Aplicación de Técnicas de Robótica e Inteligencia Artificial sobre un robot móvil utilizando el entorno de programaciónVisual Studio.Net

Directores: **Ing. Claudio Verrastro, Ing. Roberto Barneda**
Revisión: **Ing. Juan Carlos Gómez**

MÓDULO 1: "Aplicando un algoritmo de búsqueda"

Autores: **Carlos Fernando Carmona, Damián De Biase, Cesar Foti, Daniel López Amado, Sebastián Verrastro**

Índice general

Tablas

Figuras

1 Introducción

Se presenta un conjunto de prácticas orientadas a la aplicación de técnicas de Inteligencia Artificial en Robótica.
Para el desarrollo de estas prácticas se utiliza la plataforma de programación *Visual Studio.NET* de *Microsoft®* y un robot *ER1* de *Evolution Robotics®.*
(http://www.evolution.com/er1/) (http://msdn.microsoft.com/vstudio/)
Se trabaja con una PC portátil (notebook), montada sobre el robot, con un módulo de comunicación inalámbrico (Wi-Fi). La ejecución, sobre la PC portatil, del programa suministrado por *Evolution Robotics*, llamado RCC (Robot Control Center), permite controlar todas las funciones del robot y sus accesorios: cámaras y pinzas. Este control puede realizarse en forma local o remota , vía Internet, a traves de otra computadora utilizando comunicación interproceso (**Socket**). Esta última es la modalidad elegida para el desarrollo de las prácticas. Se crean entonces aplicaciones propias que ejecutadas en otras PCs permiten acceder al control del robot.

SOCKET:
Un socket es un identificador para un servicio concreto en un nodo concreto de la red. El socket consta de una dirección de nodo y de un número de puerto que identifica al servicio

2 Objetivos

El objetivo de este módulo es aplicar un **algoritmo** de búsqueda **heurística** para que el robot pueda efectuar tareas como encontrar el camino más corto para recorrer una serie de puntos y evitar obstáculos. Para llegar a este objetivo se realizarán una serie de prácticas previas, que consisten en:

- Configuración del programa del robot para su comunicación con el programa de control.
- Conexión al mismo desde un ordenador remoto,
- Envío de comandos simples para moverlo.
- Ejecución de una secuencia de comandos de una lista.
- Construcción de una grilla para dibujar **waypoints**.
- Visita de waypoints y evasión de obstáculos predefinidos en la grilla.

Estas prácticas se acompañan con ejemplos y ejercicios, además se vinculan de manera tal que al final de este módulo se obtenga un programa integrado.

ALGORITMO:
Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema.

HEURÍSTICA:
Técnica de la indagación y del descubrimiento. En algunas ciencias, manera de buscar la solución de un problema mediante métodos no rigurosos, como por tanteo, reglas empíricas, etc.

WAYPOINT:
Punto fijo de una ruta marcado sobre un plano.

3 Requisitos fundamentales para la realización de las prácticas

Para la realización de las prácticas que se presentan a continuación se debe contar con el siguiente equipamiento:

- Al menos un robot ER1 armado.
- Una PC portátil[1], la cual será montada sobre el robot, dotada de 2 puertos USB (como mínimo) y tecnología Wi-Fi, Blue tooth u otra.
- Un ordenador con acceso a Internet, desde el cual se controlará al robot.
- El robot debe ser armado siguiendo las instrucciones del manual que viene con el equipo.
- El software que viene en el kit debe ser instalado en ambos ordenadores (control y remoto).
- Se debe controlar que las baterías de la PC y del robot se encuentren cargadas.

[1] La PC portátil (notebook, tablet PC...) maneja su propia batería, la del robot es un módulo metálico con interruptor y terminal de carga.

4 Práctica N°1: Configuración del Software RCC (Robot Control Center)

Autores: **Carlos Fernando Carmona, Damián De Biase, Cesar Foti**

4.1 Desarrollo de la Práctica

IP:
El Protocolo Internet (IP) es un protocolo de conmutación de paquetes que realiza direccionamiento y encaminamiento. Cuando se transmite un paquete, este protocolo añade una cabecera al paquete. Además, IP es el responsable del empaquetado y división de los paquetes requerido por los niveles físico y de enlace de datos del modelo OSI. Cada paquete IP está compuesto por una dirección de origen y una de destino, un identificador de protocolo, un checksum (un valor calculado) y un TTL (tiempo de vida)

Una característica de este robot, es que su programa permite conectarse a él vía Internet y se le pueden enviar comandos que ya están predefinidos. Para operar el robot desde un ordenador remoto, se debe conocer la dirección de **IP** y el número de puerto de la PC portátil que está sobre el robot. En nuestro caso utilizaremos el puerto 9000 que viene por defecto en el RCC. (los puertos del 0 al 1023, están reservados para otras aplicaciones).

Para la configuración del software RCC, en primer lugar se debe tener instalado el mismo la PC portátil, luego con esta encendida enchufar a los puertos USB, la cámara y el módulo de control del ER-1. Se podrá observar que la PC detecta un nuevo hardware y requiere que se instalen los drivers correspondientes. Los mismos se encuentran en el disco en el cual viene el software RCC, dentro de las carpetas “CameraXP” y “USB Robot Control Module” respectivamente. Una vez instalados, ejecutar el software RCC, encender la batería del robot y observar si aparece la imagen de la cámara en la pantalla llamada MAIN CAMERA del software Como se muestra en la Figura 1

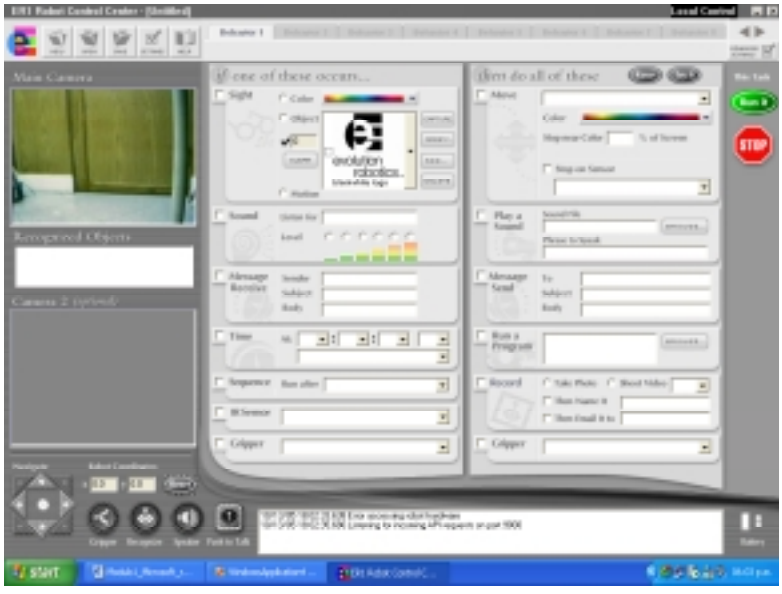

Figura 1 Pantalla del Robot Control Center.

Intente mover el robot con el botón múltiple de dicha pantalla (con cuatro puntas opuestas y perpendiculares entre sí, ubicado en el vértice inferior izquierdo). Si esto no sucede, verificar las conexiones de los motores y la carga de la batería. Si aún no funciona, cerrar el programa y volver a ejecutarlo (Consulte el manual del ER1).

Luego de comprobar que el robot funciona correctamente moviendo el mismo manualmente (con el mouse desde las flechas), se está en condiciones de configurar el software para su control por medio de la **API**.

