]

Es decir, cuando la llanta gira en una dirección y el encoder ha enviado 48 pulsos quiere decir que el robot ha avanzado aproximadamente 13.1946 cm. Por ende, la distancia que el robot habrá recorrido por cada pulso será de aproximadamente 0.274889 cm.

Ahora, para que el robot realice un giro de 90°, y haciendo uso de ambas llantas, se considera el diámetro entre llantas. El arco de giro es una cuarta parte de la circunferencia. Por ejemplo, para hacer un giro a la izquierda la llanta derecha girará hacia adelante y la de la izquierda girará en dirección contraria (*Véase la Figura 5.23*).

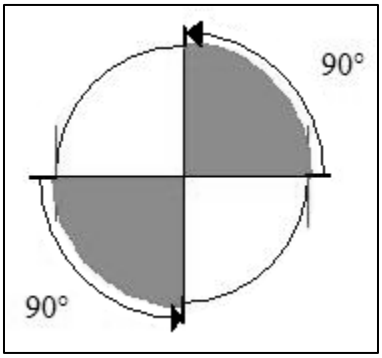


Figura 5.23. Giro a la izquierda

En la *Tabla 5-1* se presenta una relación de los pulsos con la distancia recorrida para realizar un giro de 90°, además del diámetro que debe haber entre cada llanta. Los renglones que están sombreados son los rangos de diámetro posibles para la colocación de las llantas del robot MicroMouse. En primer lugar, los que se encuentran por debajo del rango son imposibles debido a las dimensiones del kit de motores y llantas, y en segundo lugar, el rango abarca únicamente hasta los 10 cm. de diámetro, pues esa es la propuesta para el diámetro total del robot.

Haciendo el análisis de la *Tabla 5-1*, se ha decidido que el diámetro de llanta a llanta sea de **8.4 cm.** Por lo tanto, la base del MicroMouse para las llantas quedaría como se muestra en la *Figura 5.24*.

Pulsos	**Distancia recorrida/pulso [cm]**	**Diámetro *d* (entre llantas) [cm]**	**Longitud de arco de 90° con *d* [cm]**	**Pulsos para recorrer el arco**
1	0,274889	0,35	0,549778714	2,0000026 → 2
2	0,549778	0,7	1,099557429	4,0000052 → 4
3	0,824667	1,05	1,649336143	6,0000078 → 6
4	1,099556	1,4	2,199114858	8,0000104 → 8
5	1,374445	1,75	2,748893572	10,000013 → 10
6	1,649334	2,1	3,298672286	12,0000156 → 12
7	1,924223	2,45	3,848451001	14,0000182 → 14
8	2,199112	2,8	4,398229715	16,0000208 → 16
9	2,474001	3,15	4,948008429	18,0000234 → 18
10	2,74889	3,5	5,497787144	20,000026 → 20
11	3,023779	3,85	6,047565858	22,0000286 → 22
12	3,298668	4,2	6,597344573	24,0000312 → 24
13	3,573557	4,55	7,147123287	26,0000338 → 26
14	3,848446	4,9	7,696902001	28,0000364 → 28
15	4,123335	5,25	8,246680716	30,000039 → 30
16	4,398224	5,6	8,79645943	32,0000416 → 32
17	4,673113	5,95	9,346238144	34,0000442 → 34
18	4,948002	6,3	9,896016859	36,0000468 → 36
19	5,222891	6,65	10,44579557	38,0000494 → 38
20	5,49778	7	10,99557429	40,000052 → 40
21	**5,772669**	**7,35**	**11,545353**	**42,0000546 → 42**
22	**6,047558**	**7,7**	**12,09513172**	**44,0000572 → 44**
23	**6,322447**	**8,05**	**12,64491043**	**46,0000598 → 46**
24	**6,597336**	**8,4**	**13,19468915**	**48,0000624 → 48**
25	**6,872225**	**8,75**	**13,74446786**	**50,000065 → 50**
26	**7,147114**	**9,1**	**14,29424657**	**52,0000676 → 52**
27	**7,422003**	**9,45**	**14,84402529**	**54,0000702 → 54**
28	**7,696892**	**9,8**	**15,393804**	**56,0000728 → 56**
29	7,971781	10,15	15,94358272	58,0000754 → 58
30	8,24667	10,5	16,49336143	60,000078 → 60

Tabla 5-1. Relación de pulsos con la distancia del arco de 90°

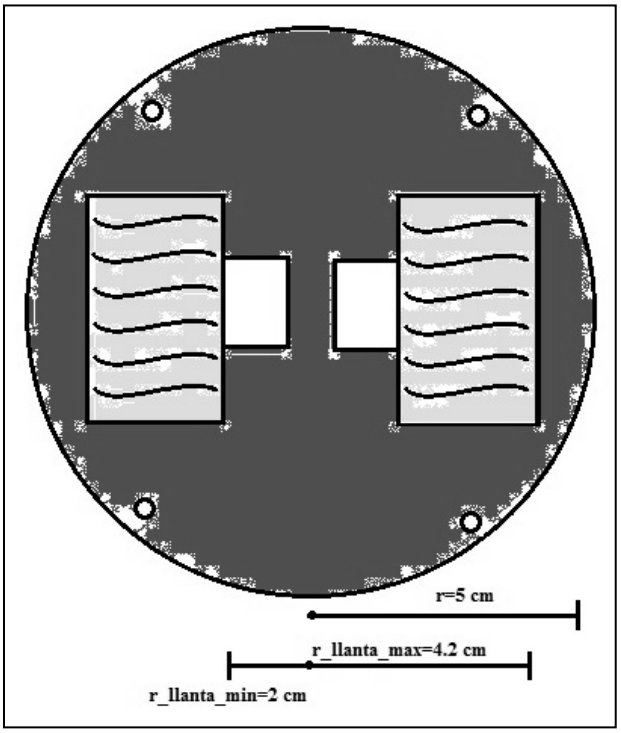


Figura 5.24. Base del robot para las llantas

El material con el que se fabricará la base será SINTRA® [20]. Es un material bastante ligero, además, es rígido, aunque también se puede cortar fácilmente.

El diseño del circuito impreso se realizó con la herramienta EAGLE[21], ésta nos permite diseñar el circuito esquemático y, posteriormente, diseñar la tarjeta PCB (*Printed Circuit Board*) para el circuito impreso con las conexiones realizadas antes en el esquemático. De manera automática, EAGLE genera una propuesta de líneas de conexión en la PCB, con algoritmos que utilizan como parámetros ciertas características del circuito, como son, ancho de líneas, distancia entre ellas, área de trabajo, etc.

En principio, se debe diseñar el diagrama esquemático. Para esto, es necesario conocer las hojas de especificaciones de cada dispositivo utilizado. En la *Figura 5.25* se muestra el diagrama esquemático del circuito del sistema (para una vista más detallada de los diferentes módulos y sus conexiones en el diagrama, refiérase al *Apéndice A. Diagramas esquemáticos*).

[20] *http://www.plastitec.com.mx/Page/productos/sintra/sintra.html*
[21] *http://www.cadsoftusa.com/eagle-pcb-design-software/product-overview/*

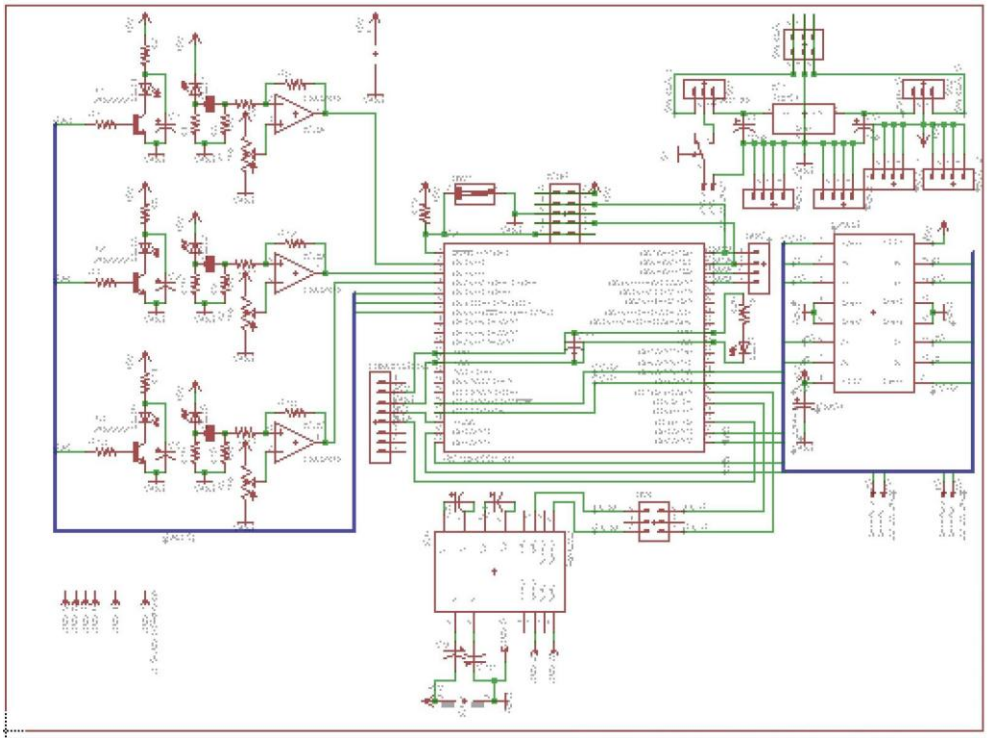


Figura 5.25. Diagrama esquemático general[22]

Al realizar la PCB del diagrama esquemático de la *Figura 5.25* se obtiene un circuito como el que se muestra en la *Figura 5.26*.

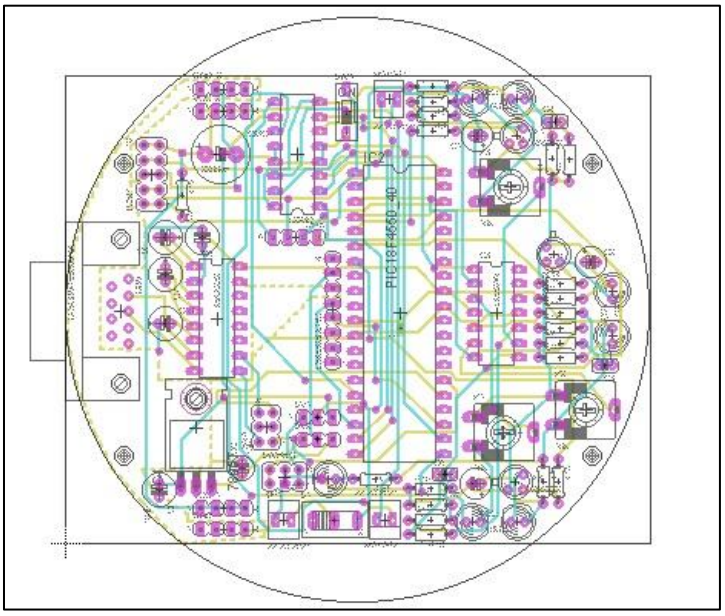
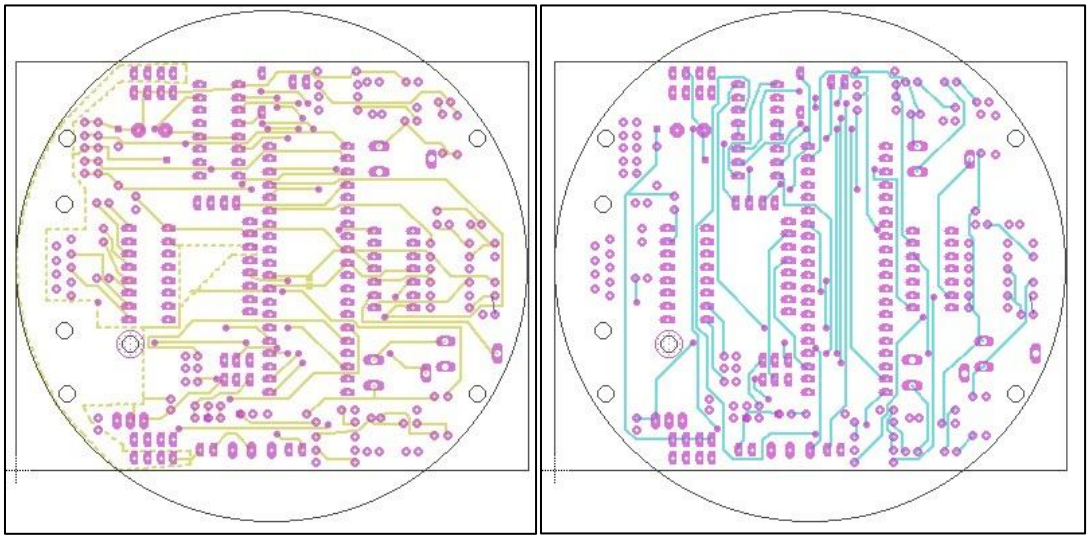


Figura 5.26. Diseño de la PCB para el robot MicroMouse

[22] Una vista más detallada de los diferentes módulos que componen el diagrama esquemático se encuentra en *Apéndice A. Diagramas esquemáticos*

En la *Figura 5.27* se muestran las caras separadas del circuito.



(a) (b)

Figura 5.27. Cara Inferior (a) y cara superior (b) del circuito para la PCB

Una vez que se tiene la PCB, puede realizarse el circuito impreso físico.

5.4 Diseño del algoritmo

Particularmente, la manera en la que estoy utilizando el algoritmo de Inundación es para dar un costo, como una función heurística, a cada celda del laberinto, que es así como funciona el algoritmo.

