10 Inteligencia artificial

David: Martin es el verdadero hijo de Mami y de Henry. Después de que encuentre al Hada Azul podré ir a casa. Mami va a amar a un nene de verdad. El Hada Azul me transformará en uno.
Gigolo Joe: El Hada Azul es Mecha, Orga, hombre o mujer?
David: Mujer.
Gigolo Joe: ¿Mujer? Yo conozco a las mujeres! A veces me piden por mi nombre. Conozco todo acerca de las mujeres. Casi todo cuanto haya que saber. No hay dos iguales, y después de que me conocen, no hay dos iguales. Y sé dónde se puede encontrar a la mayoría de ellas.
David: ¿Dónde?
Gigolo Joe: Rouge City. Cruzando el Delaware.

Diálogo entre dos entidades de inteligencia artificial: Gigolo Joe (interpretado por by Jude Law) y David (interpretado por Haley Joel Osment) en la película, *Inteligencia Artificial*, Dirigida por Steven Speilberg, Warner Bros., 2001.

Página opuesta: I.A. Inteligencia Artificial

Del póster de la película. Warner Bros., 2001.

La cuestión de la inteligencia

El intento de comprender la inteligencia humana probablemente es uno de los interrogantes humanos más viejos que, sin embargo, aún queda por responder en su totalidad. Con el advenimiento de las computadoras y los robots, la cuestión de si los robots y las computadoras pueden ser tan inteligentes como los humanos ha llevado a las búsquedas científicas en el campo de la *Inteligencia Artificial (IA)*. La pregunta de si una computadora puede ser inteligente fue discutida lúcidamente por el Profesor Alan Turing en 1950. Para ilustrar las cuestiones subyacentes a la inteligencia de las máquinas, Turing ideó un experimento en forma de *juego de imitación*. Se juega con tres personas, un hombre, una mujer y un interrogador. Todos están en habitaciones separadas e interactúan entre ellos tipeando un texto en la computadora (bastante parecido al modo en que la gente interactúa entre sí con dispositivos de mensajería instantánea). La tarea del interrogador es identificar cuál de las personas es el hombre (o la mujer). Para que el juego resulte interesante, cualquiera de los jugadores puede intentar ser engañoso al dar sus respuestas. Turing argumenta que una computadora debería ser considerada inteligente si se la pudiera hacer jugar el rol de cualquiera de los jugadores sin darse a conocer. Este *test* de inteligencia se ha llamado el *Turing Test* y ha generado mucha actividad en la comunidad de investigadores de IA (ver Ejercicios). El diálogo que se muestra arriba, de la película *Inteligencia Artificial*, describe un aspecto del test de inteligencia diseñado por Alan Turing, basado en el intercambio entre Gigolo Joe y David, ¿pueden concluir que ambos son inteligentes? ¿Humanos?

Luego de más de cinco décadas de investigación en IA, el campo ha madurado y evolucionado de muchas maneras. En primer lugar, el foco sobre la inteligencia no se limita ya a los humanos: los insectos y otras formas de animales con variadas formas de inteligencia han sido objeto de estudio dentro de la IA. También ha habido un intercambio enriquecedor de ideas y modelos entre científicos, biólogos, psicólogos, científicos cognitivos, neurocientíficos, lingüistas y filósofos de la IA.

Han visto ejemplos de estas influencias en los modelos de los vehículos de Braitenberg introducidos anteriormente. Dada la diversidad de investigadores involucrados en la IA, también ha habido una evolución acerca de qué se trataba la IA. Volveremos sobre esto más adelante en el capítulo. Primero, les daremos algunos ejemplos de modelos que pueden ser considerados inteligentes y que son comúnmente usados por científicos de IA.

Comprensión de Lenguaje

Un aspecto de la inteligencia reconocido por muchas personas es el uso del lenguaje. Las personas se comunican entre sí usando un lenguaje. Hay muchos (varios miles) de lenguajes en uso en este planeta. Tales lenguajes son llamados *lenguajes naturales*. Muchas teorías interesantes se han enunciado sobre los orígenes del lenguaje en sí mismo. Una cuestión interesante a considerar es: ¿puede la gente comunicarse con las computadoras usando lenguajes humanos (naturales)? En otras palabras, ¿puede hacerse una computadora para que comprenda un lenguaje? Considérenlo por un momento.

Para que la cuestión de la comprensión del lenguaje resulte más concreta, piensen en su robot Scribbler. Hasta ahora, han controlado el comportamiento del robot escribiendo programas Python para el mismo. ¿Es posible hacer que el Scribbler comprenda nuestro lenguaje (castellano o inglés) de modo tal que puedan interactuar con él? ¿Cómo sería una interacción con el Scribbler? Obviamente, no esperarían tener una conversación con el Scribbler acerca de la cena que comieron anoche. Sin embargo, probablemente tendría sentido pedirle que se mueva en cierta dirección. O preguntarle si está viendo un obstáculo frente a él.

Realizar la siguiente actividad: Escriban una serie de comandos breves de una palabra como: `forward`, `right`, `left`, `stop`, (adelante, derecha, izquierda, detenerse) etc. Creen un vocabulario de comandos y luego escriban un programa que ingrese un comando a la vez, lo interprete y haga que el Scribbler lo lleve a cabo. Por ejemplo:

```
Ustedes: forward
Scribbler: comienza a moverse hacia adelante
Ustedes: right
el Scribbler empieza a moverse hacia la derecha
Ustedes: stop
…
```

Experimenten con el comportamiento del robot basándose en estos comandos y reflexionen acerca de la interpretación adecuada que pueda hacer que este comportamiento sea más natural.

Se encontrarán haciendo varias hipótesis acerca de la interpretación de incluso los comandos más simples en el ejercicio de arriba. Por ejemplo, ¿qué sucede cuando después de que le ordenan al Scribbler que se mueva hacia delante, le piden que se mueva a la derecha? ¿El Scribbler debería dejar de avanzar o debería detenerse y luego doblar?