API:
Las Interfaces de Programación de Aplicaciones (API) proporcionan acceso a los protocolos de transporte TCP/IP, Los sockets de Windows (WinSock) son una API de red diseñada para facilitar la comunicación entre aplicaciones y jerarquías de protocolos TCP/IP diferentes

Para esto hacer clic sobre el botón *Settings* (se abrirá una ventana como la que muestra la Figura 2)**,** seleccionar la pestaña *Remote Control* y hacer clic en el checkbox *Allow API Control of this instance***,** esta configuración permitirá crear un socket para comunicarse con el software RCC desde el ordenador remoto; por último presionar OK.

Se puede observar en la ventana de información que se encuentra en la parte inferior del programa la frase "*Listening for incoming API requests on port 9000"*, esto significa que el programa esta esperando una conexión.

Con esta simple configuración se está en condiciones de realizar una conexión vía Internet con el software del robot y probar algunos comandos.

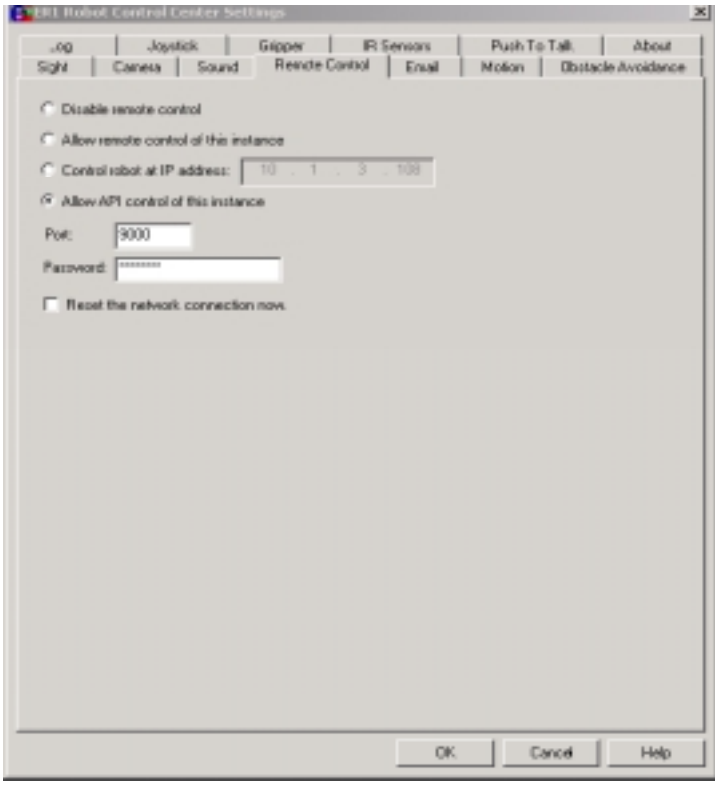

Figura 2 Ventana de configuración del software RCC

TELNET:
La aplicación y servicio de archivos e impresión o telnet sirve para acceder mediante una red a otra maquina en modo Terminal, es decir sin visualización grafica, pueden configurar hasta 65.536 puertos. Las aplicaciones y servicios TCP/IP suele utilizar los primeros 1.023
Los números de puerto del protocolo se utilizan para hacer referencia a la localización de una aplicación en particular en cada máquina (en el nivel de aplicación). Al igual que una dirección IP identifica la dirección de un host de la red, el número de puerto identifica la aplicación al nivel de transporte, por lo que proporciona una conexión completa de una aplicación de una máquina a una aplicación de otra máquina.

A continuación se probará la configuración y los comandos mediante el programa **Telnet** que se encuentra disponible en cualquier Windows. Para esto dirigirse a Inicio, clic en Ejecutar, y escribir:

telnet <ip> <puerto>↵

Ej.:
telnet 192.168.1.10 9000↵

Donde la dirección de IP debe ser la correspondiente a la computadora que este sobre el robot, y el puerto se establece desde el software RCC como se vio anteriormente, el cual por defecto es 9000.
Para verificar que se encuentra conectado presionar la tecla Enter y deberá observar la palabra *ok*, además debe observar en la pantalla de la PC portátil ubicada en el robot una ventana similar a la que muestra la Figura 3, indicando que el ER 1 se encuentra en modo remoto, bajo control de las API, si esto sucede se encuentra entonces en condiciones de enviar comandos al robot.

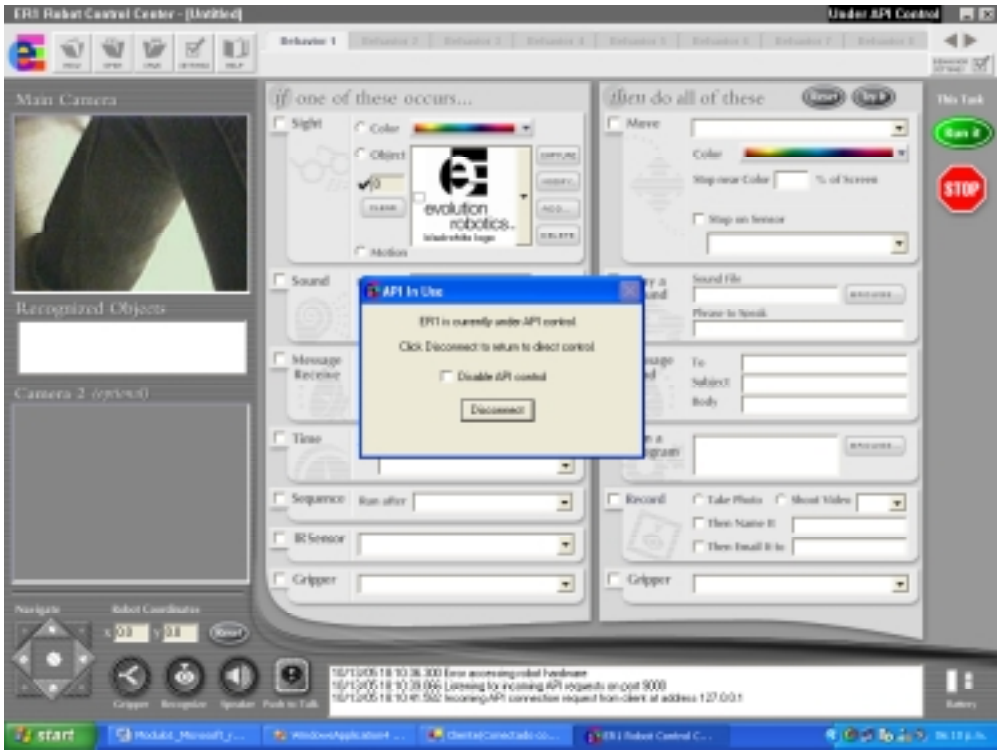

Figura 3 Conexión del ER1 Vía Telnet

COMANDOS: Estos son sólo algunos comandos que se le pueden enviar al robot, pero suficientes para los fines de este módulo, para saber la lista completa de comandos ver "ER1 *User Guide*"

Como ejemplo se probará enviar un comando para que diga una frase, escribir: *play phrase "hello"*, si todo funciona bien el robot debiera decir "hello".

En la Tabla 1 se observan una lista de los **comandos** básicos que pueden enviarse al robot. Probar reiteradamente con los mismos hasta familiarizarse con su significado y sintáxis.