Utilizo una búsqueda del tipo *Greedy Best-First* para realizar los movimientos a través del laberinto mismo. Lo que quiere decir que cuando el robot llega a una celda, busca una celda adyacente disponible (que no haya muro entre ambas) y con un costo menor, pero tomará la primera que haya encontrado. La primera celda a revisar será la que se encuentre enfrente del robot, si está disponible y tiene menor costo, irá a través de ella sin revisar las demás. Por esta razón el algoritmo se hace voraz.

La jerarquía de búsqueda del camino más corto es la siguiente:

- Revisa la celda de enfrente si se puede mover a ella.
- Si no se realizó lo anterior, revisa la celda de la derecha.
- Si la celda a la derecha del robot no está disponible, revisa la celda de la izquierda.
- Si no se puede mover a ninguna de las celdas anteriores, recalculará los costos de cada celda y regresará una celda anterior, entonces realizará la búsqueda en esa celda del camino más corto.

5.5 Implementación del algoritmo

El algoritmo tendrá más o menos el siguiente funcionamiento:

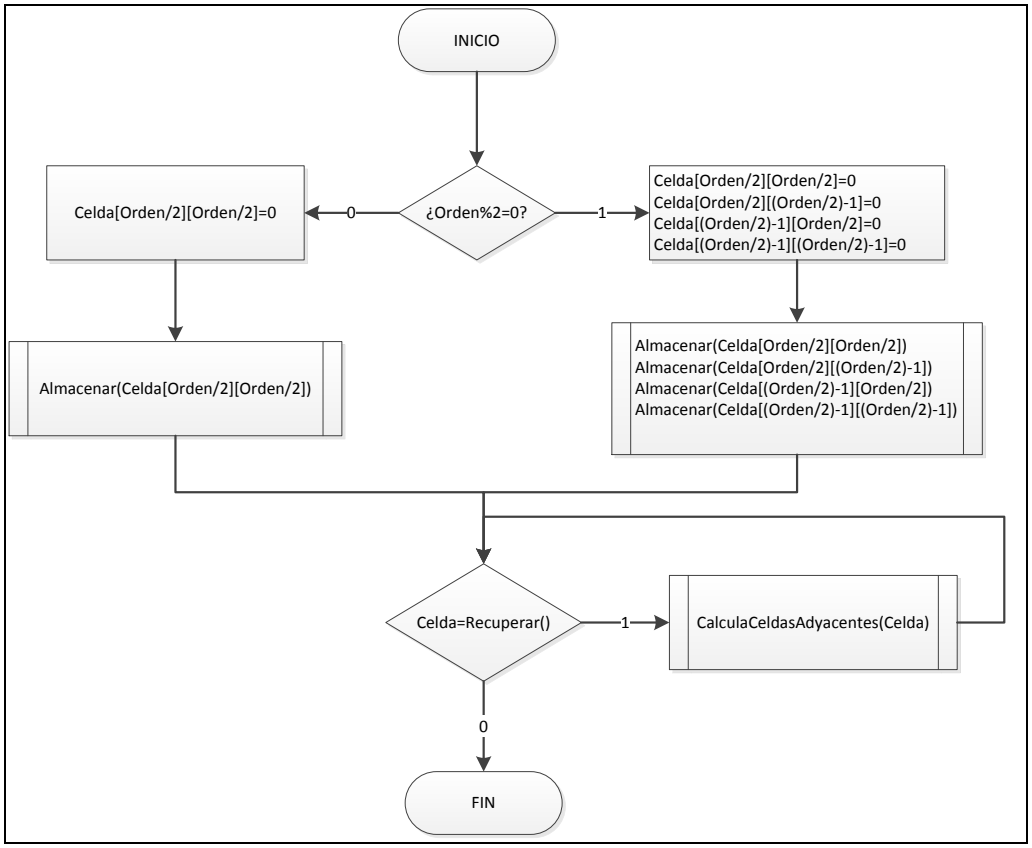


Figura 5.28. Diagrama de flujo de la función *FloodFill*

La función presente en la *Figura 5.28*, únicamente se utiliza para generar los costos de las celdas del laberinto al hacer la llamada. He decidido ayudarme de una estructura de datos de tipo cola para ir generando las celdas adyacentes a otra, empezando con las celdas objetivo. Esto quiere decir que en la cola el primer elemento será la celda objetivo, se obtiene de la cola ese elemento y se calculan sus adyacentes. Esos adyacentes se introducen en la cola y cuando finaliza con el primer elemento, el siguiente será el primer adyacente generado. De esta manera puede asegurarde que las celdas se van generando por niveles.

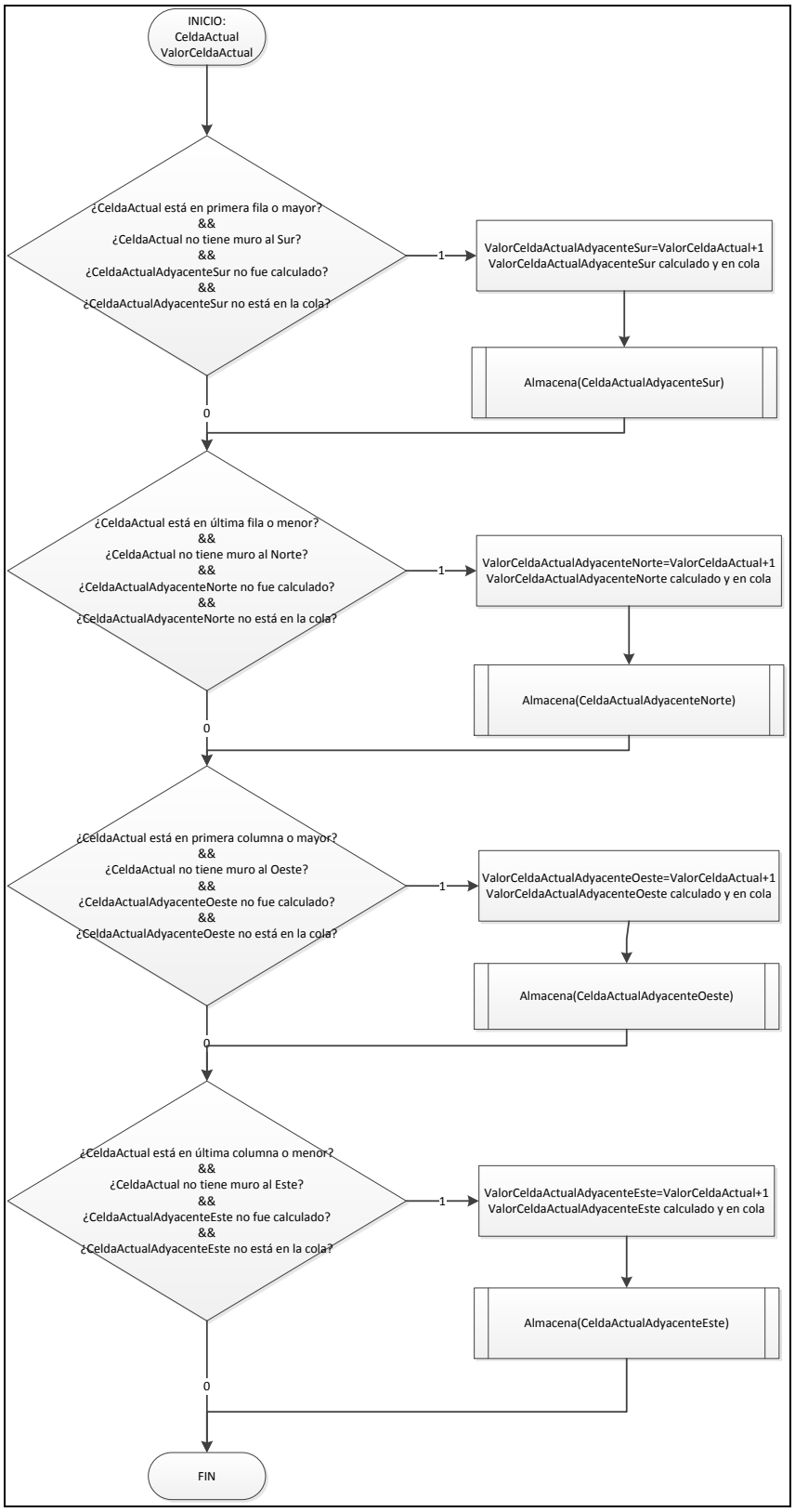


Figura 5.29. Diagrama de flujo de la función *CalculaCeldaAdyacente*

La función *CalculaCeldaAdyacente* de la *Figura 5.29* está implementada para ir generando las celdas adyacentes al momento de calcular los costos de cada celda en el algoritmo de inundación.

Esta función deberá revisar minuciosamente las celdas adyacentes a la que se está analizando, ya que puede haber varias posibilidades para no considerar esas celdas adyacentes. La primera posibilidad es que la celda adyacente esté fuera del rango del orden del laberinto, por lo que se tiene que revisar si la celda actual está en el rango. La segunda posibilidad es que no haya muro hacia la celda adyacente, es decir, que sea accesible a partir de la celda en la que está el robot. La tercera posibilidad, que la celda adyacente ya haya sido calculada y no volver a hacerlo. Y la última posibilidad, es que la celda adyacente ya se encuentre en la cola, en espera de que sea analizada.

Ya que se han calculado los valores de cada celda, ahora el robot procede a solucionar el laberinto.

Para las funciones anteriores se hace uso de la cola. Por ello, hay que definir cómo funcionará.

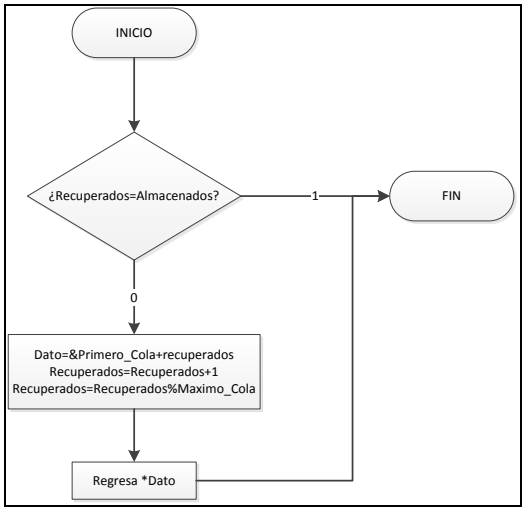


Figura 5.30. Diagrama de flujo de la función *Recupera* de la cola

Básicamente, las variables *Recuperados* y *Almacenados* son índices utilizados para indicar en qué posición están los elementos almacenados primero y último de la cola. Esto quiere decir que cuando ambas variables sean iguales no hay elementos en la cola. Si no son iguales, entonces se modifican las variables de modo que se indique que un elemento ha sido "sacado" de la cola.

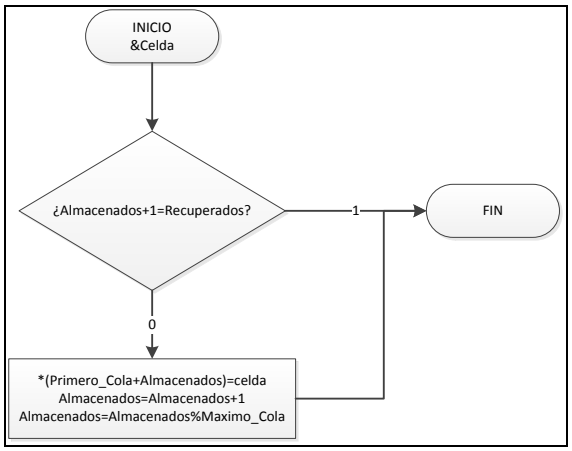


Figura 5.31. Diagrama de flujo de la función *Almacena* de la cola

La función para almacenar elementos dentro de la cola está delimitada en la *Figura 5.31*. En ella es posible observar que para almacenar un dato debe revisarse que la celda está vacía, por lo tanto, se hace una comparación para saber si el número de elementos almacenados es igual al número de los que han sido recuperados, aunque dichas variables siguen siendo índices, de modo que lo que se compara es si los índices son iguales.

6 Pruebas, resultados y conclusiones

6.1 Pruebas de Software

Simulación de robot MicroMouse en Visual C++

De manera previa al desarrollo y elaboración físicos del robot MicroMouse, se realizó una simulación con ayuda de la biblioteca para gráficos en lenguaje C de código libre llamada OpenGL. Esto con el fin de efectuar una simulación del algoritmo aplicado sobre el sistema autónomo, previo a la implementación en un entorno real.

Para este caso, el comportamiento podría ser el ideal, sin embargo, es sabido que en un entorno real las variables que deben considerarse son diferentes (diseño físico, fricción, peso, iluminación, electrónica, etc.). Por lo tanto, la simulación únicamente es para presentar el funcionamiento del algoritmo antes de implementarlo en el robot.

Lo primero a considerar para la simulación es el entorno de trabajo. En este caso elegí el IDE de Microsoft, Visual C++, en su versión Express 2008. Esta decisión se realizó debido a que ya se cuentan con conocimientos previos de la herramienta y el uso de la biblioteca de código libre OpenGL, y con el fin de realizar la simulación de una manera gráfica.