Las decisiones como éstas también nos dan una profunda conciencia sobre nuestras propias habilidades para comprender el lenguaje. También pueden ver que, como en el caso de la percepción visual, el procesamiento del lenguaje (o de un texto) comienza en un nivel muy primitivo: las palabras. Si la entrada (input) es el discurso, las unidades básicas son las señales eléctricas, quizás provenientes de un micrófono. Al igual que el procesar píxeles individuales para tratar de comprender los contenidos de una imagen, uno debe comenzar en un nivel bajo de representación para comenzar a comprender el lenguaje.

Los investigadores que trabajan en el campo de la *lingüística computacional* (o la *comprensión del lenguaje natural*) han propuesto muchas teorías acerca del procesamiento del lenguaje que forman la base de un modelo computacional para que el Scribbler comprenda un pequeño subconjunto del lenguaje inglés[1]. En esta sección examinaremos un modelo de éstos que está basado en el procesamiento de la sintaxis y la semántica de la interacción del lenguaje. Imagínense interactuando con el Scribbler utilizando el siguiente grupo de sentencias:

```
Ustedes: ¿Viste una pared?
```

1 Tengan en cuenta que en este libro se basa en una experiencia en un instituto de estados Unidos y, por lo tanto, en estas primeras experiencias se trabaja con el idioma inglés.

Scribbler: No

Ustedes: Hacé un Bip cada vez que veas una pared.
Ustedes: Doblá a la derecha cada vez que veas una pared a tu izquierda.
Ustedes: Doblá a la izquierda cada vez que veas una pared a tu derecha.
Ustedes: Movete durante 60 segundos.

[El robot Scribbler se mueve alrededor durante 60 seundos doblando cada vez que ve una pared. También hace un bip cada vez que ve una pared.]

Anteriormente, han escrito programas Python que llevan a cabo comportamientos similares. Sin embargo, ahora imagínense interactuando con el robot de la manera descripta. Desde una perspectiva física, imagínense que están sentados frente a la computadora y tienen una conexión Bluetooth con el robot. La primera pregunta que aparece es: ¿Están diciendo o tipeando los comandos de arriba? Desde la perspectiva de la IA, ambas modalidades son posibles: Podrían estar sentados frente a la computadora hablando a un micrófono; o podrían estar tipeando esos comandos en el teclado.

En primera instancia, necesitarían una capacidad de comprensión del lenguaje. Actualmente, pueden obtener software (comercial y *freeware*) que les permitirá hacerlo. Algunos de estos sistemas son capaces de distinguir acentos, entonaciones, voces femeninas o masculinas, etc. Efectivamente, la comprensión del lenguaje hablado es un campo fascinante de estudio que combina conocimientos de lingüística, procesamiento de señales, fonología, etc.

Pueden imaginar que el resultado de hablar a una computadora es un fragmento de texto que transcribe lo que han dicho. Por lo tanto, la pregunta que se le planteó al Scribbler arriba: *¿Ves una pared?* Deberá ser procesada y transcripta a un texto. Una vez que tengan el texto, es decir, un string “*¿Ves una pared?*” o en inglés: “`Do you see a wall?`”, éste puede ser posteriormente procesado o analizado para comprender el *significado* o el contenido del texto. El campo de la *lingüística computacional* provee muchas formas de análisis sintáctico y de extracción de significado de textos. Los investigadores en IA han desarrollado formas de representar el conocimiento en una computadora usando anotaciones simbólicas (por ej., la *lógica formal*). Al final, el análisis del texto resultará en un comando `getIR()` o `getObstacle()` para el robot Scribbler y producirá la respuesta mostrada arriba.

Nuestra meta de traer el escenario de arriba aquí es para presentarles varias dimensiones de la investigación en IA que pueden involucrar a personas de distintas disciplinas. En estos días, es completamente posible, incluso para ustedes, diseñar y construir programas de computación o sistemas capaces de interactuar con robots usando el lenguaje.

Juegos

En la historia temprana de la IA, varios científicos plantearon varias tareas desafiantes que, si son realizadas por computadoras, pueden ser usadas como forma de demostrar la viabilidad de la inteligencia de las máquinas. Era una práctica común pensar en los juegos en este campo. Por ejemplo, si una computadora pudiera jugar un juego como el ajedrez o las damas en el mismo nivel

o mejor que los humanos, podríamos convencernos de que es factible pensar en una computadora como un posible candidato para la inteligencia de la máquina. Algunas de las demostraciones iniciales de la investigación en IA incluyeron pruebas de modelos de computadoras para jugar varios juegos. El ajedrez y las damas parecían ser las elecciones más populares, pero los investigadores se han dado el gusto de examinar modelos de computadoras para muchos juegos populares: Poker, Bridge, Scrabble, Backgammon, etc.



En muchos juegos, ahora es posible que los modelos de computadoras jueguen en los más altos niveles de la práctica humana. En el ajedrez, por ejemplo, aunque los primeros programas fácilmente vencían a lo novatos en los '60, no fue sino hasta 1996 que un programa de ajedrez de una computadora IBM llamada Deep Purple venció al campeón mundial Gary Kasparov en un juego a nivel competición, aunque logró ganar la partida 4-2. Un año después, Deep Blue le ganó a Kasparov en un juego de 6 partidas, era la primera vez que una computadora vencía al mejor jugador humano en un juego de ajedrez de estilo clásico. Mientras que estos logros son dignos de halagos, también es claro ahora que la búsqueda de la inteligencia artificial no se responde necesariamente con juegos de computadora. Esto ha tenido como resultado un gran progreso en los sistemas de juegos de computadora y en la tecnología de juegos que ahora es una industria multi-millonaria.