COMANDOS	**PARAMETROS**	**DESCRIPCION**	**EJEMPLOS**
Move <*distancia*> <unidad>	*Distancia*: es un número decimal negativo o positivo. *Unidad*: puede ser pulgadas (i), metros (m), centímetros (c) o grados (d).	Mueve el robot una distancia, en una dirección y unidad determinada.	* **move** 30 c * **move** -25 i * **move** 45 d * **move** -30 d Ver aclaración al final de la tabla.
play phrase <"*frase a decir*">	*Frase a decir:* es lo que queremos que diga el robot.	Permite al robot reproducir sonidos o frases que le enviemos.	* **play phrase** "hello" * **play phrase** "adios" * **play phrase** "let's go"
set linear velocity <*velocidad*>	*Velocidad:* valor de la velocidad lineal en cm/seg.	Configura la velocidad lineal del robot. Máx. valor: 50	* **set linear velocity** 40 * **set linear velocity** 50
set angular velocity <*velocidad*>	*Velocidad:* valor de la velocidad angular en grados/seg.	Configura la velocidad angular del robot. Máx. valor: 90	* **set angular velocity** 75 * **set angular velocity** 15
Events	--------	Permite al robot enviarnos información de los sensores o cuando termina de realizar una acción nos devuelve un *ok.*	* **events**

Tabla 1 Comandos Básicos

Aclaración: los comandos deben respetar los siguientes espacios:
move(e)distancia(e)unidad
Donde (e) es un espacio simple.

En la Tabla 2 se dan algunos ejemplos de comandos

Comando	**Distancia**	**Unidad**	**Descripción**
move	30	c	Avanza el robot 30 centímetros hacia el frente.
move	-25	i	El robot retrocede 25 pulgadas
move	45	d	El robot rota 45 grados hacia la derecha del mismo.
move	-30	d	El robot rota 30 grados hacia la izquierda.

Tabla 2: Ejemplo de Comandos

4.2 Sumario

Durante el desarrollo de esta primera práctica introductoria se ha realizado la configuración del software RCC y luego, mediante el empleo de comandos telnet se han enviado comandos específicos al robot ER 1 observando sus resultados.
De esta manera se ha cumplido con el objetivo inicial que permitirá realizar satisfactoriamente las restantes prácticas.

5 Práctica N° 2: Comunicación con el Software RCC

Autores: Carlos Fernando Carmona, Damián De Biase, Cesar Foti

5.1 Introducción Teórica

En principio se dará una introducción teórica básica sobre algunos aspectos de la Programación Orientada a Objetos (**OOP**) necesarios para una mejor comprensión de las prácticas. [Brad J. Cox. Object Oriented Programming: an Evolutionary Approach . Addison-Wesley, 1986]

Todo **Visual Studio.NET** esta basado en clases u objetos, y cumple los requisitos que debe cumplir cualquier lenguaje orientado a objetos, soporta:

- Herencia
- Encapsulación
- Polimorfismo

OOP:
Object Oriented Programming.
Es un paradigma de programación concebido alrededor de objetos, su clasificación y la capacidad de los mismos de intercambiar mensajes y procesar datos. Soporta encapsulación, herencia y polimorfismo. De esta manera una instancia de un programa de computadora es una colección de objetos que interactúan entre sí.
Su empleo supone un grado de abstracción mayor que en la visión clásica, de manera de permitir un mejor tratamiento del problema a resolver , al igual que una mejor y mayor reutilización de los programas.

Se contrapone con la visión tradicional donde se hace énfasis en las acciones, funciones o procedimientos y en donde un programa es una lista de instrucciones ejecutadas secuencialmente.

FRAMEWORK:
Es la estructura o marco para el desarrollo de aplicaciones con tecnología .NET. Incluye Visual Studio.NET, Common Language Runtime (CLR) y Base Class Libraries (BCL).

Clases: El **framework** de **Visual Studio.NET** esta basado sobre clases u objetos. Las clases poseen código como cualquier programa, pero con algunas particularidades. En una clase se pueden definir dos cosas diferentes: los datos que la clase maneja o contiene y, la forma de acceder a esos datos. Por ejemplo se puede tener una clase llamada “Hombre”, la cual contendrá los datos y características comunes a los hombres y la forma de acceder o modificar esos datos. Los datos están representados por todos los campos y **propiedades** que contiene la clase y la forma de acceder a estos será a través de los métodos que posea la clase. Es decir que una clase queda definida por sus campos de propiedades y sus métodos.

Objetos: Para poder usar las propiedades y métodos que se definen en una clase se deberá crear un objeto de esa clase, a partir del cual se podrá acceder a los métodos y propiedades públicas que la clase contenga. Es decir, un objeto es una instancia en memoria de una clase determinada, y además se podrán tener varias instancias en memoria de una misma clase.

Ejemplo: Se crean dos objetos de la clase “Hombre” llamados “Juan” y “Pedro”.

Herencia: Existen dos tipos de clases, las clases “bases” y las clases “derivadas”. Según el **MSDN** de Microsoft: “herencia es la capacidad de definir clases que sirvan como base para las clases derivadas. Las clases derivadas heredan, y pueden extender las propiedades, métodos y eventos de la clase base”.

PROPIEDADES: Son las características o datos que contiene una clase.

HERENCIA: Es la capacidad que tienen las clases derivadas de emplear métodos y propiedades que fueron definidos en las clases base.

Es decir, la herencia brinda la posibilidad de crear clases que estén basadas en otras clases ya existentes, de esta forma la nueva clase heredará todas las propiedades y métodos de la clase base, y además se podrán agregar otros procedimientos nuevos.

Ejemplo: La clase llamada "Hombre" está derivada de la clase "Humanos", derivada a su vez de "Animales" y esta a su vez de "Seres Vivos". La clase "Hombre" hereda todas las características comunes de la clase "Humanos", además de añadirle algunas características particulares que hacen a los hombres. De la clase "Humanos" podría también derivarse una clase llamada "Mujer", que herede las características comunes a estos, pero le agrega, como en el caso de "Hombre", características particulares de las mujeres. De esta forma, ambos han heredado de la clase Humanos pero los dos son diferentes.

ENCAPSULA-CIÓN : Es la capacidad de separar lo que la clase hace (métodos, eventos, etc.), del código que lo hace posible.

Encapsulación: Según el **MSDN** de Microsoft: "La encapsulación es la capacidad de contener y controlar el acceso a un grupo de elementos asociados. Las clases proporcionan una de las formas más comunes de encapsular elementos".
Para que sea posible la interacción entre objetos estos exhiben interfaces materializadas mediante métodos propios de la clase a la que pertenecen. Esto permite que otros objetos conozcan o alteren su estado pero sólo de manera controlada a través de métodos cuyo código es propio de su clase.
De esta forma quedan separadas las respuestas a eventos o llamados a métodos que ofrece un objeto de una determinada clase, de la forma (código) que hace posible esa respuesta. Es decir: para la interacción con el objeto sólo hay que conocer la interfaz y cómo usarla.

Ejemplo: Si mediante la invocación del método "Moverse", se le pide al objeto "Juan", una instancia de la clase "Hombre", que se mueva, este ejecutará la acción solicitada haciendo que "Juan" se mueva de la manera que lo hacen los seres humanos (caminando), siendo transparente para el solicitante cual es esa forma y la manera en que se encuentra codificada.

Polimorfismo: Según el **MSDN** de Microsoft: "El polimorfismo se refiere a la posibilidad de definir múltiples clases con funcionalidad diferente, pero con métodos o propiedades denominados de forma idéntica, que pueden utilizarse de manera intercambiable mediante código cliente en tiempo de ejecución.".
Con el polimorfismo se cuenta con la posibilidad de usar métodos de distintas clases de forma genérica sin preocuparse por la clase a la que pertenecen.

Ejemplo: Se tiene un conjunto de "Seres Vivos", "Juan" una instancia de la clase "Hombre" y "Pío" uno de la clase "Pájaro". Si a cada objeto del conjunto se le pide que se mueva mediante la invocación del método

"Moverse", ambos responderán moviéndose, pero lo harán de maneras diferentes: el hombre caminando y el pájaro volando.