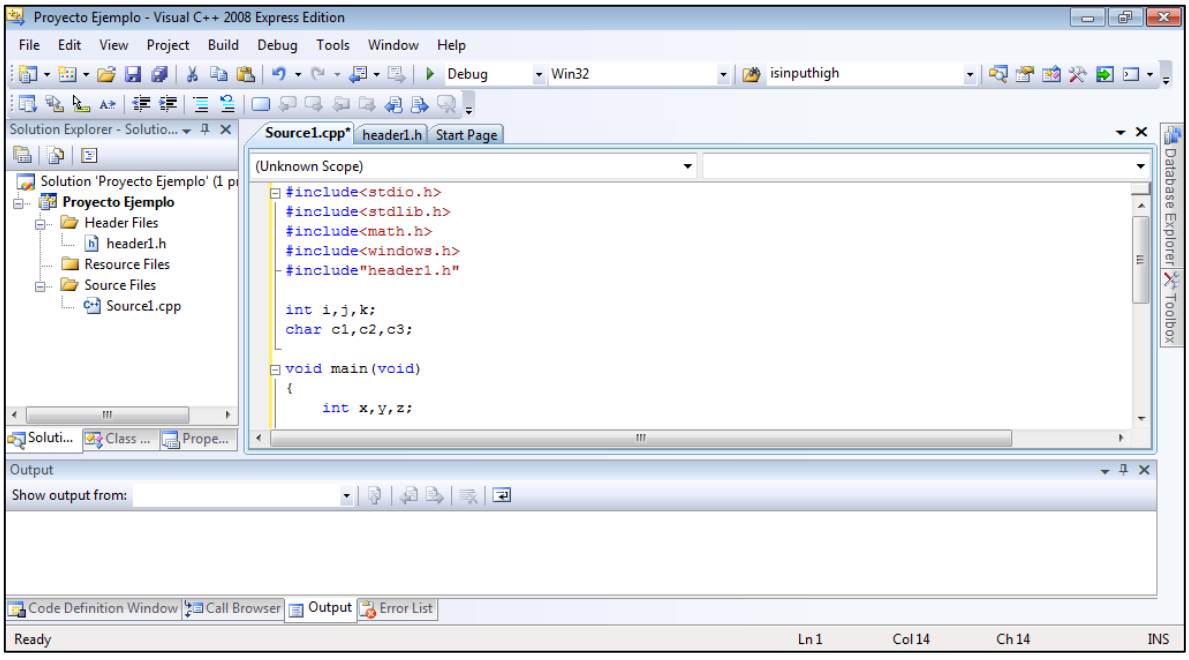


Figura 6.1. Ventana principal de un proyecto en Microsoft Visual C++ 2008 Express Edition

Para comenzar con la programación se utilizó una plantilla proporcionada en la asignatura de Computación Gráfica que se imparte en la Facultad de Ingeniería. En esta plantilla se pueden encontrar las funciones básicas para la ejecución de animaciones con la biblioteca OpenGL, así como las principales funciones para la creación de elementos dentro de la aplicación gráfica, utilizando vectores, modelos, texturas, etc.

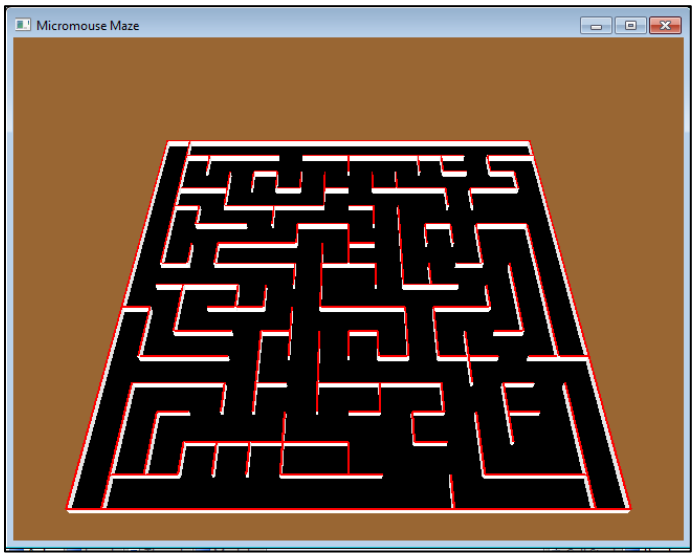


Figura 6.2. Visualización del laberinto creado con especificaciones del estándar del IEEE

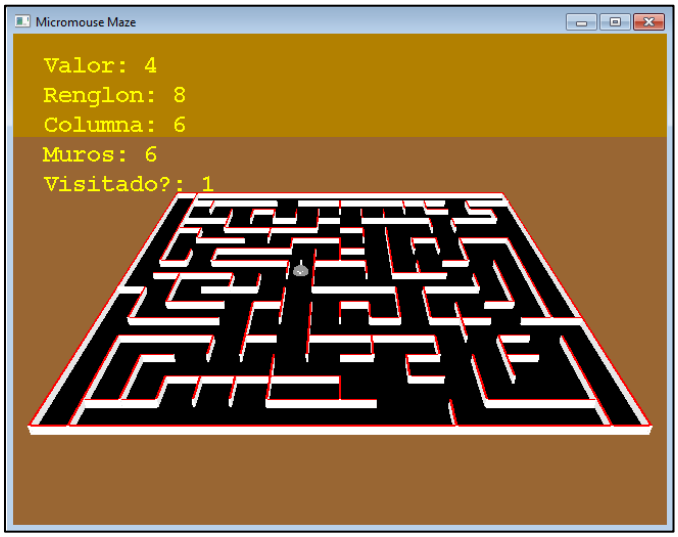


Figura 6.3. Simulación de MicroMouse (vista lateral)

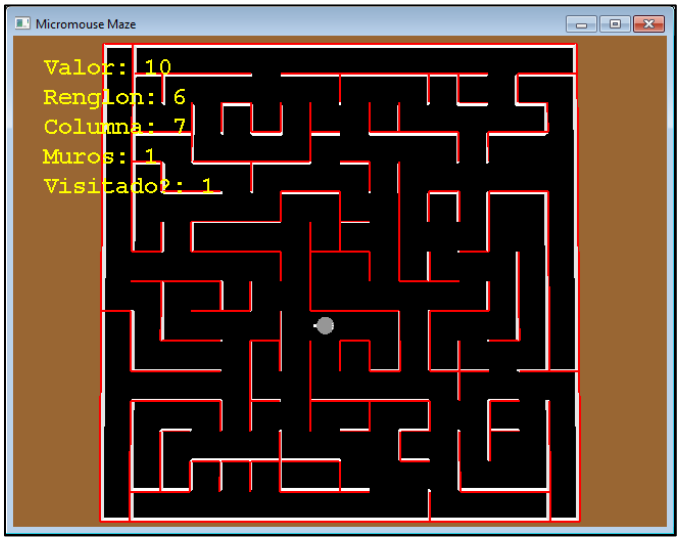


Figura 6.4. Simulación de MicroMouse (vista superior)

El modelo de MicroMouse usado para la simulación fue creado con figuras primitivas con la aplicación 3DSMax.

Algunas de las funciones más representativas son las siguientes:

Inicializar laberinto

```
void InicializaLaberinto()
{
	int i,j;
	for(i=0;i<Orden_L;i++)
		for(j=0;j<Orden_L;j++)
		{
			if(i==0)
				celda[i][j].muros=celda[i][j].muros|0x08;
			else if(i==Orden_L-1)
				celda[i][j].muros=celda[i][j].muros|0x01;
			if(j==0)
				celda[i][j].muros=celda[i][j].muros|0x04;
			else if(j==Orden_L-1)
				celda[i][j].muros=celda[i][j].muros|0x02;
		}
}
```

La función *InicializaLaberinto* nos ayuda a inicializar el mapa de las celdas del laberinto en memoria. Lo importante es que se construyen los bordes del laberinto de acuerdo al orden que se le haya especificado en la constante simbólica *Orden_L*.

Calcular celdas adyacentes

```
void CalculaAdyacentes(unsigned char renglon,unsigned char columna,unsigned char pared)
{
```

```
	unsigned char pared_aux;
	pared_aux=pared&0x08;
	if(	(pared_aux!=0x08) && (renglon-1>=0 && celda[renglon-1][columna].calculado==false) && (!celda[renglon-1][columna].en_cola) ) //Calcula adyacente hacia abajo
	{
		celda[renglon-1][columna].valor=celda[renglon][columna].valor+1;
		celda[renglon-1][columna].calculado=true;
		celda[renglon-1][columna].en_cola=true;
		almacena(renglon-1,columna);
	}

	pared_aux=pared&0x01;
	if(	(pared_aux!=0x01) && (renglon+1<=Orden_L-1 && celda[renglon+1][columna].calculado==false) && (!celda[renglon+1][columna].en_cola) ) //Calcula adyacente hacia arriba
	{
		celda[renglon+1][columna].valor=celda[renglon][columna].valor+1;
		celda[renglon+1][columna].calculado=true;
		celda[renglon+1][columna].en_cola=true;
		almacena(renglon+1,columna);
	}
	pared_aux=pared&0x04;
	if(	(pared_aux!=0x04) && (columna-1>=0 && celda[renglon][columna-1].calculado==false) && (!celda[renglon][columna-1].en_cola) ) //Calcula adyacente hacia la izquierda
	{
		celda[renglon][columna-1].valor=celda[renglon][columna].valor+1;
		celda[renglon][columna-1].calculado=true;
		celda[renglon][columna-1].en_cola=true;
		almacena(renglon,columna-1);
	}
	pared_aux=pared&0x02;
	if(	(pared_aux!=0x02) && (columna+1<=Orden_L-1 && celda[renglon][columna+1].calculado==false) && (!celda[renglon][columna+1].calculado) ) //Calcula adyacente hacia la derecha
	{
		celda[renglon][columna+1].valor=celda[renglon][columna].valor+1;
		celda[renglon][columna+1].calculado=true;
		celda[renglon][columna+1].en_cola=true;
		almacena(renglon,columna+1);
	}
}
```

Esta función llamada *CalculaAdyacentes* es la que está definida en la *Sección 5.2.3* de *Implementación* de *Software* de este trabajo. Apoya en el cálculo de los costos de las celdas al hacer la inundación de valores.

Flood-Fill

```
void FloodFill()
{
	int i,j; //Definir variables locales para esta función
	inicializa_cola();
	//Inicializar valores de las celdas
	for(i=0;i<Orden_L;i++)
		for(j=0;j<Orden_L;j++)
```

```
        {
            celda[i][j].valor=255;
            celda[i][j].calculado=false;
            celda[i][j].en_cola=false;
        }
    //Comienza la inundación dando el valor de cero a la solución
    if(!regreso)
        if(Orden_L%2==1)    //En caso de que el orden del laberinto sea impar (una celda de solución)
        {
            celda[SolR][SolC].valor=0;
            celda[SolR][SolC].calculado=true;
            celda[SolR][SolC].en_cola=true;
            almacena(SolR,SolC);
        }
        else                //En caso de que el orden del laberinto sea par (4 celdas de solución)
        {
            celda[SolR][SolC].valor=0;
            celda[SolR][SolC].calculado=true;
            celda[SolR][SolC].en_cola=true;
            almacena(SolR,SolC);

            celda[SolR][SolC2].valor=0;
            celda[SolR][SolC2].calculado=true;
            celda[SolR][SolC2].en_cola=true;
            almacena(SolR,SolC2);

            celda[SolR2][SolC].valor=0;
            celda[SolR2][SolC].calculado=true;
            celda[SolR2][SolC].en_cola=true;
            almacena(SolR2,SolC);

            celda[SolR2][SolC2].valor=0;
            celda[SolR2][SolC2].calculado=true;
            celda[SolR2][SolC2].en_cola=true;
            almacena(SolR2,SolC2);
        }
    else
    {
        celda[Ren0][Col0].valor=0;
        celda[Ren0][Col0].calculado=true;
        celda[Ren0][Col0].en_cola=true;
        almacena(Ren0,Col0);
    }

    //Inunda las celdas adyacentes
    while(recupera()!=NULL)
    {
        Indice[0]=cola[recover-1][0];    //Renglón
        Indice[1]=cola[recover-1][1];    //Columna
        CalculaAdyacentes(Indice[0],Indice[1],celda[Indice[0]][Indice[1]].muros);
    }
    return;
}
```

Esta función llamada *FloodFill* también está definida en la *Sección 5.2.3* del presente trabajo. Se utiliza para calcular los costos de cada celda con respecto al centro del laberinto. Esta función es la base para nuestro algoritmo.

6.2 Pruebas de Hardware

Primeramente, quise realizar una prueba independiente de cada sensor utilizado en el modelo para conocer su funcionamiento con el microcontrolador elegido.

Sensores infrarrojos

Para los sensores infrarrojos, opté por realizar el circuito llamado *Side Looking Sensors*[23] que resulta una buena opción por ser simple y económica. Las pruebas permitieron reconocer el funcionamiento de estos sensores.

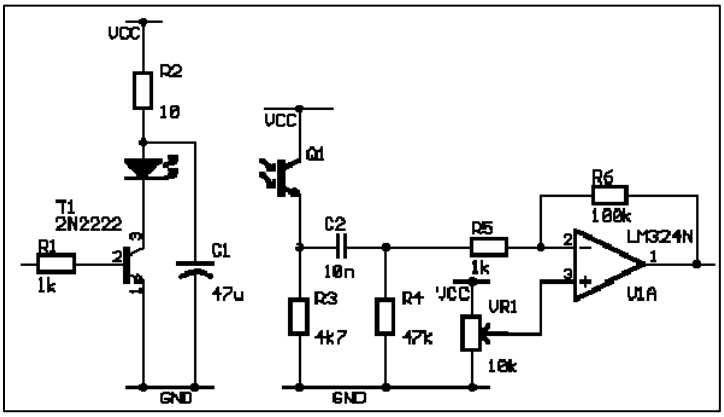


Figura 6.5. Diagrama de conexiones del circuito *Side Looking Sensors*

Este circuito permite que, a partir de una señal digital a cierta frecuencia, se pueda obtener una señal analógica. En otras palabras, prendiendo y apagando un LED infrarrojo y que su luz se refleje en alguna superficie a altas velocidades, el receptor facilitará una señal que depende de la cantidad de luz infrarroja reflejada que haya podido percibir, cuanto más alejada se encuentre la superficie, la luz percibida será menor. Lo que quiere decir que puede hacerse una relación de luz percibida por la distancia a la que se encuentre el objeto.

Para hacer la lectura de la señal analógica, será necesario implementar el Convertidor Analógico/Digital del microcontrolador. El PIC18F4520 cuenta con hasta 13 canales de éstos [66], se utilizarán únicamente 3 para los sensores infrarrojos de muros.