Resulta que en muchos juegos tipo ajedrez, la estrategia para que juegue una computadora es muy similar. Tales juegos se clasifican como juegos-de-dos personas-suma-cero: dos personas/computadoras juegan como oponentes y el resultado del juego es que un jugador gana y el otro pierde, o es un empate. En muchos juegos de este tipo, la estrategia básica para dar el siguiente paso es simple: ver todos los posibles movimientos que tengo, y para cada uno, los posibles movimientos del otro jugador y así hasta el final. Entonces, rastrear hacia atrás desde la instancia de la victoria (o el empate) y hacer el siguiente movimiento basado en los resultados deseables. Pueden ver esto aún en juegos simples como Ta Te Ti, donde es fácil ver mentalmente hacia delante futuras movidas y luego tomar decisiones informadas sobre qué hacer a continuación. La mejor manera de comprender esto es realizando un programa que juegue de esta manera.

Ta Te Ti

También conocido como *Nudos y Cruces* (*Noughts and Crosses)* o *Abrazos y Besos*, el Ta Te Ti es un juego de chicos bastante popular. Desarrollaremos un programa que pueda ser usado para jugar este juego contra una persona. Casi cualquier juego de mesa puede ser programado usando el loop (iteración) básico mostrado abajo:

```
def jugar():
    # Inicializar tablero
    tablero = inicializarTablero()

    # Determinar quién mueve: por ejemplo, X mueve siempre primero
    jugador = 'X'

    # Mostrar el tablero inicial
    mostrar(tablero)

    # El juego
    while (not gameOver(tablero)):
        mover(tablero, jugador)
        display(tablero)
        jugador = oponente(jugador)

    # fin del juego, mostrar resultados
    winningPiece = ganador(tablero)

    if winningPiece != 'Tie':
        print winningPiece, "won."
    else:
        print "It is a tie."
```

La función de arriba puede ser usada para jugar una ronda de cualquier juego de mesa de dos personas. La variable `jugador` es el jugador (o la pieza) cuyo movimiento es el siguiente. Ya estamos usando la pieza de Ta Te Ti “X” en la función de arriba. Seis funciones básicas (resaltadas arriba) conforman los bloques de construcción básicos del juego. Para el Ta Te Ti, pueden ser definidos como:

1. **`InicializarTablero()`**: Devuelve un tablero nuevo que representa el inicio del juego. Para el Ta Te Ti, esta función devuelve un tablero vacío representando nueve cuadrados.

2. **`mostrar(tablero)`**: Muestra el tablero en pantalla para que el usuario lo visualice. La visualización puede ser tan simple o elaborada como deseen. Es bueno comenzar con el que más fácilmente puedan escribir. Más adelante lo harán más elaborado.

3. **`oponente(jugador)`**: Devuelve el oponente del actual jugador/pieza. En el Ta Te Ti, si el jugador es X, devolverá un O, y viceversa.

4. **`mover(tablero, jugador)`**: Actualiza el tablero haciendo un movimiento para el jugador. Si el jugador es el usuario, pedirá el movimiento del usuario. Si el jugador es la computadora, decidirá cómo hacer el mejor movimiento. Aquí es donde entra en escena la inteligencia.

5. **`gameOver(tablero)`**: Devuelve `True` si no hay más movimientos por hacerse, de lo contrario, devolverá `False`.

6. **`ganador(tablero)`**: Examina el tablero y devuelve la pieza ganadora o que el juego aún no ha terminado, o que hay un empate. En el Ta Te Ti, devolverá una X, un O, un blanco (representando que el juego aún no ha terminado), o un EMPATE.

Necesitaremos algunas funciones más para completar el juego pero estas seis forman en núcleo básico. Las escribiremos primero.

El juego en sí mismo consiste de nueve cuadrados que empiezan vacíos. Los jugadores eligen una pieza de juego: una “X” o una “O”. Asumiremos que “X” siempre va primero. Lo primero que debe hacerse entonces es diseñar una representación del tablero de juego. Usaremos la siguiente representación simple:

```
tablero = [' ',' ',' ',' ',' ',' ',' ',' ',' ']
```

Esta es una lista de 9 strings de un carácter. Observen que estamos usando esta representación lineal en lugar de la de 2-dimensiones.

Sin embargo, como verán, esta representación hace que sea más fácil llevar a cabo muchas manipulaciones para el juego. Durante el juego, el tablero de juego puede ser visualizado en su formato natural. Abajo, mostramos dos funciones: una crea un tablero nuevo cada vez que se la convoca, y una la muestra:

```
def inicializarTablero():
    # Representamos un tablero de 3x3 como una lista de 9 elementos.
    # Usaremos la siguiente numeración para ubicar una casilla
    #  0 | 1 | 2
    # ---|---|---
    #  3 | 4 | 5
    # ---|---|---
    #  6 | 7 | 8

    return [' ',' ',' ',' ',' ',' ',' ',' ',' ']


def mostrar(tablero):
    for i in range(0, 9, 3):
       if i > 0:
           print '---|---|---'
       print " %c | %c | %c "%(tablero[i],tablero[i+1],tablero[i+2])
    print
```

Una ventaja de escribir la función de visualización como se muestra es que nos da una forma rápida de crear y mostrar el juego. Más adelante, cuando terminan, pueden escribir una versión más elegante que permite visualizar el juego gráficamente (ver los ejercicios). Con las funciones de arriba, fácilmente podemos crear un nuevo tablero y mostrarlo de la siguiente manera:

```
tablero = inicializarTablero()
mostrar(tablero)
```

Determinar el oponente de una pieza dada es suficientemente simple:

```
def oponente(jugador):
  if jugador == "X":
```

```
    return "O"
  else:
    return "X"
```

A continuación, tenemos que escribir la función `mover`. Primero determina a quién le toca jugar. Si es el turno del usuario, tomará la jugada del usuario. Si no, determinará la mejor jugada para la computadora. Luego, efectivamente realizará la jugada. Como primer diseño, dejaremos que `mover` haga una elección al azar entre los posibles movimientos para la computadora. Luego, tomaremos una decisión más informada. La función `choice` está disponible en el módulo de librería `random`:

```
from random import choice
Vos = 'X'
Yo = 'O'

def mover(tablero, jugador):

    if jugador == Vos:       # movimiento del usuario?
        casilla = input("Ingresa tu movida: ") - 1
    else:                   # mi turno
        # el jugador es la computadora, se toma una elección aleatoria
        casilla = choice(movimientosPosibles(tablero, jugador))

    # ubica la pieza del jugador en la casilla elegida
    aplicarMovimiento(tablero, jugador, casilla)
```