5.2 Desarrollo de la práctica.

MÉTODO:
es un procedimiento que se utiliza para realizar una tarea especifica en una clase o módulo. Simplemente ejecuta el código que hay dentro de él.

El objetivo de la clase "`Cliente`" es facilitar ciertos **métodos** y **eventos** para realizar la comunicación con el programa RCC de una forma sencilla. Para lograr esto la misma posee métodos públicos, como por ejemplo "`Conectar`", "`EnviarDatos`", y eventos tales como "`DatosRecibidos`" y "`ConexionTerminada`". Es decir, incluyendo esta clase en un proyecto de **Visual Studio.NET** podrá realizar un programa de comunicación con el software RCC de una forma sencilla.
La clase esta dividida en secciones o regiones para facilitar su comprensión y obtener así, una estructura ordenada y con secciones bien diferenciadas.

EVENTO:
Es un procedimiento que generalmente se ejecuta cuando el sistema Windows lo provoca. Un ejemplo es hacer clic en una ventana, hacer clic en un botón o escribir en una caja de texto.

A continuación se describe en forma general como utilizar los métodos y eventos de la clase "`Cliente`".

Para la utilización de la clase se debe crear un nuevo proyecto en **Visual Studio.NET,** y agregar la clase al proyecto. Una vez que se tiene la clase incluida en el proyecto se debe crear una instancia en memoria, con el fin de tener un objeto de dicha clase que permita la utilización de los métodos y eventos de la misma. Para esto se debe agregar el siguiente código:

```
Dim WithEvents Cliente As New Cliente
```

La sentencia `WithEvents` permite instanciar la clase con todos sus eventos públicos, sin esta sentencia no se podrían utilizar los eventos de la clase.
Una vez creado el objeto se puede utilizar los métodos que contiene la clase. Primero se debe indicar a que dirección de IP y número de puerto se desea conectar, para lo cual se utilizan dos propiedades que posee la clase llamadas `IPDelHost` y `PuertoDelHost` respectivamente. Después de asignar los valores correspondientes a las propiedades se puede conectar utilizando para esto el método `Conectar` de la clase en cuestión. El siguiente bloque de código hace referencia a la asignación de las propiedades y al método `Conectar`:

```
Cliente.IPDelHost = txtIP.Text              'La propiedad
                                            IPDelHost de la
                                            clase Cliente
                                            toma el valor que
                                            está en el
                                            textbox txtIP.

Cliente.PuertoDelHost = txtPuerto.Text      'La propiedad
                                            PuertoDelHost de
                                            la clase Cliente
                                            toma el valor que
                                            está en el
                                            textbox txtPuerto


Cliente.Conectar()                          'Llamo al método
                                            Conectar de la
                                            clase Cliente
                                            para establecer
                                            la conexión.
```

Si la conexión se realiza satisfactoriamente la clase dispara un evento llamado `Conexión` con los parámetros IP y puerto los cuales refieren a la dirección de IP y Puerto del host al cual se conectó. Este evento se puede utilizar para informar que se ha establecido la conexión. El código es el siguiente:

```
'Evento de la clase Cliente que se dispara cuando el
programa se conecta al programa del robot.

Private Sub Cliente_Conexion(ByVal IP As String, ByVal
puerto As Integer) Handles Cliente.Conexion

Me.Text = Me.Text & "(Conectado con el servidor" & IP & "en
el puerto" & puerto & ")" 'Muestra en el título del
formulario que está conectado.

End Sub
```

THREAD:
nombre de una clase de Visual Studio.Net que contiene métodos para crear uno o varios subprocesos dentro de un proceso.

Cuando se realiza la conexión, la clase `Cliente` además de disparar el evento `Conexión` crea e inicia un nuevo hilo (**thread**) el cual hace referencia a una función privada llamada `LeerSocket`. Lo que hace esta función es leer constantemente el **stream** de la conexión con el fin de disparar un evento en el momento que se recibe algún dato. Como la función esta declarada como privada no se tiene acceso a la misma.
Al recibir un dato desde el servidor (en nuestro caso la función de servidor la cumple el software RCC) la función `LeerSocket` dispara un evento llamado `DatosRecibidos` el cual tiene como parámetros los datos que se reciben desde el servidor. Esto significa que en el código del programa se debe incluir dicho evento de la siguiente forma:

```
Private Sub Cliente_DatosRecibidos(ByVal datos As String)
Handles Cliente.DatosRecibidos

'El código que escriba aquí se ejecutará cada vez que el
software RCC envíe datos.

End Sub
```

Por ultimo veremos como enviar datos al servidor (software RCC), para lo cual se dispone de un método perteneciente a la clase "`Cliente`" llamado `EnviarDatos`. El mismo tiene como argumento los datos a enviar, que en este caso serán los comandos del robot. Para enviar los comandos basta con escribir la siguiente línea de código:[2]

```
Cliente.EnviarDatos("aquí se escribe el comando a enviar en
formato string")
```

Además del comando se debe enviar el retorno de carro y el salto de línea, para esto el lenguaje Visual Basic dispone de la constante **vbCrLf**, con lo cual el formato general sería:

```
Cliente.EnviarDatos("comando" + vbCrLf)
```

Ejemplo:

En la siguiente línea se envía el comando **move 20 c** (avanzar 20 centímetros hacia el frente):

vbCrLf: constante proporcionada por Visual Studio.Net que representa el retorno de carro y salto de línea.

```
Cliente.EnviarDatos("move 20 c" + vbCrLf)
```

Para la implementación de la clase se adjunta un programa ejemplo llamado "Cliente", el cual permite enviar comandos simples como también ejecutar una lista de comandos secuencialmente.

La Figura 4 muestra la pantalla del programa ejemplo para la utilización de la clase `Cliente`.

[2] Si se hubiese declarado la clase sin la sentencia WITHEVENTS no se podría tener acceso a los eventos de dicha clase.

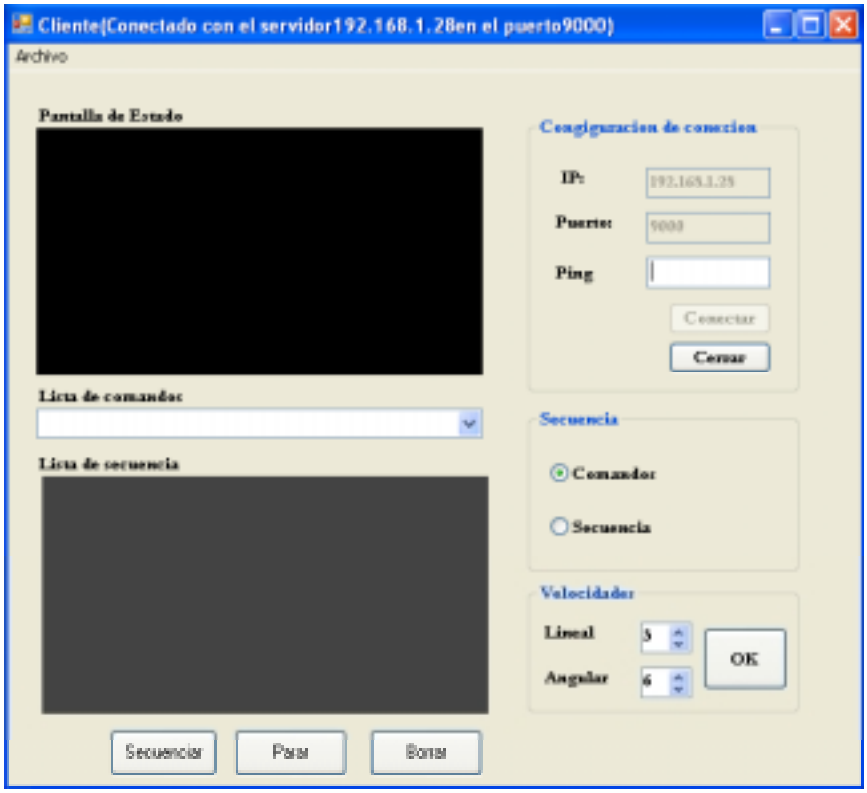

Figura 4: Clase Cliente

5.3 Sumario

En esta práctica se realizó una breve descripción de la POO (Programación Orientada a Objetos) con el objetivo de dar al lector ciertos conceptos básicos de este tipo de programación y se cumplió con el objetivo de la misma realizando una explicación de la clase (Clase `Cliente`) que permitirá al lector realizar la comunicación por un socket con el RCC y a su vez desarrollar las prácticas siguientes.