El código que fue utilizardo para realizar las lecturas de los sensores infrarrojos es el siguiente:

[23] El circuito puede encontrarse en la siguiente página URL, que también abarca información relevante acerca de los robots MicroMouse. Fue usado por el Ing. Moisés Meléndez en [65]:
http://www.micromouseonline.com/micromouse-book/sensors/side-looking-sensors/

```
Sensado()
{
      bitset(PORTA,3);            //Enciende el LED infrarrojo 1
      Delay10TCYx(8);       //retraso de 10 uS
      CONVERAD(ADC0); //Recibe dato del sensor al canal ADC0
      sensor_izq=LEEADC();      //p*0.0041992*10000; //Calcula voltaje a resolución de 10 bits
      bitclr(PORTA,3);            //Apaga el LED infrarrojo 1
      bitset(PORTA,4);            //Enciende el LED infrarrojo 2
      Delay10TCYx(8);       //retraso de 10 uS
      CONVERAD(ADC1); //Recibe dato del sensor al canal ADC1
      sensor_fro=LEEADC();      //p*0.0041992*10000; //Calcula voltaje a resolución de 10 bits
      bitclr(PORTA,4);            //Apaga el LED infrarrojo 2
      bitset(PORTA,5);            //Enciende el LED infrarrojo 3
      Delay10TCYx(8);       //retraso de 10 uS
      CONVERAD(ADC2); //Recibe dato del sensor al canal ADC2
      sensor_der=LEEADC();      //p*0.0041992*10000; //Calcula voltaje a resolución de 10 bits
      bitclr(PORTA,5);            //Apaga el LED infrarrojo 3
      Delay1KTCYx(56);      //retardo de 7 ms
}
```

La función de *Sensado* hace posible obtener una señal de los canales analógicos al sensar los muros. Los datos de cada sensor son guardados en variables globales independientes para su procesamiento posterior en el programa.

Brújula digital

En el caso del magnetómetro se utiliza el modelo LSM303DLHC en una tarjeta portadora diseñada y elaborada por la empresa Pololu (ver *Figura 6.6*). El LSM303DLHC es un circuito integrado fabricado por STMicroelectronics y es un magnetómetro que también contiene un acelerómetro (haciendo las lecturas pertinentes en el dispositivo se obtiene la información que corresponde al acelerómetro).

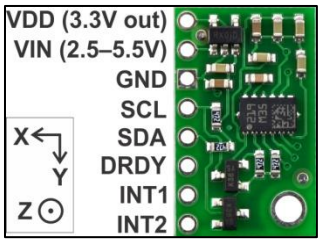


Figura 6.6. Tarjeta portadora del circuito integrado LSM303DLHC

El funcionamiento de este circuito integrado es logrado a través del protocolo I^2C. Contiene registros que pueden ser de lectura y escritura. Los registros de escritura son comúnmente utilizados para configuración del dispositivo, por ejemplo, resolución, ganancia magnética (en el caso del magnetómetro), sensibilidad de aceleración (para el acelerómetro), etc. Los registros de lectura, en cambio, son utilizados para obtener información acerca del

dispositivo, incluyendo la información leída de un campo magnético o de la aceleración lineal. Toda la información del dispositivo puede revisarse en la hoja de datos correspondiente [67].

```
void LeerBrujula(void)
{
    BSTART();               //Condición de inicio
    TX_BYTES(COM_W_CON);    //Prepara para lectura escribiendo la dirección del registro inicial a
leer 0x3C
    TX_BYTES(OUT_X_H_M); //Transmite la dirección del registro OUT_X_H_M
    BRESTART();             //Condición de inicio repetido en I2C
    TX_BYTES(COM_R_CON);    //Se transmite la dir. del magnetómetro con el bit de lectura 0x3D
    bitclr(SSPCON2,BACKDT);     //Acknowledge del dispositivo 'master' (PIC)
    RX_BYTES();             //Recibe el contenido de OUT_X_H_M (X parte alta)
    buffer[0]=datoi;            //Almacena en buffer X parte baja
    bitclr(SSPCON2,BACKDT);     //Acknowledge del dispositivo 'master' (PIC)
    RX_BYTES();             //Recibe el contenido de OUT_X_L_M (X parte baja)
    buffer[1]=datoi;            //Almacena en buffer X parte alta
    bitclr(SSPCON2,BACKDT);     //Acknowledge del dispositivo 'master' (PIC)
    RX_BYTES();             //Recibe el contenido de OUT_Z_H_M (Z parte alta)
    buffer[2]=datoi;            //Almacena en buffer Z parte alta
    bitclr(SSPCON2,BACKDT);     //Acknowledge del dispositivo 'master' (PIC)
    RX_BYTES();             //Recibe el contenido de OUT_Z_L_M (Z parte baja)
    buffer[3]=datoi;            //Almacena en buffer Z parte baja
    bitclr(SSPCON2,BACKDT);     //Acknowledge del dispositivo 'master' (PIC)
    RX_BYTES();             //Recibe el contenido de OUT_Y_H_M (Y parte alta)
    buffer[4]=datoi;            //Almacena en buffer Y parte alta
    bitset(SSPCON2,BACKDT);     //Acknowledge del dispositivo 'master' (PIC)
    RX_BYTES();             //Recibe el contenido de OUT_Y_L_M (Y parte baja)
    buffer[5]=datoi;            //Almacena en buffer Y parte baja
    BSTOP();            //Termina la transmission por I2C
}
```

La función *LeerBrujula* está implementada para realizar la lectura de los registros *X, Y, Z*, del magnetómetro. Es importante mencionar que los datos obtenidos se consideran crudos, al ser datos que no dan mucha información acerca de la orientación del dispositivo. Para conocer la orientación del dispositivo, es necesario procesar la información obtenida, con la siguiente función:

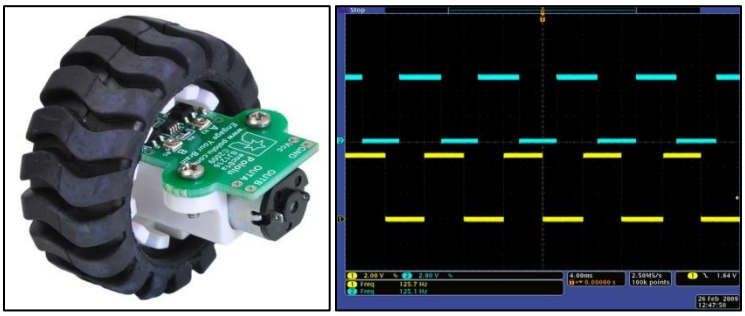
```
void Orientacion(void)
{
    LeerBrujula();              //Obtiene información de la brújula
    m.x=((int)buffer[0]<<8)|(int)buffer[1];     //Guarda X en una variable de 16 bits
    m.y=((int)buffer[4]<<8)|(int)buffer[5];     //Guarda Y en una variable de 16 bits
    m.z=((int)buffer[2]<<8)|(int)buffer[3];     //Guarda Z en una variable de 16 bits

    //De acuerdo a los valores obtenidos en X y Y, calcula la orientación
      if(m.x==0 && m.y<0)
```

```
        angulo=90;
    else if(m.x==0 && m.y>0)
        angulo=270;
        else if(m.x!=0)
            angulo=180-atan2(-m.y,m.x)*180/PI;
}
```

La función *Orientacion* calcula, como su nombre lo dice, la orientación del robot MicroMouse dentro del laberinto. Esto con respecto al campo magnético terrestre, ya que una brújula está basada en eso.

Encoders

Por último, los encoders que se probaron fueron unos proporcionados por Pololu, que cuentan con salida A y B, es decir, en cuadratura (*ver Figura 6.7b*), y que tienen un total de 48 pulsos por vuelta (12 en cada salida y 48 al obtener la relación de cuadratura). Estos encoders están situados en el eje del motor justo después del motorreductor y antes de la llanta, pues están diseñados específicamente para el chassis y las llantas utilizadas (ver *Figura 6.7a*). Esto es una desventaja, ya que no se puede adaptar a otro tipo de neumático, además, está diseñado para cierto tipo de motores (micromotores de Pololu).



(a) (b)

Figura 6.7. (a) Chassis con micromotor, llanta y encoder[24] y (b) salidas de encoder en cuadratura[25]

Para la implementación de los encoders en el robot, se ha diseñado el código que logra capturar los pulsos generados por el encoder. Cabe mencionar que la relación de cuadratura necesariamente debe hacerse por software, así que dentro del código se puede ver cómo funciona.

Otra situación que requiere especial atención es el uso de la interrupción por cambio de estado en el puerto B del PIC16F4520 [66] para la captura de los pulsos de los encoders. Esta interrupción indica al microcontrolador cuándo existe un pulso y dentro de ella se analiza qué tipo de pulso fue generado, si fue positivo o negativo.

[24] Este mecanismo de funcionamiento es proveído por la empresa Pololu.
[25] Imagen tomada de la página web de Pololu.

```
void RUTINTH (void)  //rutina de interrupcion de alta prioridad
{
    unsigned char masI,menosI,masD,menosD;
    bitclr(INTCON,BRBIE);           //Deshabilita interrupción para atenderla
    motor1a=PORTB&0b10000000; //Captura el bit más significativo del puerto B
    motor1b=PORTB&0b01000000; //Captura el segundo bit más significativo del puerto B
    motor2a=PORTB&0b00100000; //Captura el tercer bit más significativo del puerto B
    motor2b=PORTB&0b00010000; //Captura el cuarto bit más significativo del puerto B
    bitclr(INTCON,BRBIF);           //Deshabilita la bandera de interrupción
      //Opera el estado anterior con el estado actual para identificar hacia dónde giró el motor
    masI=motor1a^(ant_m1b<<1);   //El valor será diferente de cero si hubo pulso positivo
    menosI=motor1b^(ant_m1a>>1);       //El valor será diferente de cero si hubo pulso positivo
    masD=motor2a^(ant_m2b<<1); //El valor será diferente de cero si hubo pulso positivo
    menosD=motor2b^(ant_m2a>>1);       //El valor será diferente de cero si hubo pulso positivo
    if(masI)    //Si hubo pulso positivo en motor izquierdo
        pulsosI++;
    if(menosI) //Si hubo pulso negativo en motor izquierdo
        pulsosI--;
    if(masD)   //Si hubo pulso positivo en motor derecho
        pulsosD++;
    if(menosD)//Si hubo pulso negativo en motor derecho
        pulsosD--;
    ant_m1a=motor1a;    //Guarda el estado actual en el anterior
    ant_m1b=motor1b;    //Guarda el estado actual en el anterior
    ant_m2a=motor2a;    //Guarda el estado actual en el anterior
    ant_m2b=motor2b;    //Guarda el estado actual en el anterior
      //Las siguientes comparaciones aplican solamente para la función MoverMotores.
      //Cuando el número de pulsos de cada motor alcanza el especificado en la función MoverMotores,
      //los motores se desactivan
    if(pulsosI==avance_izquierda)
    {
        bitset(PORTD,0);      //Pone en alto el primer pin de polarización del motor
        bitset(PORTD,1);      //Pone en alto el segundo pin de polarización del motor
        //Cuando los dos pines de polarización se encuentran en alto, el motor se detiene
    }
    if(pulsosD==avance_derecha)
    {
        bitset(PORTD,2);      //Pone en alto el primer pin de polarización del motor
        bitset(PORTD,3);      //Pone en alto el segundo pin de polarización del motor
    }
    bitset(INTCON,BRBIE);     //Vuelve a habilitar interrupción para continuar su uso
}
```

Como base para crear el código que procesa los pulsos, fueron revisadas las bibliotecas para el uso de los motores con encoders, creadas por Pololu y para un microcontrolador diferente.

Otras funciones de apoyo para el funcionamiento de los encoders y además, de los motores, son las siguientes:

```
//Función para mover los motores
void MoverMotores(int motor_izq,int motor_der)
{
	unsigned char auxPD;
	pulsosI=0;		//inicializa conteo de pulsos en motor izquierdo
	pulsosD=0;			//inicializa conteo de pulsos en motor derecho
	avance_derecha=motor_der;	//cantidad de pulsos del motor derecho
	avance_izquierda=motor_izq;	//cantidad de pulsos del motor izquierdo
	//Define la polarización del motor izquierdo, de acuerdo a los pulsos que se quieren
	if(motor_izq>=0)
		PORTD=0x02;			//hacia adelante
	else
		PORTD=0x01;			//hacia atrás
	//Define la polaridad del motor derecho, de acuerdo a los pulsos que se quieren
	if(motor_der>=0)
		PORTD|=0x08; //hacia adelante
	else
		PORTD|=0x04; //hacia atrás
	GenerarPWM(40,40); //Envía señal PWM
	while(PORTD!=0x0F);	//Espera hasta que los motores se detengan con la rutina de interrupción
}
```

La función *MoverMotores* define la dirección de giro de los motores. Recibe como parámetros dos datos, el primero es el número de pulsos para el motor izquierdo y el segundo es el número de pulsos para el motor derecho.

```
//Función para realizar los giros del robot
void Gira(char dir) //Si dir=0 => giro izquierda, si no, giro derecha
{
	GenerarPWM(0,0);	//detiene la señal PWM
	pulsosI=0;		//Inicializa pulsos del motor izquierdo
	pulsosD=0;			//Inicializa pulsos del motor derecho
	if(!dir)	//Si la dirección de giro es derecha
	{
		MoverMotores(VUELTA,-VUELTA);
	}
	else
	{
		MoverMotores(-VUELTA,VUELTA);
	}
	PORTD=0x0F;		//Desactiva motores
	Orientacion();		//Calcula la orientación actual, después del giro
	angulo0=angulo;			//Almacena la orientación actual
}
```

La función *Gira* es efectuada para realizar giros de 90°, ya sea a la derecha o a la izquierda. Recibe como parámetro un valor, el cual es el que dirá si el giro es a la izquierda o a la derecha, precisamente. Cuando dicho valor sea cero, el robot girará a la izquierda, de lo contrario, será a la derecha.