Hemos establecido las variables globales `Vos` y `Yo` para piezas específicas. Esta es una simplificación provisoria. Más adelante pueden volver y reescribirlas como para que en cada juego el usuario pueda seleccionar su pieza. En el Ta Te Ti, X siempre se mueve primero. Entonces, haciendo que el usuario tenga la pieza X, asumimos que X siempre va primero. Nuevamente, más adelante podemos regresar y modificar esto (ver los Ejercicios). También observen que no estamos chequeado errores en el input del usuario para asegurarnos de que se introdujo una movida legal (ver Ejercicios).

El usuario introduce su movimiento ingresando un número del 1..9 donde la numeración de cuadrantes es como se muestra abajo. Esto es ligeramente diferente de nuestra numeración interna de cuadrantes y resulta más natural para la gente. En la función de arriba, restamos 1 del número de input para que lo ubique en el cuadrado apropiado en nuestro esquema interno.

```
 1 | 2 | 3
---|---|---
 2 | 5 | 6
---|---|---
 7 | 8 | 9
```

Nuevamente, por ahora hemos simplificado la interfaz. Los ejercicios sugieren cómo mejorar este diseño.

La función `mover` definida arriba requiere dos funciones adicionales (se muestran resaltadas). Estas son también funciones básicas para cualquier juego basado en tablero y se describen abajo:

7. **movimientosPosibles(tablero, jugador)**: Devuelve una lista de posibles movimientos para el jugador dado.

8. ***aplicarMovimiento(tablero, jugador, casilla)***: Dado un cuadrado específico y un jugador/pieza, esta función aplica el movimiento en el tablero. En el Ta Te Ti, todo lo que uno debe hacer es ubicar la pieza en el cuadrado dado. En otros juegos, como el ajedrez o las damas, hay muchas piezas que pueden ser removidas.

En el Ta Te Ti todos los cuadrados vacíos son posibles lugares donde un jugador puede moverse. Abajo, escribimos una función en la cual, dado un tablero, devuelve un listado de todos los posibles lugares donde una pieza puede ser ubicada:

```
def movimientosPosibles(tablero):
    return [i for i in range(9) if tablero[i] == ' ']
```

Arriba estamos usando *comprensiones de listas (list comprehensions)* (una característica de Python) para crear la lista de posibles movimientos. Pueden estar más cómodos escribiendo la siguiente versión:

```
def movimientosPosibles(tablero):
  movimientos = []

  for i in range(9):
     if tablero[i] == ' ':
        movimientos.append(i)

  return  movimientos
```

Para completar el programa de juego tenemos que escribir dos funciones más definidas arriba. La función `ganador` examina el tablero y determina quién ganó. Devuelve la pieza ganadora (una `'X'`, `'O'`) o el string `'Tie'`. En el caso de que el juego aún no haya terminado, devuelve un `' '`. Abajo, primero definimos todas las posiciones ganadoras en el Ta Te Ti, basadas en nuestra representación del tablero. A continuación, definimos la función para examinar el tablero.

```
# Estas ternas de casillas representan los casos ganadores
GANADORES = [[0, 1, 2],[3, 4, 5],[6, 7, 8],      # filas
            [0, 3, 6],[1, 4, 7],[2, 5, 8],       # columnas
            [0, 4, 8],[2, 4, 6]]                 # diagonales

def ganador(tablero):
    for win in GANADORES:
        posWinner = tablero[win[0]]
        if (posWinner != ' ' and
            posWinner == tablero[win[1]] and
```

```
            posWinner == tablero[win[2]]):
            return posWinner

    # No hay ganador aún, quedan casillas vacías?
    for i in range(9):
        if tablero[i] == ' ':
            return ' '

    # El tablero está lleno y no hay tres en línea
    return 'Tie'
```

Por último, la función `gameOver` puede ser escrita ya sea basándose en el hecho de que el ganador devuelve un `' '` cuando el juego aún no ha terminado, o podemos escribirlo usando `movimientosPosibles()` de la siguiente manera:

```
def gameOver(tablero):
    return len(movimientosPosibles(tablero)) == 0
```

Ahora tenemos todos los ingredientes para que un programa juegue al Ta Te Ti. La versión de arriba está simplificada de muchas maneras, y sin embargo contiene los elementos esenciales de jugar una ronda de Ta Te Ti.

Realizar la siguiente actividad: Implementen el programa de arriba y jueguen algunas partidas de Ta Te Ti. Cuando juegan contra una computadora, están anticipando posibles movimientos en el camino y realizan los propios movimientos con aquellos en mente.

Probablemente no tuvieron problemas en ganarle a la computadora en este juego. Esto se debe a que la computadora simplemente está eligiendo movimientos al azar entre una serie de posibles movimientos:

```
# el jugador es la computadora, se toma una elección aleatoria
casilla = choice(movimientosPosibles(tablero, jugador))
```

Esta es la línea en `mover` donde podemos poner alguna inteligencia en el programa. Sin embargo, la pregunta básica que debe hacerse es: ¿cuál de los posibles movimientos es el mejor para mí? Piensen esto por un momento. En el Ta Te Ti, se juega de dos maneras: defensivamente, como para no perder después de la siguiente jugada, u ofensivamente, para intentar ganar. Si la victoria es inminente, elegirán por supuesto hacer esa jugada, pero si no lo es (y tampoco es pérdida), ¿cómo eligen el movimiento? Intentaremos generalizar esta idea de manera que también sea aplicable a otros juegos de tablero.