6 Práctica N° 3: Interfaz con el Usuario

Autores: **César Foti, Carlos Fernando Carmona, Damián De Biase**

6.1 Introducción

En esta práctica se desarrolla la interfaz visual del programa. Se emplean los controles para formularios que **Visual Studio**.**NET** provee para tal fin.

El objetivo[2] de este módulo es aplicar un algoritmo de búsqueda heurística para que el robot pueda encontrar el camino más corto para recorrer una serie de puntos definidos en un mapa del entorno. El algoritmo de búsqueda se describe en la práctica 4.

La interfaz visual con el usuario que se propone en esta práctica resuelve de una manera práctica el ingreso del punto inicial, los puntos intermedios a recorrer y finalmente dibuja la trayectoria más corta obtenida con el algoritmo del viajante de comercio (Traveling Salesman Problem - TSP).

Además, a partir de la trayectoria obtenida, se codifican los comandos necesarios para poder mover el robot como ya se explicó en la practica 1.

6.2 Descripción del Programa

PictureBox:
Es un control en el que se pueden mostrar gráficos en diferentes formatos (bitmaps, jpeg, gif, etc). Posee métodos para dibujar las figuras elementales (líneas, círculos, etc)

ListView:
Es un control que permite mostrar y almacenar colecciones de objetos. Se pueden seleccionar diferentes formas de exhibirlos: Íconosgrandes o pequeños, lista o reporte.

El programa cuenta con un **PictureBox** que sirve como área de trabajo, donde se representan las coordenadas inicial[3] y las intermedias. El ingreso de los puntos se hace mediante el click del mouse sobre el área de trabajo. Ver Figura 5. En respuesta a este evento (click del mouse), se dibuja un círculo sobre la coordenada seleccionada y sus valores se cargan secuencialmente en una lista implementada con un control **ListView**.

[3] Se considera que el robot debe volver siempre al punto de partida; el punto inicial coincide con el final.

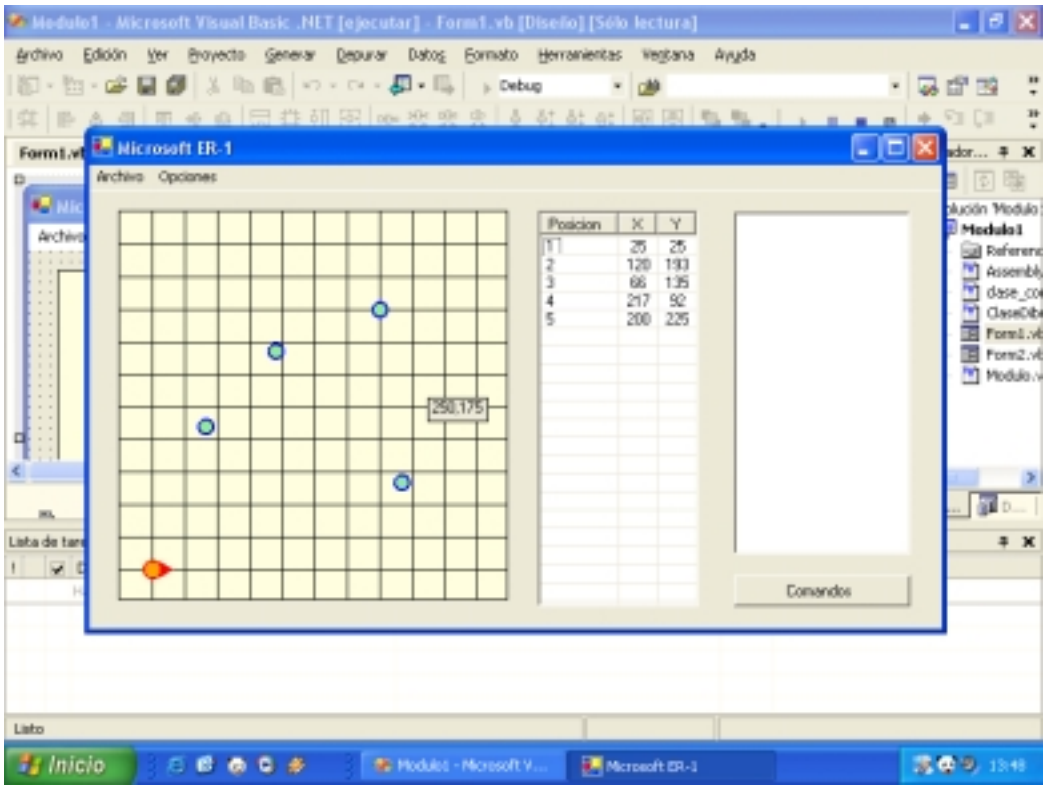

Figura 5 área de trabajo

El programa se completa mediante un menú en tiempo de diseño que se divide a su vez en otros dos llamados Archivo y Opciones.
El menú archivo tiene los siguientes ítems:

- *Nuevo*, reinicia los parámetros para comenzar una nueva búsqueda.
- *Abrir*, permite seleccionar y abrir un archivo que contiene una colección de puntos, los imprime en el ListView y dibuja la trayectoria dentro del área de trabajo.
- *Guardar*, permite almacenar la colección de puntos ingresada por el usuario en un archivo binario.
- *Cerrar*, termina el programa.

Dentro del submenú de *Opciones* se encuentra fundamentalmente el ítem *Área de Trabajo*, que mediante la presentación de un segundo formulario permite cargar el valor del punto inicial, los valores del área de trabajo y también los pasos de la grilla en base y en altura.

El programa se completa con un menú contextual, en donde podemos seleccionar distintas propiedades como:

- *Dibujar círculo*, permite ingresar una nueva “ciudad” a recorrer representada por un círculo y agrega el valor de la coordenada en el ListView.

- *Dibujar Trayectoria*, presenta en el mapa la trayectoria, indicando con flechas el sentido de circulación. Ver Figura 6
- *Borrar*, borra el último punto indicado.
- *Borrar Todo*, limpia toda el área de trabajo y las coordenadas del ListView.
- *Pegarse a Grilla*, toma una coordenada del nodo de la grilla mientras que el puntero del Mouse esté en las cercanías del nodo de la grilla.