6.3 Pruebas de integración del sistema (Software y Hardware)

Una vez finalizadas las pruebas del software y del hardware del sistema, es hora de integrar los componentes. Cabe mencionar que el código del programa de simulación no se puede transferir directamente al microcontrolador. Sin embargo, debido a que está realizado en lenguaje C y el microcontrolador es programado en lenguaje C, no se complica demasiado el paso del programa.

6.3.1 Implementación

Realmente, el código para el microcontrolador con respecto al de la simulación hecha en Visual C++ no varía del todo, salvo por lo que es la parte mecánica. En la simulación no se tienen sensores, motores, encoders, y demás dispositivos de hardware necesarios para el robot, de manera que hay que programar su funcionamiento.

La función principal, o *main*, estará definida por el siguiente diagrama de flujo:

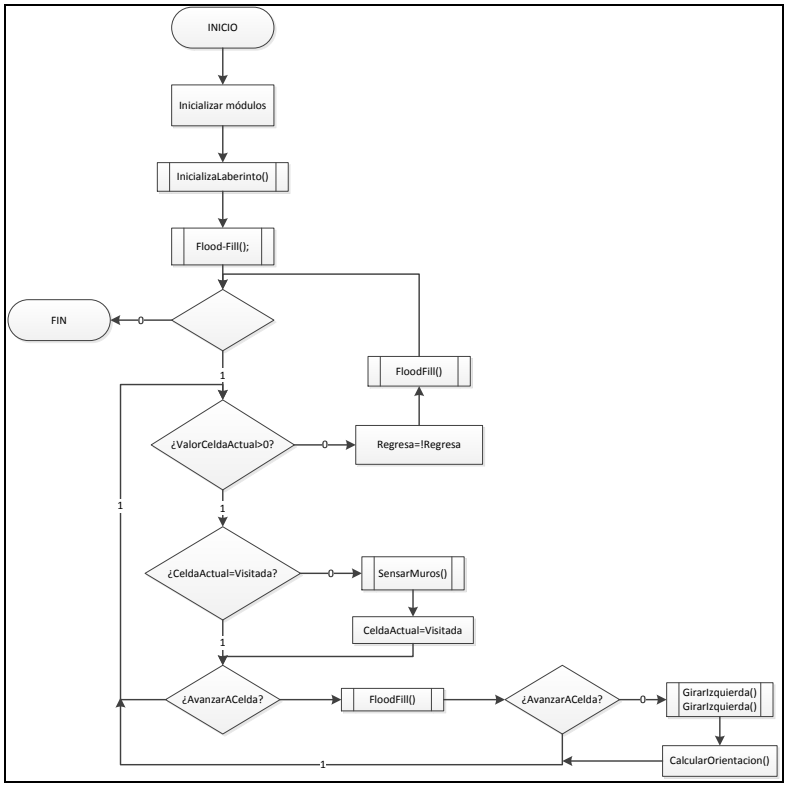


Figura 6.8. Diagrama de flujo para la función *main*

El código de la función *main* está dado a continuación:

```
void main (void)
{
	PLLOSC();	// activa pll para operar a 32 mhz
	InicializaPuertos();//Inicializa puertos paralelos
	InicializaSerie();	//Inicializa puerto Serie a 115200 bauds/s
	RET100MS(); //Retardo de 100 ms
	InicializaI2C();  //Inicializa MSSP como I2C a 100 kHz RC3=SCL RC4=SDA
	//Inicializa variables de estado de encoders con la lectura actual de los encoders
	ant_m1a=PORTB&0b10000000;
	ant_m1b=PORTB&0b01000000;
	ant_m2a=PORTB&0b00100000;
	ant_m2b=PORTB&0b00010000;
	InicializaPWM2();//Inicializa PWM para dos señales (a la misma frecuencia)
	InicializaADC();//Inicializa Convertidor A/D (agregar al utilizar los sensores infrarrojos)
	InicializaLaberinto();//Inicializa el mapa del laberinto
	FLOODFILL();//Calcula los valores de cadacelda
	Orientación=90;
	p=&celda[0][0];//El acceso a las celdas del laberinto es de manera indirecta
	pv=&valor_celda[0][0];//El acceso a las celdas del laberinto también se hace indirecto
	bitset(INTCON,BGIE);	//Habilita interrupciones generales
	bitset(INTCON,BPEIE);	//Habilita interrupciones de periférico
	bitset(INTCON,BRBIE);	//Habilita interrupción por cambio de estado en RB4-RB7
	//Configuración del circuito LSM303DLHC
	BSTART();//Señal de inicio para comunicación I2C
	TX_BYTES(COM_W_CON);//Escritura en brújula
	TX_BYTES(0x00);//Dirección de registro CRA_REG_M (se incrementa a partir del siguiente dato transmitido)
	TX_BYTES(0x14);//Sensor de temperatura dehabilitado, Data Output Rate: 30 Hz (CRA_REG_M)
	TX_BYTES(0x20);//Rango de campo: +-1.3 gauss; Ganancia X,Y:1100 LSB/gauss Ganancia Z:980 LSB/gauss (CRB_REG_M)
	TX_BYTES(0x00);//Modo de operación de magnetómetro (brújula): ContinuousMode (MR_REG_M)
	BSTOP();//Señal de paro
	while(1)
	{
		Orientacion();	//Calcula la orientación de la brújula
		angulo0=angulo; //Guarda el valor de orientación
		while(*pv>0)		//Mientras no se ubique en la solución
		{
			if(!(*p&0x10))	//Si la celda actual no ha sido visitada
			{
				SensarMuros();
				*p|=0x10;//bit de celda visitada
			}
			if(!AvanzarACelda())  //Si no se puede mover a la siguiente celda
			{
				FLOODFILL();//Recalcula costos de celdas
```

```
                    if(!AvanzarACelda()) //Si aún no se puede mover a la siguiente celda retrocede una
celda
                    {
                        Gira(IZQUIERDA);//Giro de 90 grados
                        RET100MS();
                        Gira(IZQUIERDA);//Giro de 90 grados. Dio medio vuelta
                        orientacion+=180;//
                        orientacion%=360;
                    }//Fin if
                }//Fin if
            }//Fin while
            regresa=!regresa; //Una vez que se resolvió el laberinto, se cambia el sentido de avance
            FLOODFILL();  //Calcula nuevos valores con el objetivo cambiado
        }//Fin while
}//Fin main
```

Implementando las funciones básicas definidas en la *Sección 5.2.3*, se obtiene el código siguiente:

```
void CalculaCeldasAdyacentes(unsigned char *ap_celda,unsigned char *ap_valor)
{
    unsigned int ultima_celda;
    ultima_celda=ORDEN_C*ORDEN_R;
    ultima_celda+=0x300;
    //Revisa el adyacente hacia abajo
    If    (
            ((int)ap_celda-ORDEN_R)>=0x300 && //Si la celda actual está en el rango del laberinto
            !((*ap_celda)&0x08) &&   //Si la celda no tiene muro hacia abajo
            !((*(ap_celda-ORDEN_R))&0x20) && //Si la celda no está en cola
            !((*(ap_celda-ORDEN_R))&0x40) //Si la celda no ha sido calculada
            )
    {
        //Calcula el valor de la celda adyacente al sur
            *(ap_valor+((int)ap_celda-ORDEN_R-0x300))=(*(ap_valor+((int)ap_celda-0x300)))+1;
        *(ap_celda-ORDEN_R)|=0x60;   //Celda sur calculada y en cola
        Almacenar(ap_celda-ORDEN_R);       //Se almacena a la cola
    }
    //Revisa adyacente hacia arriba
    If    (
            (((int)ap_celda+ORDEN_R)<ultima_celda) && //Si la celda actual está en el rango del
    laberinto
            !((*ap_celda)&0x01) &&          //Si no hay muro al norte
            !((*(ap_celda+ORDEN_R))&0x20) &&      //Si la celda no está encola
            !((*(ap_celda+ORDEN_R))&0x40)     //Si no ha sido calculada
            )
    {
        //Calcula el valor de la celda adyacente al norte
            *(ap_valor+((int)ap_celda+ORDEN_R-0x300))=(*(ap_valor+((int)ap_celda-0x300)))+1;
        *(ap_celda+ORDEN_R)|=0x60;  //celda norte calculada y en cola
```

```
        Almacenar(ap_celda+ORDEN_R);      //almacena celda
    }
    //Revisa adyacente a la izquierda
    If    (
            (((int)ap_celda-0x300)%ORDEN_C)!=0 && //Si la celda actual está en el rango
            !((*ap_celda)&0x04) &&          //Si no hay muro al oeste
            !((*(ap_celda-1))&0x20) &&      //Si no está en cola
            !((*(ap_celda-1))&0x40)         //Si no ha sido calculada
            )
    {
        //Calcula el valor de la celda adyacente al oeste
            *(ap_valor+((int)ap_celda-0x300)-1)=(*(ap_valor+((int)ap_celda-0x300)))+1;
        *(ap_celda-1)|=0x60; //celda oeste calculada yen cola
        Almacenar(ap_celda-1);      //se almacena en cola
    }
    //Revisa adyacente a la derecha
    If    (
            (((int)ap_celda-0x300)%ORDEN_C)+1<ORDEN_C && //Si la celda actual está en el
      rango
            !((*ap_celda)&0x02) &&          //Si no hay muro al este
            !((*(ap_celda+1))&0x20) &&      //Si no está en cola
            !((*(ap_celda+1))&0x40)   //Si no ha sido calculada
            )
    {
        //Calcula valor de la celda adyacente al este
        *(ap_valor+((int)ap_celda-0x300)+1)=(*(ap_valor+((int)ap_celda-0x300)))+1;
        *(ap_celda+1)|=0x60;        //celda este calculada y en cola
        Almacenar(ap_celda+1);    //se almacena en cola
    }
}
```

6.3.2 Pruebas

Las pruebas de integración han ayudado a revisar algunos detalles. Por ejemplo, la necesidad de utilizar un procedimiento para poder alinear al robot dentro del laberinto fue un detalle que pudo observarse. De momento, se implementó un control P para poder dar un poco de mayor precisión, utilizando la lectura de los encoders y la lectura de los sensores. La función realizada para dicho efecto es la siguiente:

```
//Función para alinear al robot mientras avanza (utilizando la lectura de los encoders)
void AlinearRobot3()
{
    int izq,der;
    //Si existe muro a los lados y no se realiza giro, entonces alinea al robot respecto de los muros
    if(sensor_izq<MURO_I && sensor_der<MURO_D && giro)
    {
        izq=40+sensor_izq-sensor_der;
```

```
            der=40+sensor_der-sensor_izq;
        }
        //Si no hay muro en alguno de los lados de la celda actual o se está realizando un giro
        else
        {
            izq=40+Abs(pulsosD)-Abs(pulsosI);
            der=40+Abs(pulsosI)-Abs(pulsosD);
        }
        GenerarPWM(izq,der);        //Genera la señal PWM a través de las dos salidas utilizadas
}
```

También, por el momento, en las pruebas no está implementada la función para almacenar la información del laberinto en memoria EEPROM. El fin es utilizar los 256 bytes disponibles de esta memoria para guardar el mapa del laberinto y así cargar dicha información a la memoria de datos y poder hacer uso de la información cuando comience el programa.

6.4 Resultados

El comportamiento del algoritmo ha sido el deseado. Las pruebas realizadas en el programa de simulación del robot MicroMouse fueron existosas, puesto que permiten observar que el algoritmo se está comportando de la manera correcta. Sin embargo, existen deficiencias en el código y se puede optimizar de muchas maneras. Mientras, el programa funciona satisfactoriamente.

En el caso de la transición al hardware, el algoritmo sigue funcionando, no así el robot. Para que la solución del laberinto se pueda observar correctamente, es necesario que el robot MicroMouse tenga el comportamiento deseado. Me refiero a que el robot tiene defectos al avanzar en línea recta. Lo idóneo es que el robot logre avanzar en línea recta, lo cual difícilmente sucede aunque se haya implementado el algoritmo para alinear el robot. Por consiguiente, el robot choca con las paredes y se pierde por completo. Otra situación importante de considerar son los giros, que de igual manera no se completan de la manera correcta. Para corregir esto es necesario que el algoritmo utilizado para que el avance en línea recta funcione, puesto que ayudaría a corregir el error acumulado al hacer el giro. Otra manera de resolver ese problema sería utilizando mejores motores, con mucho mayor precisión. Si es posible tener llantas delgadas con encoder, de preferencia con buena resolución (arriba de 200 pulsos por revolución), mejoraría el funcionamiento.

Analizando la complejidad del algoritmo, puedo concluir que en espacio, se comporta con un orden de crecimiento de $O(n^2)$, donde n es el orden del laberinto. De hecho, la función de espacio respecto al orden del laberinto estaría compuesta más o menos así: $S(n)=2n^2+c$, porque el número de celdas del laberinto es nxn, pero será el doble debido a que una localidad almacena el costo de la celda y la otra para almacenar el mapa del laberinto; la

variable *c* en la función estará definida por el número de variables utilizadas en el programa. En cuanto a tiempo, el algoritmo para la búsqueda de la solución tiene orden de crecimiento de $O(b^n)$ en el peor caso, donde *n* sigue siendo el orden del laberinto, *b* es el número de niveles del laberinto y en el que 8 es el mínimo y el máximo ronda los 250.

El comportamiento de la búsqueda de la solución en este algoritmo, y en la manera en que lo utilicé, es voraz; como ya lo había mencionado, es parecido a la búsqueda *Greedy Best-First Search*. Una desventaja de este algoritmo radica en que no encuentra la mejor solución en la primera navegación en el laberinto. Sólo irá expandiendo nodos como vaya encontrando al de menor costo. Sin embargo, la ventaja que tiene es que encontrará la solución más rápida, ya que siempre estará avanzando. Si necesitara retroceder para buscar otro camino, tomaría mucho más tiempo, pues el giro de los motores, además del procesamiento necesario del programa, lo retrasarían.