Deleguemos la responsabilidad de encontrar el mejor movimiento a una función: `mejorJugada` que devolverá la mejor jugada de entre un grupo de posibles movimientos. Es decir, podemos cambiar la línea en movimiento de la siguiente manera:

```
# el jugador es la computadora, se toma una elección aleatoria
casilla = mejorJugada(tablero, jugador, movimientosPosibles(tablero, jugador))
```

Si `mejorJugada` hiciera una elección al azar (como arriba) podríamos escribirla como:

```
def mejorJugada(tablero, jugador, movimientos):
   return choice(movimientos)
```

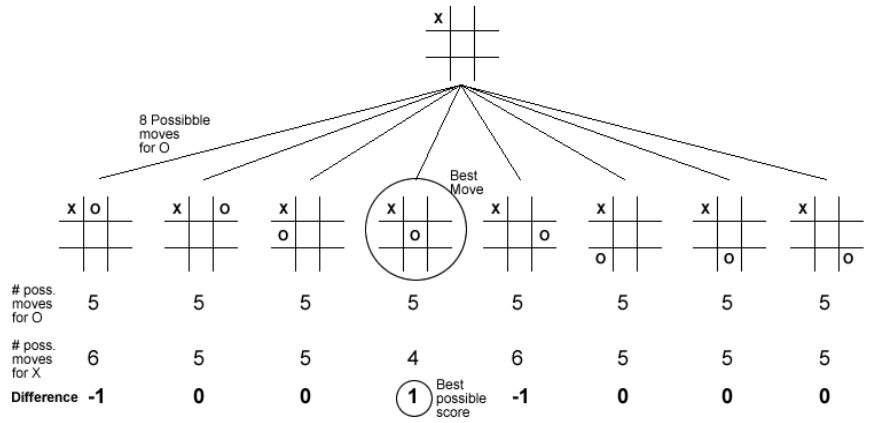
Imaginen que han decidido ir a ver a dos jugadores en una partida de ajedrez (o de Ta Te Ti). Sin embargo, llegan en el medio del juego. Entran en la habitación, miran el tablero y son capaces de evaluar cómo le está yendo a cada jugador sin saber nada de cómo se llegó a esa situación en el tablero. Por ejemplo, miren el siguiente tablero de Ta Te Ti:

```
 X | O | X       O | X | X       X | O | X       X |   |
---|---|---     ---|---|---     ---|---|---     ---|---|---
   | O |           |   |         O | X | O         |   |
---|---|---     ---|---|---     ---|---|---     ---|---|---
 O | X | X       X |   | O         | X |           |   |
     1               2               3               4
```

En todos los casos de arriba, la computadora está jugando con O y a continuación es su turno. Examinemos cada tablero y pensemos cuál sería el mejor movimiento para O.

En el caso 1, la computadora debe jugar defensivamente y ubicar una O en el cuadrado 6 para evitar perder el juego. En el caso 2, puede estar a la ofensiva y reconocer que ganará el juego ubicando una O en el cuadrado 5. En el caso 3, debe jugar para evitar perder. En el último caso, el tablero está ampliamente abierto. De las posibles ocho elecciones, ¿existe la mejor jugada? Por experiencia, probablemente vayan a ubicar al O en el centro del cuadrado (cuadrado 5). ¿Por qué? Elaboremos esto mirando hacia adelante (ver la figura en la siguiente página).

Piensen cómo podríamos cuantificar cada una de las posiciones del tablero arriba con respecto a las posibilidades de ganar que tiene O. Para cada tablero, cuenten el número de posibles posiciones ganadoras que le restan a O y compárenlas con las de X. Por ejemplo, en el primer tablero, O tiene 5 posiciones con posibilidades de ganar y X tiene 6. Podríamos decir que O es 5 – 6 = -1 tiene una posibilidad menos que X. Si hacen esto para el segundo y tercer tablero, verán que ambos X y O están parejos en la cantidad de posibles posiciones ganadoras restantes (5 cada uno). Sin embargo, en el cuarto caso, a O le restan 5 posibles posiciones ganadoras, y X tiene sólo 4. Intenten resolver las posiciones de tablero que quedan para confirmar los puntajes mostrados arriba. Claramente verán que ubicar a O en el cuadrado 5 es la mejor de todas las posibilidades. Intenten elaborar los 5 posibles movimientos para O en el caso 2 arriba. Verán que uno de los cinco conduce al triunfo para O. De modo similar, cuando elaboren las dos opciones para O en el caso 1, notarán que uno conduce a un empate y el otro a una derrota para O. Podemos capturar todas estas situaciones en una función que mire el tablero y devuelve un número que representa el puntaje a favor de O: cuanto más alto sea el valor devuelto, más deseable es el tablero.



$$\text{evaluar(tablero)} = \begin{cases} \infty, \text{ si el tablero es ganador para la computara} \\ -\infty, \text{ si el tablero es perdedor para la computadora} \\ \texttt{jugadasAbiertas}(\text{computadora}) - \texttt{jugadasAbiertas}(\text{usuario}), \text{ o/w} \end{cases}$$

Es decir que el triunfo para la computadora recibe un puntaje alto, una derrota recibe un puntaje bajo. De otra manera, es la diferencia en la cantidad de triunfos abiertos que restan para cada uno. Podemos capturar esto en Python de la siguiente manera:

```
INFINITO = 10
def evaluar(tablero):
    # si el tablero es ganador para el usuario, retornamos INFINITO
    pieza = ganador(tablero)

    if pieza == Yo:
        return INFINITO
    elif pieza == Vos:
        return –INFINITO
    else:
        return jugadasAbiertas(tablero,Yo) - jugadasAbiertas(tablero,Vos)
```

Definimos `INFINITO` como 10, un número lo suficientemente alto con relación a los otros valores que puede ser devuelto por `evaluar`. `jugadasAbiertas` mira a cada triple triunfo en la tabla y cuenta el número de aperturas para el jugador dado.

```
def jugadasAbiertas(tablero, jugador):
   possWins = 0

   for pos in wins:
      n = 0

      for i in range(3):
         if (tablero[pos[i]] == jugador) or
            (tablero[pos[i]] == ' '):
```

```
            n += 1
        if n == 3: possWins += 1

    return possWins
```

Realizar la siguiente actividad: Implementen las dos funciones de arriba y pruébenlas. Creen varias posiciones de tableros (usen las de los ejemplos de arriba) y confirmen que el tablero está siendo evaluado con los valores correctos.