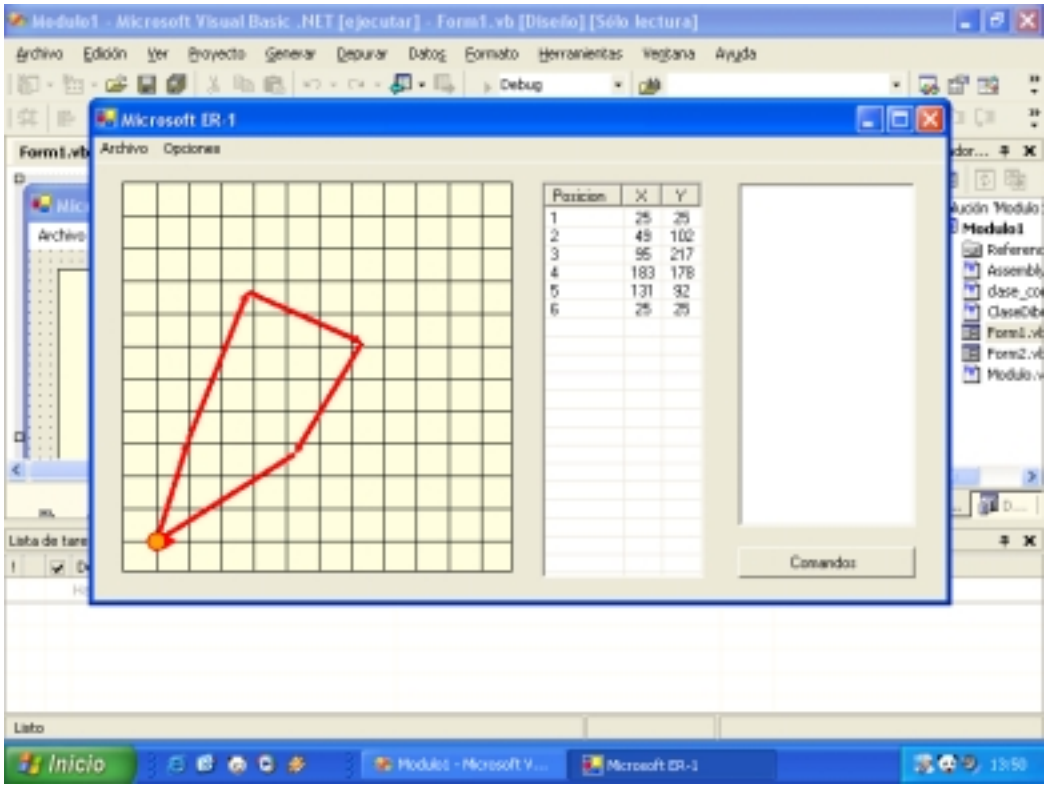

Figura 6 Trayectoria

6.3 Conformación del Programa

El programa se encuentra conformado por dos formularios y tres clases. El formulario principal, el secundario, una clase que contiene los métodos necesarios para dibujar, una clase que manipula el ListView y la clase que a partir de la trayectoria codifica los comandos del robot.

Clase Dibujar.

La clase `Dibujar` contiene cuatro procedimientos públicos y dos funciones. Tanto las funciones como los procedimientos utilizan parámetros en común como la base y altura del área de trabajo y la base y

la altura del objeto donde se dibujará. Estas variables se declaran como propiedad dentro de esta clase, y se inicializan al instanciar un objeto de la misma.

Una de las opciones que presenta el programa es la de cambiar la altura y el área de trabajo. Si el PictureBox tiene una altura y una base fija se tendrá que construir una función para llevar el valor de la coordenada X a la escala conveniente. Lo mismo ocurre con la coordenada Y. Estas funciones se llaman `EscalaX` y `EscalaY`. Se le pasan como parámetros de entrada los valores de la coordenadas y la función devuelve la conversión a la escala de las mismas.

El procedimiento `Grilla` es una rutina que tiene como fin dibujar una grilla en el objeto que elijamos como área de trabajo. Para esto es necesario pasarle los pasos de la grilla en altura y base y por supuesto el objeto donde queremos que se dibuje[4].

Tanto la base y altura del área de trabajo, como la base y la altura del objeto se declaran como propiedades de la clase `Dibujar`. Esto es así debido a que son variables de uso común por parte de diferentes funciones y procedimientos de los objetos de esta clase.

El procedimiento `Línea` dibuja una línea recta con punta de flecha desde un punto inicial hasta uno final, ambos pasados como parámetros, sobre el objeto seleccionado. Ver Figura 6.

De manera análoga la rutina `Circulo` dibuja un círculo considerando la coordenadas X y Y donde se quiere dibujar el circulo, el radio, el color de relleno y el color de la circunferencia. Al igual que las anteriores también reciben el objeto sobre el que dibuja, el área de trabajo.

El procedimiento `Dibujar_puntoinicial` dibuja un ovoide terminado en flecha. Recibe el objeto sobre el que se quiere dibujar, las coordenadas X, Y y el ángulo del objeto respecto del sistema de coordenadas.

Clase Lista.

El ListView es un control muy versátil. Permite tener cuatro conformaciones diferentes:

- En columnas (conformación usada, en este ejemplo).
- En una lista.
- Con iconos grandes.

[4] Para poder realizar gráficos vectoriales es necesario importar las clases “System.Drawing” y “System.Drawing.Drawing2D”.

- Con iconos pequeños.

Como se trabaja con un listado detallado, se debe definir la columnas que lo conforman. En este caso el ListView se separó en tres columnas:
La primera la columna "*Posición*" donde se imprime la posición de ingreso del punto, la columna "*X*" y la columna "*Y*" correspondiente a los valores de la coordenada del punto.

En un listado por columnas la primera columna se asocia con los elementos del listado y el resto de las columnas se refiere a los subitems de cada elemento. En nuestro caso la columna *Posición* es el item del elemento y las columnas *X* y *Y* son los subitems.

Cada columna se crea a partir de la propiedad `columns` en la ventana de propiedades.

La clase `Lista` contiene dos procedimientos necesarios para su manejo.

El procedimiento `Llenar_lista` agrega al ListView las coordenadas del punto ingresado. Recibe como parámetros de entrada el nombre del ListView y las coordenadas *X* e *Y* del punto considerado. Este procedimiento completa automáticamente por orden de ingreso el valor de la columna *Posición*. El segundo procedimiento es `Borrar_lista`, que al ser invocado borra todo el contenido del objeto.

Clase Comandos.

La clase `Comandos` esta compuesto por un único procedimiento llamado `comandos`. Recibe como parámetro un vector que contiene las coordenadas de los puntos que se quiere que el robot recorra. Luego por trigonometría[5] se sacan las distancias y los ángulos relativos. Finalmente devuelve un vector con la lista de comandos necesarios para que el robot cumpla con su recorrido en la secuencia previamente establecida. El formato de estos comandos es el que utiliza el robot cuando se accede por vía *TelNet*.

Formulario:
Es una zona de pantalla, usualmente rectangular que se utiliza para presentar información al usuario y también para aceptarla de él.

Los formularios.

Como se explico anteriormente las características del área de trabajo se cargan a partir de otro **formulario**. Dentro de este nuevo formulario se agregan los controles que son necesarios para nuestro fin.

Lo que se busca es que cuando se produzca el evento `click` sobre el ítem "*Área de trabajo*" se abra el segundo formulario y se le puedan

[5] Para poder realizar las herramientas matemáticas es necesario importar la clase "System.Math".

asignar valores a las siguientes variables en respuesta al evento `click` del botón *Aceptar*.

```
If formulario2.ShowDialog = DialogResult.OK Then

      Altura = formulario2.datos.Altura
      Base = formulario2.datos.Base
      Altura_g = formulario2.datos.Altura_g
      Base_g = formulario2.datos.Base_g
      Angulo = formulario2.datos.Angulo
      Xinicial = formulario2.datos.Xinicial
      Yinicial = formulario2.datos.Yinicial

End If
```

Para esto se utiliza una estructura de condición `If…Then`, y dentro de la condición se establece que si el dialogo de los dos formularios resulta bien, las variables del formulario principal tomaran los valores de la propiedad datos del formulario secundario.
Para que esto se efectúe o que el dialogo resulte “Bien”, dentro del evento del botón `aceptar` del formulario secundario tenemos que agregar:

```
Me.DialogResult = DialogResult.OK
```

O sea que el dialogo entre el Formulario principal y el secundario resultó “OK”. Una vez que sucede esto los valores ingresados en los TextBox del Formulario secundario se igualaran a las variables del Formulario Principal.