En cuanto al uso de memoria, con ayuda de MPLAB, se peude observar la cantidad de memoria, tanto de programa como de datos, que se utiliza para el sistema completo. Esto es en el microcontrolador. (*Figura 6.9*)

Con el análisis de memoria utilizada es posible notar que la memoria de programa no alcanza ni siquiera la mitad de su capacidad. Por otro lado, la memoria de datos sí ocupa alrededor del 65% total. En futuras revisiones, estos datos podrían optimizarse.

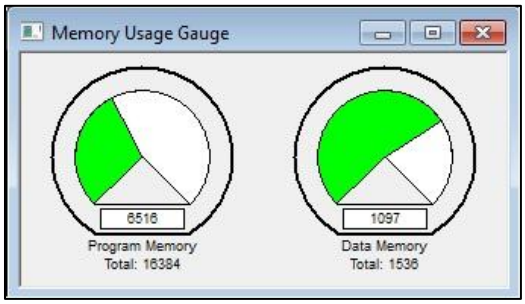


Figura 6.9. Uso de memoria de programa y de datos en el microcontrolador

6.5 Conclusiones

Esta manera de implementar el algoritmo *Flood-Fill* nos da las siguientes conclusiones:

Software

- Este algoritmo difícilmente da la solución óptima en una corrida[26], es decir el camino más corto, sin embargo, encuentra una posible solución. Siendo que en una

[26] Una corrida se refiere al recorrido del robot MicroMouse a través del laberinto hasta que encuentra la solución del laberinto y debe regresar a la celda inicial.

competencia de robots MicroMouse del IEEE se pueden hacer hasta 3 corridas, el robot puede seguir recorriendo el laberinto y así encontrar la solución óptima, pero es complicado que sea en la primera cuando las dimensiones del laberinto son 16x16. En la simulación del robot que se realizó en Visual C++, se encontró la solución óptima en la segunda corrida.

- En principio, se pensó implementar memoria dinámica para determinar la posición del robot dentro del laberinto y para diseñar el laberinto mismo, pero siendo que las dimensiones del laberinto son fijas, no es necesario utilizarla. Aunque en el caso de la simulación por software puede ser útil para el sistema operativo en el que se encuentra ejecutando la simulación, ya que la asignación de memoria para el proceso que ejecuta a la aplicación gráfica puede ser dinámica, al hacer el paso del programa al microcontrolador no es conveniente, puesto que es un sistema embebido que está dedicado a resolver un laberinto de paredes y nada más, así que todos los recursos del microcontrolador estarán dedicados a conseguir su objetivo.
- Se realizaron las acciones de orientación (Norte, Sur, Este y Oeste) por software, lo que hizo más conveniente el problema. Debido a que únicamente son 4 orientaciones diferentes, resulta mejor utilizar una variable que almacene la orientación y que por cada giro dentro del laberinto se haga el cálculo correspondiente. Si se hubiese realizado por hardware, es decir, con ayuda de la brújula digital, el procedimiento para hacer la lectura de la brújula es relativamente tardado, además de hacer los cálculos necesarios para transformar los datos crudos a datos más legibles.
- El programa de simulación en Visual C++ para el robot MicroMouse puede implementarse con otros algoritmos, únicamente haciendo las adaptaciones necesarias.

Hardware

- Es importante considerar la distribución de los componentes dentro del circuito, pues el peso puede afectar el funcionamiento del robot.
- Se utilizó una placa fenólica de doble cara para realizar el circuito impreso y poder ahorrar hardware y espacio. Aunque hubo bastante dificultad al momento de soldar los componentes ubicados en cierta orientación del circuito.
- La brújula tenía interferencias de algún campo magnético presente dentro del circuito que afectaba su funcionamiento, y del cual no pude determinar su origen. Se realizó una extensión, a base de conectores, para alejar la brújula del campo que provocaba dichas interferencias. Para evitar esto, sería conveniente realizar una jaula de Faraday dentro del circuito del robot, en especial alrededor de la brújula, para que no puedan afectar campos magnéticos no deseados y que el sistema no presente lecturas erróneas.

- Podría ser conveniente agregar más sensores infrarrojos al sistema. Permitiría ejecutar la acción de alinear al robot sobre el laberinto de mejor manera.
- El uso de componentes de montaje superficial puede ayudar al ahorro de espacio, incluso permitiría la adición de otros componentes como los sensores. Además, permitiría reducir el tamaño del robot MicroMouse

En general, el algoritmo *Flood-Fill* no solamente se utiliza en sistemas robóticos, pues éste no está centrado hacia la resolución de un laberinto. Otra implementación que ha llegado a tener es, por ejemplo, la herramienta para realizar el relleno de un color en una aplicación de dibujo, como *Paint*, o también en el popular Buscaminas, para ejecutar el barrido de las figuras que pueden eliminarse[27]. Para estos últimos casos no necesariamente es una búsqueda informada, puesto que no requiere usar valores, sino que sólo utiliza celdas contiguas para calcular las acciones necesarias.

El algoritmo *Flood-Fill* como algoritmo de búsqueda es muy conveniente cuando el entorno es completamente conocido. El cálculo de los costos de cada nodo es relativamente sencillo y no requiere de mucho cómputo. El problema está cuando el entorno es desconocido, pues necesitaría realizar la navegación a través de los nodos y esto implicaría hacer una búsqueda voraz, que como ya fue analizado, no nos da la solución óptima. Aunque es posible adaptar la búsqueda para obtener una solución completa y óptima, pero ello involucra más cómputo.

6.6 Trabajo futuro

En cuanto a lo que podría desarrollarse con este algoritmo, el número de ámbitos para introducirlo es amplio. Para empezar, implementarlo en sistemas de Robótica mucho más avanzados podría suponer una aplicación conveniente, como pueden ser sistemas de domótica o en robots de servicio.

Otra aplicación es en Redes de datos para el cálculo del camino más corto a través del cual pueden viajar los datos. Es decir, con ayuda de la información de los ruteadores de red, conocer el tráfico de red (entorno) y realizar el algoritmo de inundación empezando por el nodo objetivo para determinar el mejor camino. Empero, como se ha mencionado, si el entorno no es completamente conocido puede resultar más costoso.

Incluso, una aplicación para sistemas de Cómputo Gráfico, como en el caso de una herramienta de relleno en procesamiento de imágenes.

Otra parte fundamental de la elaboración de esta investigación es fomentar el desarrollo de sistemas robóticos. La elaboración de un sistema de este tipo, y en particular de un robot

[27] *http://en.wikipedia.org/wiki/Flood_fill*

MicroMouse, representa un reto intelectual; en primera, para elaborar una solución al problema; y en segunda, realizar el diseño de un robot que cumpla con las características requeridas.

La última competencia de robots MicroMouse realizada en México fue en el Torneo Mexicano de Robótica del año 2011[28] y, aparentemente, no tuvo éxito. En ediciones posteriores del TMR (hasta 2013), la categoría de laberinto MicroMouse no fue considerada.

[28] *http://tmr.itam.mx/*

REFERENCIAS

[1] F. A. MARTÍNEZ GIL y G. M. Quetglás, Introducción a la programación estructurada en C, Valencia: Educació. Materials, 2003, pp. 266.

[2] F. ALONSO AMO, L. Martínez Normand y F. J. Segovia Pérez, Introducción a la Ingeniería del Software. Modelos de desarrollo de programas., Madrid: Delta Publicaciones, 2005.

[3] «RoboCup 2012. Mexico City,» 2012. [En línea]. Available: http://www.robocup2012.org/. [Último acceso: 6 Junio 2013].

[4] The RoboCup Federation, «About RoboCup,» 2013. [En línea]. Available: http://www.robocup.org/. [Último acceso: 24 Marzo 2013].

[5] «RoboCup 2013,» 2013. [En línea]. Available: http://www.robocup2013.org/robocup-soccer/. [Último acceso: 6 Junio 2013].

[6] Escuela Universitaria de Ingeniería Técnica Industrial de Zaragoza, «Historia de la Ingeniería de Control. Autómatas en la Historia,» 2002. [En línea]. Available: http://automata.cps.unizar.es/Historia/Webs/automatas_en_la_historia.htm. [Último acceso: 21 Mayo 2013].

[7] U. NEHMZAW, Mobile Robotics. A practical introduction, Segunda ed., Springer-Verilag London, 2003, pp. 280.

[8] Conductix-Wampfler, «Factory Automation,» [En línea]. Available: http://www.wampfler.com. [Último acceso: 6 Junio 2013].

[9] Factronics Systems Engineering Pte Ltd, «Manufacturing & Logistics Handling Systems,» 2004. [En línea]. Available: http://www.factronics.com.sg/p_mfg_agv.htm. [Último acceso: 6 Junio 2013].

[10] Robots Dreams, «MicroMouse,» 2011. [En línea]. Available: http://www.robots-dreams.com/micromouse. [Último acceso: 6 Junio 2013].

[11] IEEE UCSD, «Home: California Micromouse,» 2010. [En línea]. Available: http://ieee.ucsd.edu/micromouse/. [Último acceso: 24 Marzo 2013].

[12] IEEE UCSD, «California Micromouse Competition - 2011,» 2011. [En línea]. Available: http://ieee.ucsd.edu/micromouse/files/micromouse_rules2010.pdf. [Último acceso: 24 Marzo 2013].

[13] S. RUSSELL y P. Norvig, Artificial Intelligence. A Modern Approach, Segunda ed., Prentice-Hall, 2003, pp. 1081.

[14] J. P. MÜLLER, The Design of Intelligent Agents. A Layered Approach, Springer-Verilag Berlin, 1996, pp. 227.

[15] R. J. SCHALKOFF, Intelligent Systems. Principles, paradigms, and pragmatics., Sudbury, Massachusetts: Jones and Bartlett Publishers, 2011, pp. 758.

[16] G. TECUCI, Building Intelligent Agents, Gran Bretaña: Academic Press, 1998, pp. 320.

[17] D. C. AHOGADO ÁLVAREZ y A. M. Reinemer Valencia, «Programación Orientada a Agentes: Metodologías de Desarrollo de Software,» Junio 2002. [En línea]. Available: http://pegasus.javeriana.edu.co/~poa/Documentos/Articulo_J4.doc. [Último acceso: 19 Mayo 2013].

[18] Y. SHOHAM, «Agent-Oriented Programming,» de *Artificial Intelligence*, Universidad de Stanford, California, 1993, pp. 51-92.

[19] B. HENDERSON-SELLERS y P. Giorgini, Agent-Oriented Methodologies, Londres: Idea Group Inc., 2005, pp. 413.

[20] B. BAUER y J. Odell, «UML 2.0 and agents: how to build agent-based systems with the new UML standard,» Marzo 2005. [En línea]. Available: http://www.jamesodell.com/EAAI-Bauer-Odell.pdf. [Último acceso: 19 Junio 2013].

[21] P. BRESCIANI, A. Perini, P. Giorgini, F. Giunchiglia y J. Mylopoulos, «Tropos: An Agent-Oriented Software Development,» de *Autonomous Agents and Multi-Agent Sytems*, The Netherlands, Kluwer Academic Publishers, 2004, pp. 203-236.

[22] Universiteit van Amsterdam, «CommonKADS. General Info,» [En línea]. Available: http://www.commonkads.uva.nl/frameset-commonkads.html. [Último acceso: 8 Julio 2013].

[23] M. MORENO ESPINO, A. Rosete Juárez, A. Simón Cuevas, R. Valdés González, E. Leyva Pérez, R. Socorro, J. Pina Amargós y A. García Fernández, «Artículo: "Ingeniería de Software Orientada a Agentes: Roles y Metodologías",» Julio 2006.

[En línea]. Available: http://rii.cujae.edu.cu/index.php/revistaind/article/view/118/95. [Último acceso: 10 Julio 2013].

[24] D. KNUTH, The art of computer programming. Volume 1: Fundamental Algorithms, Segunda ed., Addison-Wesley, 1973.

[25] R. GUEREQUETA y A. Vallecillo, Mayo 2000. [En línea]. Available: http://www.lcc.uma.es/~av/Libro/. [Último acceso: 7 Abril 2013].

[26] S. HARRIS y J. Ross, Beginning Algorithms, Wiley Publishing, Inc, 2005.

[27] E. ESPINOSA ÁVILA, «Notación Asintótica,» 2013. [En línea]. Available: http://dicyg.fi-c.unam.mx:8080/lalo/news/notacion-asintotica/image/image_view_fullscreen. [Último acceso: 6 Junio 2013].

[28] P. P. GUILLÉN Hernández, «Análisis de Complejidad,» 2011. [En línea]. Available: http://pier.guillen.com.mx. [Último acceso: 18 Marzo 2013].

[29] F. HANIK, «The Kiss Principle,» 2013. [En línea]. Available: http://people.apache.org/~fhanik/kiss.html. [Último acceso: 18 Marzo 2013].

[30] A. FELNER, «Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm,» de *The Fourth International Symposium on Combinatorial Search (SoCS-2011)*, 2011.

[31] P. GUPTA, V. Agarwal y M. Varshney, «Design and Analysis of Algorithms,» PHI Learning Private Limited, Nueva Delhi, 2008.

[32] F. E. VALDÉS-PÉREZ y R. Palla-Areny, Microcontrollers: fundamentals and applications with PIC, CRC Press, 2009, pp. 297.

[33] E. SANTAMARÍA NAVARRETE, Electrónica digital y microprocesadores, Madrid, Universidad Pontificia de Comillas: Colección ingeniería, 1993, pp. 322.