A continuación, podemos reescribir `mejorJugada` para tomar ventaja de la evaluación:

```
def mejorJugada(tablero, jugador, movimientos):
    scores = []

    for mov in movimientos:
        b = copy(tablero)
        aplicarMovimiento(b, jugador, movimientos)
        scores.append(evaluar(b))

    return movimientos[scores.index(max(scores))]
```

Observen cómo toma cada posible jugada, crea una nueva tabla con esa jugada y luego evalúa el puntaje del tablero resultante. Finalmente, devuelve la jugada con el puntaje más alto como la mejor jugada. Modifiquen el programa de arriba para usar esta versión de `mejorJugada` y jugar varias veces. Podrán observar que hay una significativa mejora en la habilidad de juego de la computadora. ¿Pueden medirla de alguna manera? (Ver ejercicios).

La reescritura de arriba de `mejorJugada` hará que el programa juegue significativamente mejor, pero aún hay lugar para mejoras. En la mayoría de los juegos de mesa, los buenos jugadores son capaces de imaginar el partido anticipando varias jugadas. En muchos juegos, como el ajedrez, varias situaciones reconocibles conducen a resultados claramente determinados, por lo tanto, una gran parte de jugar una partida exitosa depende de la habilidad para reconocer esas situaciones.

Anticipar varias jugadas de manera sistemática es algo que las computadoras son bien capaces de hacer y por lo tanto cualquiera (¡aún ustedes!) pueden convertirlas en jugadores razonablemente buenos. El desafío reside en la cantidad de jugadas que pueden anticipar y en la capacidad limitada, si el tiempo para hacer la siguiente jugada es limitado, ¿cómo elegir entre las mejores opciones disponibles? Estas decisiones le dan un carácter interesante a los programas de juegos y continúan siendo una fuente importante de fascinación para muchas personas. Observemos cómo el programa de Ta Te Ti puede ver fácilmente todas las posibles movidas hasta el final del juego para determinar su siguiente movida (que en la mayoría de las situaciones conducen a empate, dada la simplicidad del juego).

Cuando uno mira hacia delante unas jugadas, tiene en cuenta que el oponente intentará ganar en cada jugada. El programa, al tratar de elegir la mejor jugada,

puede adelantarse a las del oponente para tenerlas en cuenta al elegir el mejor movimiento. De hecho, puede ir aún más lejos, hasta el final. La función `evaluar` que escribimos arriba puede ser usada con eficacia para evaluar futuras situaciones de tablero asumiendo que cuando es el turno de la computadora, siempre intentará elegir la jugada que prometa el mayor puntaje en el futuro. Sin embargo, al examinar los movimientos del oponente, debe asumir que el oponente realizará la jugada que resulte la peor para la computadora. En otras palabras, al mirar hacia delante, la computadora maximizará su posible puntaje mientras que el oponente minimizará las posibilidades de ganar de la computadora. Esto puede lograrse de la siguiente manera:

```
def mirarHaciaAdelante(tablero, jugador, MAX, nivel):
  movimientos = movimientosPosibles(tablero)

  if nivel <= 0 or len(movimientos)==0: # limite de mirarHaciaAdelante
     return evaluar(tablero)

  if MAX:                                          # movimiento de la computadora
     V = -INFINITO
     for m in movimientos:
        b = copy(tablero)
        b[m] = jugador
        V =max(V,mirarHaciaAdelante(b,oponente(jugador),1-MAX,nivel-1))
   else:                                           # movimiento del oponente
     V = INFINITO
     for m in movimientos:
        b = copy(tablero)
        b[m] = jugador
        V =min(V,mirarHaciaAdelante(b,oponente(jugador),1-MAX,nivel-1))
   return V
```

La función `mirarHaciaAdelante` definida arriba toma el tablero actual, el jugador al que le toca mover, ya sea la computadora (una intentando maximizar sus resultados) o la computadora (una intentando minimizar los resultados de la computadora), y los niveles aún por verse adelante, y calcula un puntaje basado en el examen de todas las jugadas hacia delante hasta el límite de la función *`mirarHaciaAdelante`*. En la función de arriba, cuando MAX es 1, representa a la computadora y 0 representa a su oponente. Por ende, dependiendo del valor de MAX, la evaluación es minimizada o maximizada respectivamente. Cada vez que mira más hacia adelante, el nivel se reduce por 1. El valor final devuelto por `mirarHaciaAdelante` puede ser usado por `mejorJugada` de la siguiente manera:

```
NIVEL = 9
def mejorJugada(tablero, jugador, movimientos):
   scores = []
   for m in movimientos:
      b = copy(tablero)
      b[m] = jugador
      scores.append(mirarHaciaAdelante(b,oponente(jugador),0,NIVEL-1)
   return movimientos[scores.index(max(scores))]
```

Como anteriormente, la jugada con el valor más alto es considerada la mejor jugada. Hemos determinado el valor de `NIVEL` arriba en 9 (por ejemplo, mirar 9 jugadas hacia delante), lo cual implica que cada vez mirará tan lejos como el final del juego antes de tomar la decisión. Sólo puede haber un máximo de 9 jugadas en un partido de Ta Te Ti. La calidad de la toma de decisiones de la computadora puede de hecho ser ajustada bajando el valor de `NIVEL`. En `NIVEL = 1`, será equivalente a la versión que escribimos atrás que sólo usaba la función `evaluar`.