En la propiedad *AcceptButton* del Formulario Secundario tiene que estar seleccionada la opción “*Name*” del botón que usamos para confirmar el dialogo entre formularios. Esta propiedad se encuentra en la ventana de propiedades.

Pero todavía hay un problema. ¿Qué pasa si el usuario ingresa un valor negativo o si bien no ingresa un número?. Evidentemente hay que tratar este “error”. Para esto se utiliza el evento `Validating`.

MsgBox:
Es una función que muestra un mensaje al usuario y le ofrece la posibilidad, mediante botones de seleccionar una alternativa. Espera que el usuario presione uno y devuelve información sobre la alternativa seleccionada.

Este evento se dispara cada vez que el cursor de texto abandona el TextBox. En respuesta a dicho evento, se intenta primero convertir la cadena de caracteres en un número entero de 16 bits, si esto no es posible debido a que lo ingresado no es un número se toma la excepción, se cancela el procedimiento y se imprime un **MsgBox** ("Error: La Altura tiene que ser un número"). De ser posible la conversión, se verifica entonces que no sea un número negativo; En caso de serlo se cancela el método y se imprime un MsgBox ("Error: La Altura tiene que ser un número positivo").

Evento Paint

Un problema que se presenta luego de dibujar los puntos y líneas sobre el área de trabajo es que lo dibujado desaparece si sobre dicha área se superpuso momentaneamente otro formulario, la ventana de otra aplicación o si se modificó el tamaño del mismo.
Para evitar esto es necesario responder al evento `Paint`. Este evento se dispara cada vez que cambiamos las dimensiones del formulario y cuando desaparece otro objeto que estaba por encima del dibujo.

La rutina invocada como respuesta al disparo de este evento debe ser capaz de redibujar todo el contenido del área de trabajo de nuevo. Por lo tanto dentro de este procedimiento se debe dibujar la grilla, luego el círculo o la trayectoria según corresponda y por ultimo la rutina del punto inicial.

El evento `Paint` también se dispara por código mediante el método refresh. Ej:

```
PictureBox1.Refresh()
```

6.4 Sumario

A lo largo de esta práctica se mostraron las clases que contiene el programa. Así como también se hicieron aclaraciones sobre algunos de sus objetos y la declaración de las variables como propiedades, de tal manera de respetar el paradigma de la programación orientada a objetos.
Se mostró como trabajar con dos formularios y la manera de dar respuesta al evento `Paint`.

7 Práctica N° 4: Implementación de una Técnica de Búsqueda

Autores: **Daniel López Amado, Sebastián Verrastro**

7.1 Introducción

El Problema del Viajante de Comercio (Traveling Salesman Problem - TSP), trata de un agente de comercio que debe visitar "n" puntos para vender u ofertar sus productos. El costo de ir de un punto a otro se define mediante cij y el camino es reversible (cij = cji). El problema supone todas los puntos conectadas con todos, el agente deberá visitar todos los puntos comenzando por uno cualquiera, pasar por todos los demás, solo una vez y volver al primero, de manera que se minimice el costo del recorrido total.

Problemas del tipo NP- completo:
Es un problema donde el número de posibles soluciones crece en forma exponencial a medida que aumenta el número de variables independientes.

Este problema está dentro de la categoría de los **problemas** de complejidad **NP-completo**. Esto es así, porque el número de posibles soluciones crece en forma exponencial según el número de nodos del grafo (puntos a recorrer), y rápidamente sobrepasa las capacidades de cálculo de los ordenadores más potentes. Los intentos por resolver este problema han sido numerosos y variados. Sin embargo, hasta hoy, se considera un problema no resuelto.

Para "n" puntos a recorrer existen factorial de "n" (n!) caminos posibles (si los caminos son reversibles es entonces n!/2). Una forma para encontrar el camino óptimo es calcular el costo total de cada camino posible y luego elegir el recorrido que arroje el menor valor calculado. Esto tiene como inconveniente que para unos pocos puntos, dependiendo del computador, el tiempo que se tarda en encontrar el camino óptimo es inaceptable.

Algoritmo A*:
Es un algoritmo que garantiza encontrar la solución óptima utilizando una función heurística de evaluación.

Existen diversos algoritmos que utilizan distintas técnicas para abordar este problema. El siguiente trabajo implementa el algoritmo A* (léase A estrella) [N.Nilsson].

El algoritmo A* utiliza una función heurística de evaluación "fn" para seleccionar los puntos del camino a recorrer. La función de evaluación es la suma de dos términos como se muestra a continuación:

$f(n) = g(n) + h(n)$

donde:

g(n) es el costo desde el punto de inicio hasta el punto "n".
h(n) es el costo **estimado** desde el punto "n" hasta llegar a la meta.
f(n) es el costo total estimado del camino pasando por el punto "n".

Heurística admisible: Una heurística es admisible cuando la estimación del valor de la solución es siempre menor al valor real de la misma.

La elección de la **estimación** de costos h(n) no debe sobreestimar el costo real (**heurística admisible**), es decir, sí el mínimo costo desde un punto "n" hasta la meta es h*(n) entonces se debe elegir una h(n) tal que $h(n) \leq h^*(n)$.
La elección de h(n) juega un papel importante en la velocidad de convergencia del algoritmo. h(n) debe ser lo más grande posible cumpliendo la restricción de no sobreestimar el mínimo costo real, a pesar de que el mismo no es conocido.

Para el algoritmo se implementó el siguiente mecanismo para la elección de h(n): Primero se comparó el costo entre el primer punto y los demás y se eligió el más bajo, luego se comparó el costo entre el segundo punto y los restantes y se eligió el más bajo, y así para todos los demás puntos. Este mecanismo garantiza que el costo estimado será menor o igual al menor camino real.
Esta evaluación es dinámica y se recalcula a medida que se van explorando los caminos.

7.2 Aplicación del Algoritmo A* al TSP

Este algoritmo consta de dos etapas Inicio y Loop.

Etapa: INICIO

0) Camino (1) = Punto Inicial
1) Se calcula la función de evaluación en el Camino(1). Para este caso g(1) es 0, entonces f(1) = h(1) .
2) Se expande el punto INICIAL. Expandir un punto significa generar tantos **caminos parciales** como puntos a los cuales se puede llegar desde el punto expandido, excluyendo los puntos que se hayan visitado anteriormente en este camino. Los caminos así generados, tendrán un punto terminal que será candidato a ser expandido.

Caminos parciales: Es una secuencia de puntos que aun no esta completa.