[34] D. RAVICHANDRAN, Introduction to Computers and Communication, Tata McGraw-Hill Publishing Company Ltd., 2001.

[35] L. NULL y J. Lobur, The Essentials of Computer Organization and Architecture, Jones & Bartlett Publishers, 2010, pp. 844.

[36] I.T.L. Education Solution Limited, Itl, Introduction to Information Technology,

Pearson Education India, 2005, pp. 668.

[37] National Instruments, «Documentos de Soporte. Comunicación Serial: Conceptos Generales,» 2012. [En línea]. Available: http://mexico.ni.com/. [Último acceso: 29 Marzo 2013].

[38] M. M. MANO, Arquitectura de Computadoras, Tercera ed., México, D.F.: PEARSON EDUCACIÓN, 1994, pp. 563.

[39] Microchip Technology Inc., «DS33023A: PICmicro™ Mid-Range MCU Family. Reference Manual,» Diciembre 1997. [En línea]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/33023a.pdf. [Último acceso: 10 Mayo 2013].

[40] Microchip Technology Inc., «SPI.Overview and Use of the PICmicro Serial Peripheral Interface,» [En línea]. Available: http://ww1.microchip.com/downloads/en/devicedoc/spi.pdf. [Último acceso: 7 Mayo 2013].

[41] Phillips Semiconductors, «UM10204. I2C-bus specification and user manual. Rev. 03,» 2007.

[42] J. C. CAMPELO RIVADULLA, F. Rodríguez Ballester y V. Carot, Periféricos e Interfaces Industriales, U. P. d. Valencia, Ed., Valencia: Servicios de Publicaciones, 1997, pp. 383.

[43] A. J. KHAMBATA, Microprocessors/Microcomputers. Architecture, Software, and Systems, Segunda ed., John Wiley & sons, Inc, 1987.

[44] D. CALCUTT, F. Cowan y H. Parchizadeh, 8051 Microcontrollers: An Applications Based Introduction (Google eBook), Newnes, 2004, pp. 416.

[45] B. B. BREY, Microprocesadores Intel, Séptima ed., México, D.F.: PEARSON EDUCACIÓN, 2006, pp. 912.

[46] ENP 9 UNAM, «Uso de sensores: Estrategias para el uso de sensores.,» [En línea]. Available: http://www.prepa9.unam.mx/academia/cienciavirtual/sensores.htm. [Último acceso: 10 Mayo 2013].

[47] M. BRAIN, «How Electric Motors Work,» HowStuffWorks, Inc., 2013. [En línea]. Available: http://electronics.howstuffworks.com/motor.htm. [Último acceso: 24 Mayo 2013].

[48] C. M. TROUT, Essentials of Electric Motors and Controls, Jones and Bartlett Publishers, 2010, pp. 118.

[49] J. RECIO MIÑARRO, «MAGNETISMO,» 19 Marzo 2013. [En línea]. Available: http://www.quimicaweb.net/grupo_trabajo_fyq3/index.htm. [Último acceso: 9 Junio 2013].

[50] M. d. C. REGAL FERNÁNDEZ, «Inducción Eletromagnética,» 2010. [En línea]. Available: http://recursostic.educacion.es/eda/web/eda2010/newton/materiales/regal_fernandez_carmen_p3/Induccion_electromagnetica/guion_induccion.html. [Último acceso: 9 Junio 2013].

[51] J. TAGÜEÑA y E. Martina, «De la brújula al espín. El magnetismo,» 2013. [En línea]. Available: http://bibliotecadigital.ilce.edu.mx/sites/ciencia/volumen2/ciencia3/056/htm/sec_3.htm. [Último acceso: 9 Junio 2013].

[52] Wikipedia, «Electroimán,» [En línea]. Available: http://es.wikipedia.org/wiki/Electroimán. [Último acceso: 9 Junio 2013].

[53] W. SHEPHERD, L. Hulley y D. Liang, Power electronics and motor control, Segunda ed., Cambridge: Cambridge University Press, 1995, pp. 539.

[54] H. A. TOLIYAT y G. B. Kliman, Handbook of electric motors, Segunda ed., Marcel Dekker, 2004, pp. 805.

[55] MICROMO, «DC Motor Application Considerations,» [En línea]. Available: http://www.micromo.com/dc-motor-application-considerations.aspx. [Último acceso: 13 Abril 2013].

[56] J. WEBB y K. Greshock, Industrial Control Electronics, Segunda ed., Merrill Publishing Company, 1993, pp. 666.

[57] V. MAZZONE, «Controladores PID,» Marzo 2002. [En línea]. Available: http://www.eng.newcastle.edu.au/~jhb519/teaching/caut1/Apuntes/PID.pdf. [Último acceso: 24 Junio 2013].

[58] G. ELLIS, Control System Design Guide, San Diego, California: Elsevier Academic Press, 2004, pp. 464.

[59] E. S. YU, «Social Modeling and i*,» de *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos*, Springer-Verlag Berlin Heidelberg, 2009, pp. 99-121.

[60] P. GIORGINI, «Tropos: basics,» Noviembre 2009. [En línea]. Available: http://www.dit.unitn.it/~pgiorgio. [Último acceso: 8 Julio 2013].

[61] M. L. MORALES RODRÍGUEZ, «Agentes y Arquitecturas de Control para Robots Autónomos Móviles,» [En línea]. Available: http://www.academia.edu/1185430/Agentes_y_Arquitecturas_de_Control_para_Robots_Autonomos_Moviles. [Último acceso: 22 Julio 2013].

[62] R. PALLÁS ARENY, Instrumentos Electrónicos Básicos, Marcombo, 2006, pp. 317.

[63] D. PARDO COLLANTES y L. A. Bailón Vega, Fundamentos de electrónica digital, Salamanca: Ediciones Universidad de Salamanca, 2006, pp. 276.

[64] F. C. FITCHEN, Circuitos integrados y sistemas, Barcelona: Ed. Reverté, S. A., 1975, pp. 461.

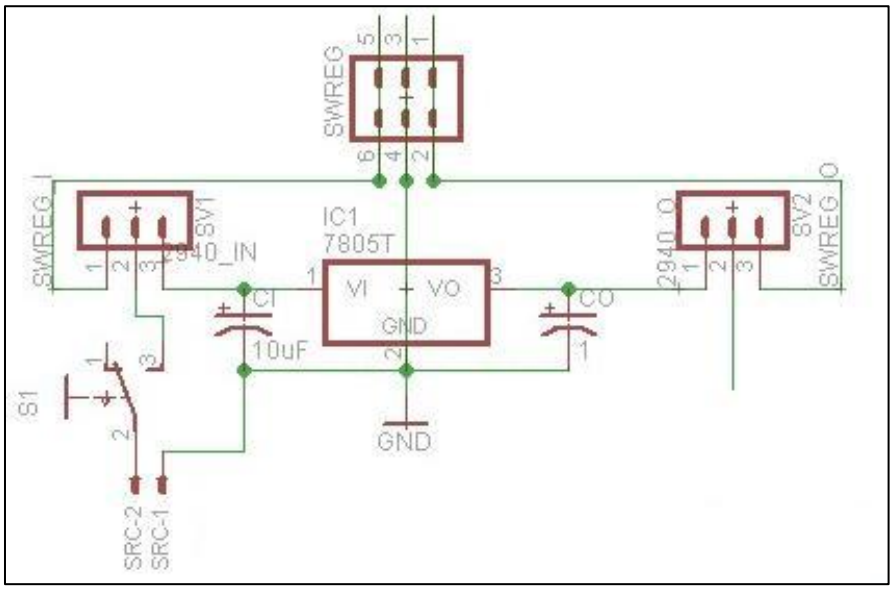
[65] M. MELÉNDEZ REYES, *Tesis: Diseño e implementación de un robot resolvedor de laberintos de paredes*, México, D.F., 2011, pp. 152.

[66] Microchip Technology Inc., «PIC18F2420/2520/4420/4520 Data Sheet,» EE.UU., 2008.

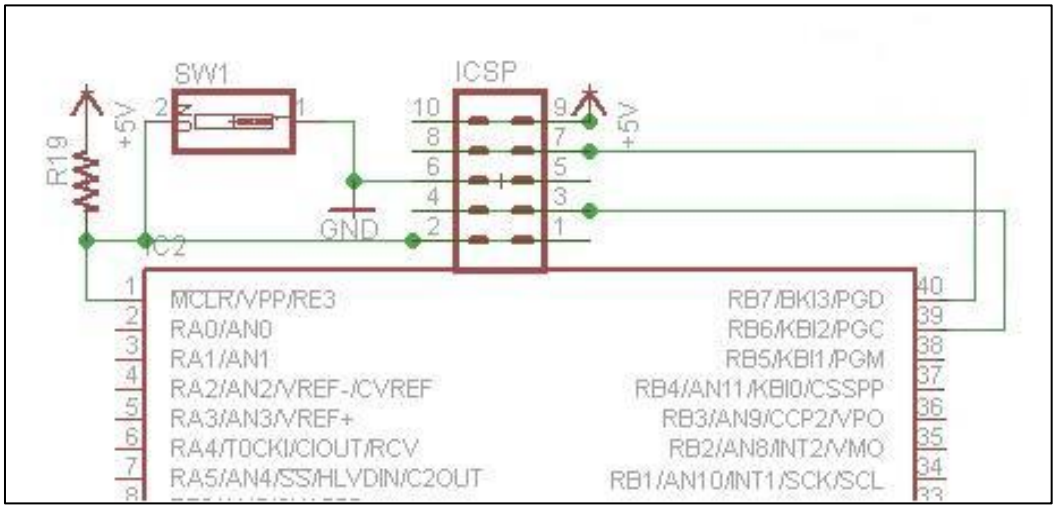
[67] STMicroelectronics, «LSM303DLHC Datasheet,» 2011.