Cuántos niveles hacia adelante se miren en un juego como éste depende del juego en sí mismo. ¿Pueden adivinar cuántas situaciones de tablero tendrá que ver la computadora al realizar una mirada hacia delante en `NIVEL = 9` después de la primera movida del usuario? ¡Serán 40,320 situaciones de tablero distintas! ¿Por qué? Además, para cuando es el momento de la segunda jugada de la computadora, sólo necesitará mirar 720 posiciones de tablero. Esto se debe a que en el Ta Te Ti, a medida que se llena el tablero, quedan menos posibles movimientos. De hecho, para cuando es el momento de la tercera jugada de la computadora, sólo necesita mirar un total de 24 tableros. Y, si la computadora llega a la cuarta jugada, sólo deberá ver dos posibles movimientos. Por lo tanto, en un exhaustivo examen de todas las posibles posiciones de tablero hasta el final de la partida cada vez que la computadora deba moverse, estará revisando un total de 41.066 posiciones de tablero. Sin embargo, si consideran un típico juego de ajedrez, en el cual cada jugador realiza un promedio de 32 movimientos y el número de jugadas factibles disponibles en cualquier momento es de alrededor de 10, pronto se darán cuenta de que la computadora debería examinar algo del orden de 10^{65} posiciones de tablero antes de realizar una jugada. Esto, aún para las computadoras más rápidas disponibles hoy en día tardaría mucho tiempo. Ampliaremos esto más adelante. Pero, para jugar un juego de dos personas-suma-cero, no es tan esencial mirar tanto hacia delante. Pueden variar la cantidad de jugadas anticipadas ajustando el valor de `NIVEL` en los programas.

Realizar la siguiente actividad: Implementen `mirarHaciaAdelante` como se describe arriba y comparen cuán bien juega la computadora contra ustedes. Traten de variar los niveles de 1, 2, ..., para ver si realmente hay mejoría en el juego de la computadora. ¿Considerarían a este programa inteligente?

Los ejercicios al final del capítulo los guiarán para transformar el programa de arriba en uno más robusto e incluso en un programa eficiente para el juego de Ta Te Ti. Sin embargo, estudien la estructura del programa cuidadosamente y serán capaces de usar la misma estrategia, incluyendo gran parte del corazón del programa para jugar muchos otros juegos de mesa de a dos personas.

Un Piedra Papel o Tijera más inteligente

En el Capítulo 7 han visto el ejemplo de un programa que jugó a Piedra Papel o Tijera contra un usuario humano. En esa versión, la estrategia de elección del programa para elegir un objeto era completamente al azar. Reproducimos esa sección del programa aquí:

```
...
items = ["Papel", "Tijera", "Piedra"]
```

```
...
# La computadora elije
miEleccion = items[randint(0, 2)]
...
```

En el segmento de programa de arriba, `miEleccion` es la elección del programa.

Como pueden ver, el programa usa un número al azar para seleccionar su objeto. Es decir, las posibilidades de elegir cualquiera de los tres objetos es de 0.33 o 33%. Las estrategias para jugar y triunfar aquí han sido extensamente estudiadas. Algunas estrategias dependen de detectar patrones en el comportamiento humano de selección. Aunque no nos demos cuenta, hay patrones en nuestro aparente comportamiento al azar. Los programas de computadora fácilmente pueden rastrear estos patrones de comportamientos manteniendo largas historias de decisiones de jugadores, detectarlas y luego diseñar estrategias para ganarle a esos patrones. Esto ha demostrado ser muy efectivo. Involucra grabar las decisiones del jugador, y buscar a través de ellas. Otra estrategia es estudiar las estadísticas humanas de elecciones en este juego. Antes de presentarles algunos datos, realicen el siguiente ejercicio sugerido abajo:

Realizar la siguiente actividad: Jugar el juego contra algunas personas. Jugar varias rondas. Registrar las elecciones realizadas por cada jugador (sólo escribir P/P/T en dos columnas). Una vez hecho esto, computen el porcentaje de veces en que cada objeto es elegido. Ahora continúen leyendo.

Resulta que la mayoría de los jugadores humanos tienden a elegir Piedra en lugar de Tijera o Papel. De hecho, varios análisis sugieren que el 36% del tiempo, la gente tiende a elegir Piedra, el 30% Papel, y el 34% Tijera. Esto sugiere que PPT no es meramente un juego de azar y que hay espacio para algunas estrategias para ganar. Lo crean o no, hay campeonatos mundiales de PPT todos los años. Aún un simple juego como éste tiene numerosas posibilidades. Podemos usar esta información, por ejemplo, para hacer que nuestro programa sea más inteligente, o más apto para jugar el juego. Todo lo que debemos hacer es en lugar de usar una posibilidad pareja de 33% de seleccionar cada objeto, podemos sesgar las posibilidades de selección basándonos en las preferencias de la gente. Por lo tanto, si el 36% del tiempo la gente tiende a elegir Piedra, sería mejor que el programa eligiera Papel el 36% de las veces dado que el Papel le gana a la Piedra. De modo similar, nuestro programa debería elegir Tijera el 30% de las veces para equiparar las posibilidades de ganarle al Papel, y elegir Piedra el 34% de las veces para equiparar las posibilidades de ganarle al Papel.

Podemos influir en el generador de número al azar usando los siguientes porcentajes:

```
Primero generar un número al azar en el rango 0..99
Si el número generado está dentro del rango 0..29, seleccionar Tijera (30%)
Si el número generado está dentro del rango 30..63, seleccionar Piedra (34%)
Si el número generado está dentro del rango 64..99, seleccionar Papel (36%)
```

La estrategia de arriba de influir en la selección al azar puede ser implementada de la siguiente manera:

```
def miSeleccion():
    # Primero, genero un numero aleatoria en el rango de 0..99
    n = randrange(0, 100)

    # Si n esta en el rango 0..29, elegimos Tijera
    if n <= 29:
        return "Tijera"
    elif n <= 63:
        # Si n esta en el rango de 30..63, elegimos Piedra
        return "Piedra"
    else:
        return "Papel"
```

Realizar la siguiente actividad: Modifiquen su programa de PPT del capítulo 7 para usar esta estrategia. Jueguen el juego varias veces. ¿Funciona mejor que en la versión previa? Deberán testear esto recolectando datos de las dos versiones contra varias personas (¡asegúrense de que sean novatos!).