Ver ejemplo en la Figura 7

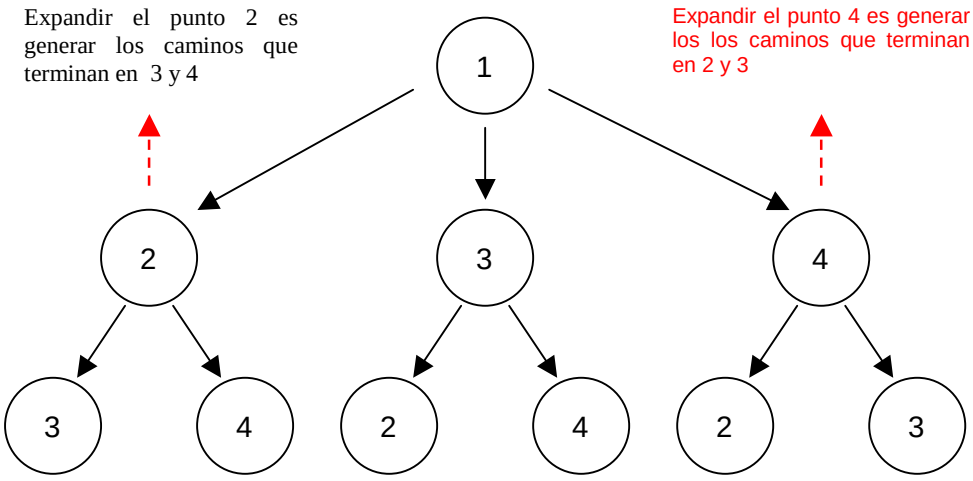

Figura 7 Ejemplo de expansión

3) Se calcula la función de evaluación para cada uno de los caminos parciales generados en 2.
4) Se reordena la lista de caminos parciales según el "fn" de cada camino. La lista se ordena en forma creciente.

Etapa: LOOP

5) Se expande el último punto del primer camino parcial de la lista, es decir, el camino con menor "fn".
6) Se calcula la función de evaluación para cada uno de de los caminos parciales generados en 5.
7) Se reordena la lista de caminos parciales según el "fn" de cada camino. La lista se ordena en forma creciente.
8) Se verifica si el primer camino de la lista (el de menor "fn") es un camino completo. Sí no es un camino completo se vuelve a 5. Sí es un camino completo entonces es el camino más corto, y la función de evaluación "fn" es el costo real.

7.3 Implementación del algoritmo A*

Se define la estructura de datos "`DatosDelCamino`". En esta estructura se podrá almacenar toda la información referida a un camino en particular.

Como se desprende de la explicación del algoritmo es necesario guardar en memoria la lista de caminos parciales que se están explorando, para esto se declaró un vector (`Camino`) del tipo de datos `DatosDelCamino`. Este vector debe permanecer ordenado según el campo "`fn`". Para que el algoritmo de ordenamiento sea rápido, no se intercambian los datos de una posición de memoria a otra, en vez de esto se utiliza el campo "`next`" como índice para apuntar al siguiente elemento ordenado.

```
DatosDelCamino
{
     uint[] Punto;
     uint fn;
     bool CaminoCompleto;
     uint next;
}
```

Campos de la estructura "`DatosDelCamino`"

`Punto`: Es un vector que se utiliza para almacenar la secuencia de puntos de un camino. Cada uno de los elementos de este vector contiene el número de un punto visitado. Se dimensiona dinámicamente según sean los puntos a recorrer.

`fn`: Se utiliza para almacenar el valor de la función de evaluación del camino.

`CaminoCompleto`: Es un flag que se utiliza para señalizar cuándo se completo el camino. Sí luego de ordenar la lista en función de "`fn`" como se explicó anteriormente, el primer camino de la lista tiene el flag `CaminoCompleto` en 1 (true) entonces, ese es el camino más corto.

`next`: Se utiliza como índice para apuntar al siguiente elemento ordenado de la lista de caminos.

Constructores

```
Viajante (uint[,] costoEntrePuntos)
```

Descripción:

Calcula la predicción de costos "hn". Inicializa el primer camino y todas las variables necesarias para realizar la búsqueda heurística

Parámetros:

`costoEntrePuntos`: Matriz cuadrada que debe contener los costos entre todos los puntos.

```
Viajante (uint[] CoordenadaX, uint[] CoordenadaY)
```

Descripción:

Calcula el costo entre todos los puntos. Calcula la predicción de costos "hn". Inicializa el primer camino y todas las variables necesarias para realizar la búsqueda heurística. (Esta función se utiliza cuando los caminos son euclideos y se los ingresa por sus coordenada x,y en el plano)

Parámetros:

`CoordenadaX`: Vector con las coordenadas "X" de todos los puntos.

`CoordenadaY`: Vector con las coordenadas "Y" de todos los puntos

Métodos públicos[6]

`RecorrerCamino`

Descripción:

Recorre parte del árbol de búsqueda, calculando en cada punto la función de evaluación: fn = gn + hn, donde "gn" es el costo entre el punto de inicio y el punto "n" y "hn" es el costo estimado desde el punto "n" hasta llegar a la meta

Retorno:

Vector con la secuencia de puntos del camino más corto. Los puntos se numeran a partir de 1 y no se agrega el punto de inicio (que siempre es el punto 1)

Si no encuentra el camino por falta de memoria retorna todo en cero.

`TiempoDeBusqueda`

Descripción:

Calcula el tiempo en segundos que el algoritmo tardó en encontrar el camino más corto

Retorno:

Cantidad de segundos

[6] ***Todos los métodos están comentados en el código.***

`CostoDelCamino`

Descripción:

Calcula el costo del camino más corto.

Retorno:

Costo del camino más corto.

7.4 Estadística

Se realizaron ensayos con distintas funciones “hn” (predicción del costo mínimo desde un punto hasta la meta) para probar la eficiencia del algoritmo.

7.4.1 Primer ensayo

Se generan en forma aleatoria las coordenadas (x,y) de 9 puntos con valores entre 0 y 200.

Búsqueda no informada: hn: Se iguala a cero.

1	31	segundos
2	27	segundos
3	22	segundos
4	13	segundos
5	25	segundos
6	6	segundos
7	21	segundos
8	10	segundos
9	32	segundos
10	28	segundos
PROMEDIO	21.5	segundos

Búsqueda heurística: hn: Se calcula como está explicado anteriormente

1	271	msegundos
2	20	msegundos
3	60	msegundos
4	50	msegundos
5	40	msegundos
6	1863	msegundos
7	641	msegundos
8	20	msegundos
9	20	msegundos
10	90	msegundos
PROMEDIO	307.5	msegundos

Conclusión del ensayo

Grado de Eficiencia para 9 puntos = 21.5 seg / (21.5 seg + 0.3075 seg) * 100 = 98.58%

7.4.2 Segundo ensayo

Se generan en forma aleatoria las coordenadas (x,y) de 11 puntos con valores entre 0 y 200.

Búsqueda poco informada: hn: No tiene en cuenta la predicción del retorno al punto de inicio.

1	185	segundos
2	11	segundos
3	4	segundos
4	65	segundos
5	159	segundos
6	98	segundos
7	2693	segundos
8	33	segundos
9	23	segundos
10	10	segundos
PROMEDIO	328.1	segundos

Ensayos con 11 ciudades con predicción del último número.

1	958	msegundos
2	232	msegundos
3	1097	msegundos
4	63945	msegundos
5	67714	msegundos
6	3069	msegundos
7	523	msegundos
8	163113	msegundos
9	673050	msegundos
10	371	msegundos
PROMEDIO	97.4	segundos

Conclusión del ensayo

Se puede observar que no teniendo en cuenta la predicción de volver al punto de inicio (búsqueda menos informada), para 11 puntos, se triplica el tiempo de búsqueda.

7.5 Sumario

Durante el desarrollo de esta práctica se implementó el algoritmo A* para la resolución del problema del viajante de comercio. Se pudo observar la importancia de la buena elección de la estimación del costo, para que el algoritmo converja a la solución rápidamente. Esto se puede verificar con los resultados de los ensayos realizados.