Apéndice A. Diagramas esquemáticos

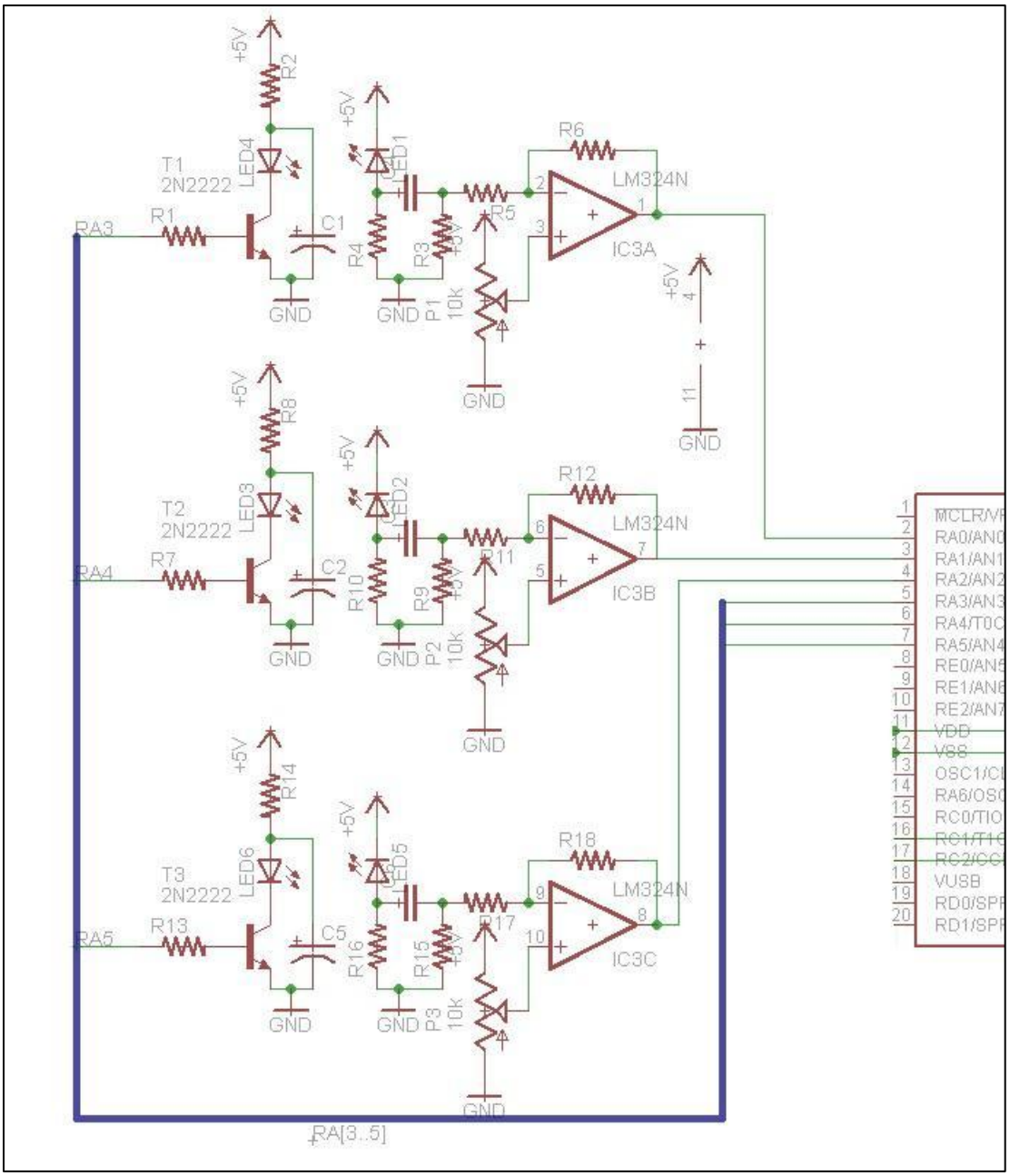
Fuente de alimentación



In-Circuit Serial Programming

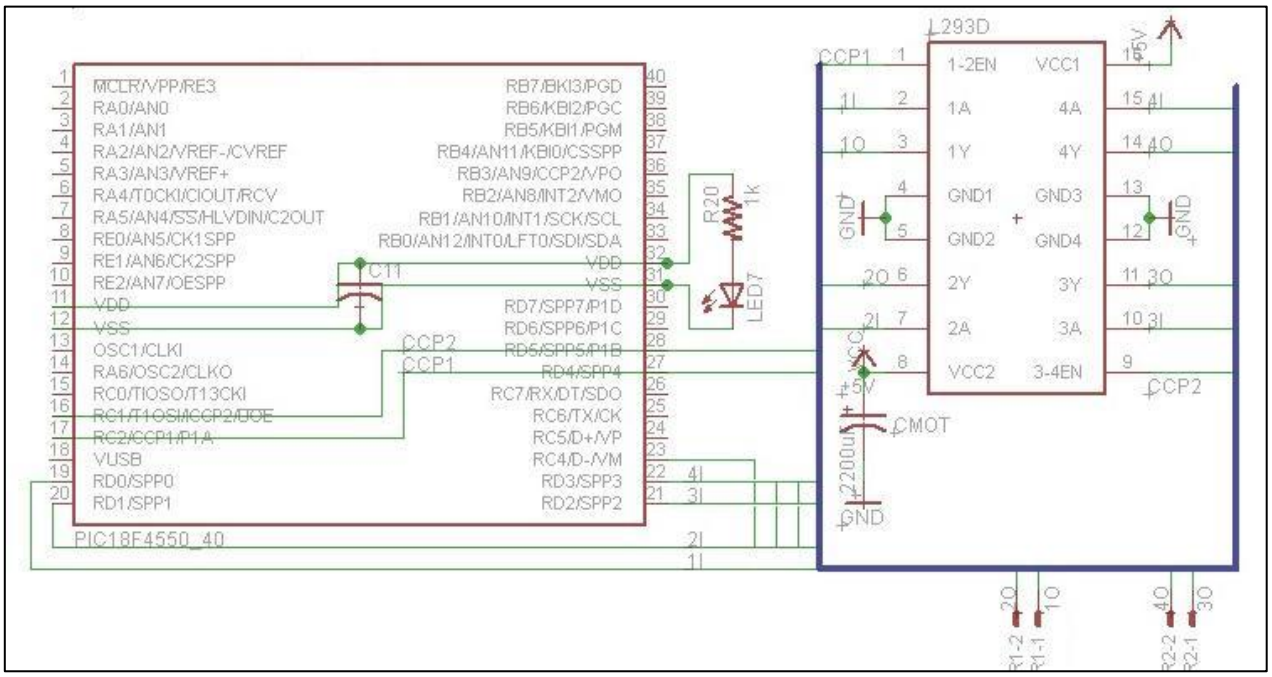


Sensores de muros infrarrojos

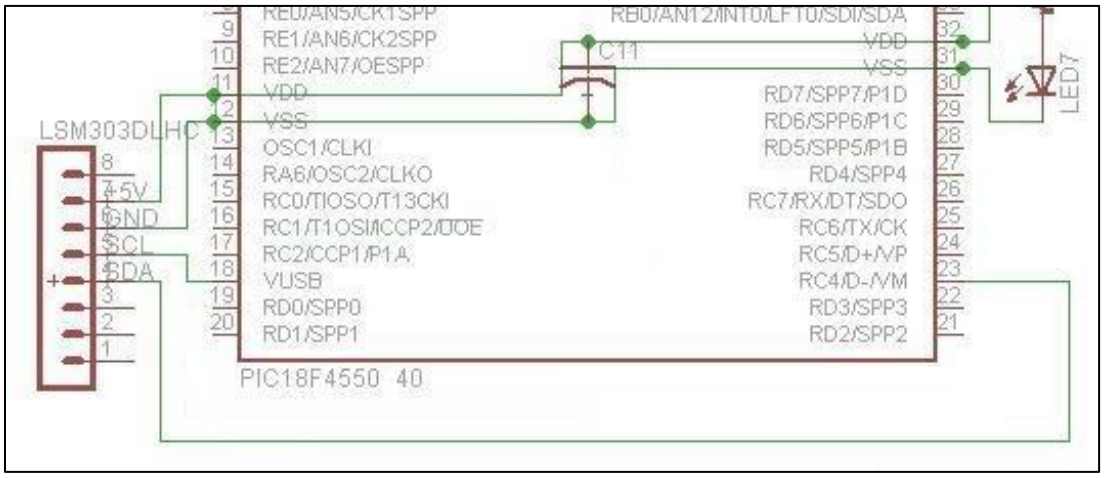


Conexión para los encoders

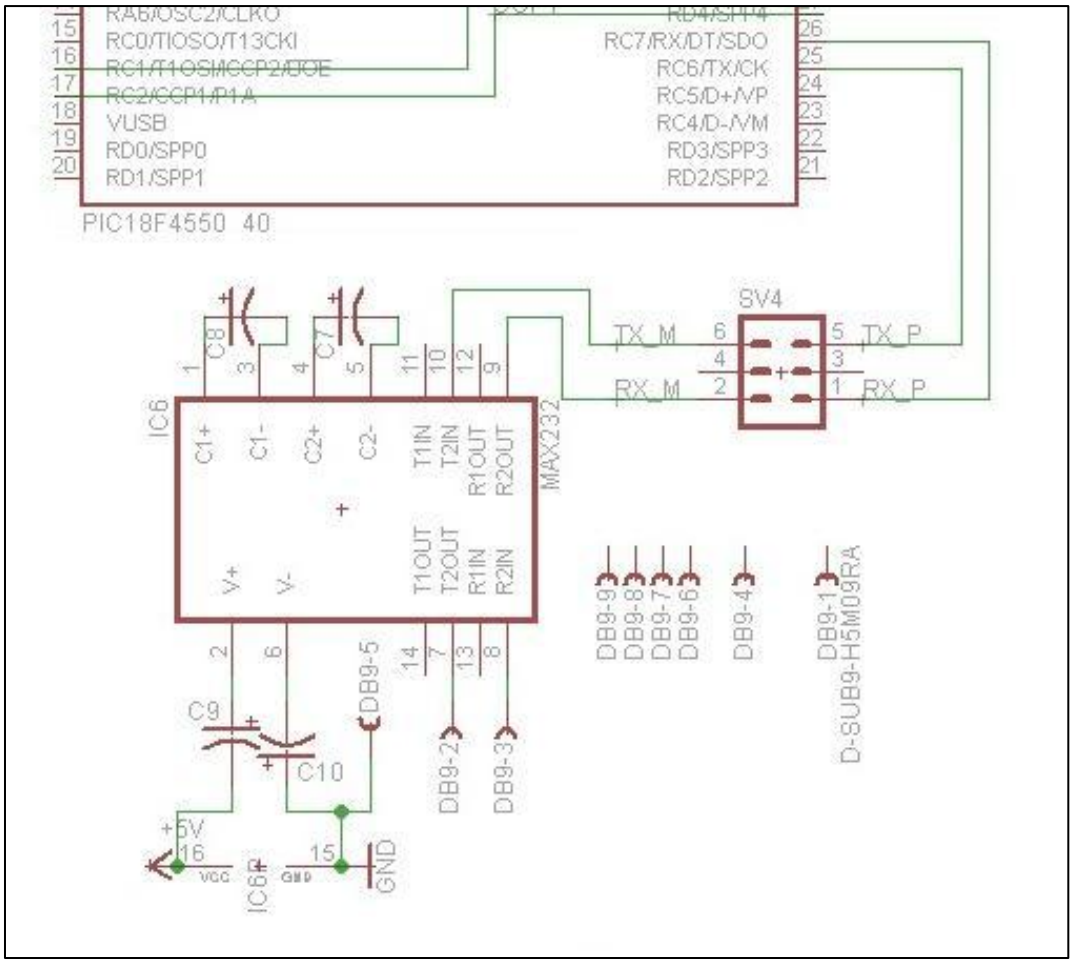
Driver L293D



Conexión para la brújula digital LSM303DLHC



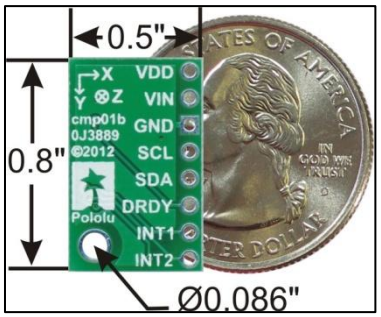
Módulo RS-232



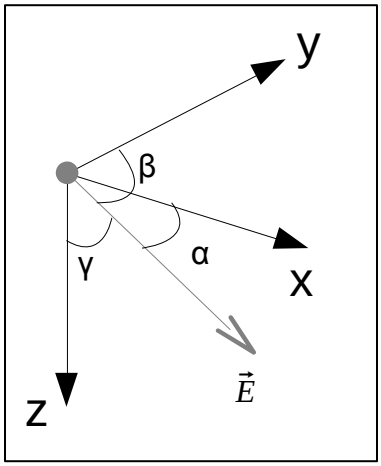
Apéndice B. Cálculo de la orientación de la brújula digital

El cálculo de la orientación de la brújula utilizando el circuito LSM303DLHC no es complicado. Simplemente se necesitan algunos conceptos de Geometría Analítica.

Consideremos que la referencia de la brújula está dada como se muestra en la figura siguiente, en la parte superior izquierda. La brújula estará posicionada de manera que la parte positiva del eje *z* esté orientada hacia abajo y la parte positiva del eje *x* se dirija hacia adelante del robot.

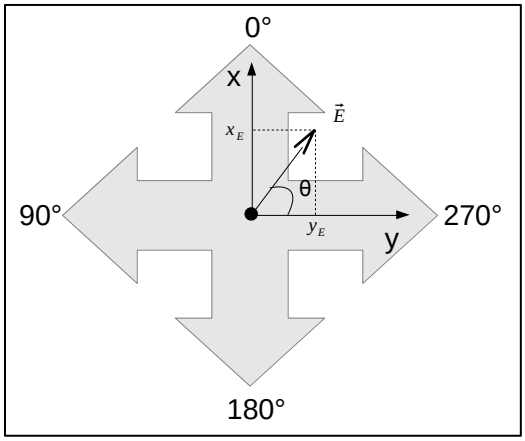


Existe el vector $\vec{E}$ que representa el campo magnético de la Tierra que va hacia el Polo Norte (esto se debe a que nos encontramos situados en el Hemisferio Norte). Este vector tiene componentes en los 3 ejes de las coordenadas cartesianas. Los ángulos α, β y γ son los ángulos que forma el vector respecto a los ejes *x*, *y* y *z*, respectivamente.



Es importante mencionar que la lectura que nos da la brújula digital son las componentes del vector en cada uno de los ejes cartesianos, es así que lo que se obtiene del registro X de la brújula será un valor x_E, lo del registro Y será un valor y_E, y lo que se lea del registro Z será un valor z_E. Lo que quiere decir que el vector $\vec{E} = (x_E, y_E, z_E)$

Se considera también que la referencia en *z* es constante, puesto que el robot se mueve sobre un plano. Esto implica que el análisis se realiza en el plano cartesiano *XY* y se simplifica el cálculo, ya que no será necesario tomar en cuenta la componente en *z* del vector.



La fórmula para calcular el ángulo del vector $\vec{E}$ está dada por:

$$\theta = \left(180 - \tan^{-1}\frac{-y_E}{x_E}\right) \times 180/\pi;\ x_E \neq 0$$

El rango de valores de $\tan^{-1}\frac{-y_E}{x_E}$ va de $(-180,0) \cup (0,180)$

Sin embargo, cuando $x_E = 0$, $\theta \rightarrow \infty$. Por lo que se hacen las siguientes comparaciones.

Si $x_E = 0$ y $y_E < 0$, entonces $\theta = 90°$. Si $x_E = 0$ y $y_E > 0$, entonces $\theta = 270°$.

Apéndice C. Lista de precios

Componente	Cantidad	Precio unitario	Precio total
Microcontrolador PIC18F4520	1	80,00	80,00
Driver L293D (Puente H)	1	40,00	40,00
Amplificador LM324	1	5,00	5,00
Fototransistor PT331C	3	5,00	15,00
LED Infrarrojo IR333	3	5,00	15,00
Resistencia var. Preset 10 KΩ	3	8,00	24,00
Transistor 2N2222A	3	10,00	30,00
Resistor 1 KΩ	6	0,20	1,20
Resistor 100 KΩ	3	0,20	0,60
Resistor 47 KΩ	3	0,20	0,60
Resistor 4.7 KΩ	3	0,20	0,60
Resistor 10 Ω	3	0,20	0,60
Capacitor electrolítico 47 µF	3	1,00	3,00
Capacitor cerámico 10 nF	3	0,50	1,50
Brújula digital LSM303DLHC	1	340,00	340,00
Kit rueda+encoder	2	680,00	1360,00
Micromotor 50:1	2	270,00	540,00
SINTRA	1	35,00	35,00
Placa fenólica de doble cara 10x10 cm	1	15,00	15,00
Header macho (tira de 40 pines)	1	6,00	6,00
Header hembra (tira de 40 vías)	1	20,00	20,00
Tira de 40 pines hembra maquinada	2	25,00	50,00
Regulador de voltaje a 5V LM2940	1	25,00	25,00
Capacitor electrolítico 10 µF	5	1,00	5,00
Capacitor electrolítico 1 µF	1	1,00	1,00
Capacitor electrolítico 1000 µF	1	6,00	6,00
MAX232N	1	14,00	14,00
Conector DB9 Hembra	1	9,00	9,00
Molex 2 pines (completo)	1	7,00	7,00
Portapilas (4 AA)	1	8,00	8,00
Broche para portapilas	1	5,00	5,00
Pila alcalina AA	4	10,00	40,00
		Total[29]	$ 2703,10	$ 214,67 (USD)
		Pesos/Dólar	$ 12,5919	Al 16 de Julio de 2013

[29] El valor total es aproximado