Otra estrategia que utiliza la gente se basa en la siguiente observación:

Luego de varias rondas, la gente tiende a hacer la jugada que le hubiera ganado a su movida previa.

Digamos que un jugador selecciona Papel. A continuación elegirá Tijera. Un programa de computadora o un jugador que esté jugando contra éste entonces debería elegir Piedra para ganarle a Tijera. Dado que la relación entre las elecciones es cíclica, la estrategia puede ser implementada eligiendo lo que le gana a lo que le gana a la jugada anterior del oponente. El Papel le gana a la Piedra. Por lo tanto, dado que la jugada anterior del jugador fue Papel, el programa puede elegir Piedra, anticipándose a la elección de Tijera por parte del jugador. Intenten pensar esto cuidadosamente, asegurándose de que la cabeza no les esté dando vueltas al final. Si el jugador puede detectarlo, puede usarlo como una estrategia ganadora. Dejaremos la implementación de esta última estrategia como un ejercicio. Los ejercicios también sugieren otra estrategia.

El punto de los ejemplos de arriba es que utilizando estrategias en sus programas pueden hacer que sean más inteligentes. Deliberadamente, hemos empezado a usar el término inteligencia de modo un poco más suelto que lo que Alan Turing sugiere en su famoso ensayo. Muchas personas argumentarán que estos programas no son inteligentes en el sentido último del término. Estamos de acuerdo. Sin embargo, escribir programas más inteligentes es una actividad natural. Si los programas incorporan estrategias o heurísticas que las personas usarían cuando realizan alguna actividad, entonces los programas tienen alguna forma de inteligencia artificial. Aún si la estrategia utilizada por el programa no tiene nada que ver con lo que usaría la gente, pero haría al programa más inteligente o mejor, lo llamaríamos inteligencia artificial. Muchas personas estarían en desacuerdo con esta observación última. Para algunos, la búsqueda por resolver la inteligencia se limita a comprenderla en los humanos (y otros animales). En la IA ambos puntos de vista prevalecen y generan debates apasionantes entre académicos.

Discusión

La sola idea de considerar a una computadora como un dispositivo inteligente tiene sus orígenes en la naturaleza del propósito general de las computadoras. Al cambiar el programa, se puede lograr que la misma computadora se comporte de maneras diferentes. En el fondo, una computadora es sólo un manipulador de símbolos: manipula codificaciones de números o letras o imágenes, etc. Se postula que también el cerebro humano es un manipulador de símbolos. Las bases de la IA residen en el hecho de que la mayoría de los sistemas inteligentes son símbolos físicos y dado que una computadora es un manipulador de símbolos de propósitos generales, puede ser usada para estudiar o simular la inteligencia.

Revisión Myro

No se introdujeron nuevas funciones Myro en este Capítulo.

Revisión Python

choice(LISTA)
Devuelve un elemento seleccionado al azar de LISTA. Esta función se importará del módulo random.

List Comprehensions
Una forma breve y elegante de construir listas. Ver la documentación Python para más información.

Ejercicios

1. Leer el artículo de Alan Turing “Computing Machinery and Intelligence” (Máquinas Computadoras e Inteligencia). Fácilmente lo podrán hallar buscando en la Web.

2. Realicen una búsqueda en la Web de “Argumento de la habitación china de Searle” para ubicar los argumentos del filósofo John Searle acerca de que no importa cuán inteligente sea una computadora o programa, nunca podrá tener una ¨mente¨.

3. Reescriban la vista del juego de Ta Te Ti para ver el tablero gráficamente.

4. Diseñen un lenguaje de comandos de una palabra para el Scribbler. Escriban un programa para ingresar un comando a la vez, interpretarlo y luego hacerlo correr al comando en el Scribbler.

5. Extiendan el lenguaje del Ejercicio 4 para incluir consultas (por ej. ¿pared?) y luego modifiquen el programa para incorporar estas consultas.

6. Realicen una investigación sobre sistemas de comprensión del habla.

7. Realicen una investigación sobre lingüística computacional.

8. En el programa de Ta Te Ti diseñado en este Capítulo, asumimos que el usuario siempre juega con una X. Modifiquen el programa para que le dé la posibilidad de elegir al inicio del juego. Más adelante, al final del juego, las piezas se intercambian.

9. En la función mover definida para el Ta Te Ti, el programa acepta lo que sea que el usuario ingrese como movimiento. Prueben el programa y en lugar de ingresar un movimiento válido, ingresen su nombre. ¿Qué sucede? Tales errores pueden ser fácilmente detectados, dado que detendrán la ejecución del programa. Sin embargo, a continuación intenten ingresar un número del 1 al 9 usando una posición de repuesto que ya esté ocupada en el tablero. ¿Qué ocurre? Modifiquen la función para aceptar sólo los movimientos correctos. Si el usuario ingresa una movida incorrecta, el programa debería señalarlo y darle al usuario otra oportunidad.

10. La función gameOver puede hacer uso de la función ganador para tomar esta decisión en el programa de Ta Te Ti. Reescriban gameOver para hacer esto.

11. Una manera de medir cómo se compara una estrategia contra otra es jugarla contra otra estrategia una y otra vez registrando la cantidad de veces que se gana, se pierde y se empata. Modifiquen su programa de Ta Te Ti o PPT para reemplazar la segunda estrategia para el usuario (en lugar de tomar la entrada (input) del usuario, usa la función que implementa la segunda estrategia). Agreguen enunciados para jugar el juego muchas veces (digamos 1000) y registren las veces que se gana, se pierde y se empata. También querrán suprimir toda salida (output) de tablero dado que el juego se está jugando enteramente dentro del programa.

Realicen una comparación entre las estrategias discutidas en este Capítulo y cómo se comparan jugando entre sí.